UNIVERSITÉ DU LUXEMBOURG

PhD-FSTC-2013-32
The Faculty of Science, Technology and Communication

DISSERTATION

Defense held on the 5th Novembre 2013, in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN **Informatique**

by

Moussa AMRANI
Born on 18th November 1977 in Annecy (Haute-Savoie, France)

TOWARDS THE FORMAL VERIFICATION OF MODEL TRANSFORMATIONS: AN APPLICATION TO KERMETA

Dedicated to my Mom, Zineb, who left us too soon, leaving this hole in our heart that is so difficult to fill.

Abstract

Model-Driven Engineering (MDE) is becoming a popular engineering methodology for developing large-scale software applications, using models and transformations as primary principles. MDE is now being successfully applied to domain-specific languages (DSLs), which target a narrow subject domain like process management, telecommunication, product lines, smartphone applications among others, providing experts high-level and intuitive notations very close to their problem domain. More recently, MDE has been applied to safety-critical applications, where failure may have dramatic consequences, either in terms of economic, ecologic or human losses. These recent application domains call for more robust and more practical approaches for ensuring the correctness of models and model transformations.

Testing is the most common technique used in MDE for ensuring the correctness of model transformations, a recurrent, yet unsolved problem in MDE. But testing suffers from the so-called coverage problem, which is unacceptable when safety is at stake. Rather, exhaustive coverage is required in this application domain, which means that transformation designers need to use formal analysis methods and tools to meet this requirement. Unfortunately, two factors seem to limit the use of such methods in an engineer's daily life. First, a methodological factor, because MDE engineers rarely possess the effective knowledge for deploying formal analysis techniques in their daily life developments. Second, a practical factor, because DSLs do not necessarily have a formal explicit semantics, which is a necessary enabler for exhaustive analysis.

In this thesis, we contribute to the problem of formal analysis of model transformations regarding each perspective. On the conceptual side, we propose a methodological framework for engineering verified model transformations based on current best practices.

For that purpose, we identify three important dimensions: (i) the transformation being built; (ii) the properties of interest ensuring the transformation's correctness; and finally, (iii) the verification technique that allows proving these properties with minimal effort. Finding which techniques are better suited for which kind of properties is the concern of the Computer-Aided Verification community. Consequently in this thesis, we focus on studying the relationship between transformations and properties.

Our methodological framework introduces two novel notions. A transformation intent gathers all transformations sharing the same purpose, abstracting from the way the transformation is expressed. A property class captures under the same denomination all properties sharing the same form, abstracting away from their underlying property languages. The framework consists of mapping each intent with its characteristic set of property classes, meaning that for proving the correctness of a particular transformation obeying this intent, one has to prove properties of these specific classes.

We illustrate the use and utility of our framework through the detailed description of five common intents in MDE, and their application to a case study drawn from the automative software domain, consisting of a chain of more than thirty transformations.

On a more practical side, we study the problem of verifying DSLs whose behaviour is expressed with Kermeta. Kermeta is an object-oriented transformation framework aligned with Object Management Group standard specification MOF (Meta-Object Facility). It can be used for defining metamodels and models, as well as their behaviour. Kermeta lacks a formal semantics: we first specify such a semantics, and then choose an appropriate verification domain for handling the analysis one is interested in.

Since the semantics is defined at the level of Kermeta's transformation language itself, our work presents two interesting features: first, any DSL whose behaviour is defined using Kermeta (more precisely, any transformation defined with Kermeta) enjoys a de facto formal underground for free; second, it is easier to define appropriate abstractions for targeting specific analysis for this full-fledged semantics than defining specific semantics for each possible kind of analysis.

To illustrate this point, we have selected Maude, a powerful rewriting system based on algebraic specifications equipped with model-checking and theorem-proving capabilities. Maude was chosen because its underlying formalism is close to the mathematical tools we use for specifying the formal semantics, reducing the implementation gap and consequently limiting the possible implementation mistakes. We validate our approach by illustrating behavioural properties of small, yet representative DSLs from the literature.

Keywords: Model-Driven Engineering, Domain-Specific Languages, Formal Semantics, Formal Verification, Kermeta, Maude.

Résumé

L'ingénierie des Modèles (IDM) est devenu ces dernières années une méthodologie de développement logiciel populaire pour gérer de larges applications, sur la base de modèles et de transformations de modèles. L'IDM est désormais appliquée aux Langages Dédiés (LDs): ces langages ont pour but de s'attaquer à des domaines restreints, comme la gestion de processus, les télécommunications, les lignes de produits ou le développement de logiciels embarqués dans les *smartphones*, pour fournir aux experts des notations proches de leur domaine d'expertise et à un haut niveau d'abstraction. Plus récemment, l'IDM a été appliquée avec succès pour développer des applications critiques, pour lesquelles les pannes sont susceptibles d'avoir des conséquences dramatiques en terme d'économie, d'écologie ou encore de pertes humaines. L'application récente de l'IDM à ces domaines sensibles appelle à plus de robustesse dans le développement du logiciel, mais aussi à développer des techniques d'analyse spécifiques pour assurer la correction des modèles et des transformations entrant en jeu.

La validation des transformations de modèles est un problème récurrent, mais qui n'a pas encore trouvé de solution convenable. Le Test est la méthode la plus répandue pour répondre à ce problème, mais cette technique souffre du problème dit de couverture, qui pose problème lorsque la sécurité devient un enjeu crucial. En fait, la couverture exhaustive devrait être la norme dans ce type de domaines : pour répondre à cette exigence, les ingénieurs écrivant des transformations devraient utiliser des méthodes et outils formelles. Malheureusement, leur utilisation se heurte à deux barrières. La première est méthodologique : les ingénieurs possédant rarement les connaissances nécessaires à la mise en œuvre de ces techniques, il leur est difficile de les déployer dans leur contexte de travail quotidien. La seconde est davantage pratique : les LDs n'ayant que rarement une sémantique explicitement écrite ou formalisée, il devient difficile de mettre en œuvre ces techniques sans ce prérequis incontournable.

Dans cette thèse, nous contribuons au problème de l'analyse formelle de transformations de modèles de chacun de ces deux points de vue: au niveau conceptuel, nous proposons un cadre méthodologique basé sur les meilleurs pratiques en la matière, pour guider les ingénieurs dans leur tâche de vérification formelle des transformations.

Pour cela, nous avons identifié trois dimensions importantes : (i) la transformation en cours de validation ; (ii) les propriétés de la transformation nécessaire pour prouver sa correction ; et (iii) la technique de vérification à mettre en œuvre pour effectivement prouver ces propriétés. Trouver la technique la mieux adaptée à chaque type de propriété est le champ de recherche de la communauté travaillant sur la Vérification Assistée (*Computer-Aided Verification*). Dans cette thèse, nous nous intéressons à mettre en exergue les relations entre transformations et propriétés.

Notre cadre formel introduit deux concepts nouveaux. L'*intention* d'une transformation réunit sous une même notion l'ensemble des transformations partageant le même but, le même type de manipulation de modèles indépendemment de la manière dont est exprimée cette transformation. Les *classes* de propriétés caractérisent sous une même dénomination toutes un ensemble de propriétés partageant la même expression mathématique, mais indépendemment du langage dans lequel seraient exprimées ces propriétés. Le cadre formel devient alors un *mapping* caractérisant chaque intention par un ensemble de propriétés caractéristiques de cette intention, dont la preuve conduirait à la validation des transformations regroupées sous cette intention.

Nous illustrons l'usage et l'utilité de ce cadre méthodologique au travers d'une description détaillée de cinq intentions au sein de l'Idm, et nous l'appliquons sur une étude de cas inspirée du domaine automobile qui consiste en une chaîne de transformation d'une trentaine de transformations.

Au niveau pratique, nous étudions le problème de la vérification formelle de LDs écrits en Kermeta, un moteur de transformation orienté object aligné sur Mof (*Meta-Object Facility*), le standard de l'Omg pour la métamodélisation. Kermeta peut être utilisé non seulement pour les activités de modélisation classique (spécifier un métamodèle et des modèles qui s'y conforment), mais aussi pour spécifier leur comportement ou, si ces métamodèles représentent des LDs, leur sémantique comportementale. Malheureusement, le langage Kermeta n'est pas formellement spécifié, ce qui pose problème pour brancher des outils de vérification. Nous avons donc commencé par formaliser un sous-ensemble de Kermeta, suffisant pour représenter les transformations les plus courantes définies ; puis nous avons sélectionné un domaine de vérification adéquat pour fournir des capacités d'analyse.

Cette sémantique est définie au niveau du langage de transformation de Kermeta, ce qui a deux avantages : premièrement, tout LD dont le comportement serait défini à l'aide de Kermeta (ou plus précisément, toute transformation définie en Kermeta) voit sa sémantique définie formellement ; et deuxièmement, il devient plus facile, sur la base de cette sémantique de référence pour Kermeta, de définir des abstractions précises utiles pour documenter la traduction des transformations Kermeta vers de nouveaux outils de vérification. Au final, il devient plus facile pour les ingénieurs de vérifier leurs transformations : au lieu de définir une translation spécifique vers chaque nouvel outil d'analyse dont ils ont besoin, pour chacun de leurs LDs, il ne devient nécessaire de définir qu'un seul de ces *mappings* directement depuis le langage de Kermeta.

Nous avons illustré cette contribution technique à l'aide de Maude, un puissant moteur de réécriture de spécifications algébriques proposant deux types d'analyse : du *model-checking* et du *theorem-proving*. Nous avons choisi Maude parce que le formalisme sous-jacent est très proche des outils mathématiques utilisés dans la spécification formelle de la sémantique du langage, ce qui permet de réduire la distance conceptuelle entre la spécification et l'implémentation, limitant ainsi les problèmes d'implémentation. Nous avons validé notre approche en illustrant notre approche sur des LDs simples, mais cependant représentatifs.

Mots-clés: Ingénierie des Modèles, Langages Dédiés, Sémantique Formelle, Vérification Formelle, Kermeta, Maude.

Acknowledgement

This Thesis is the concrete and tangible outcome of a long trip: it started a few years ago in Grenoble, Isère (France) and finally ended here in Luxembourg. It was not easy, to say the least: the writing, and more importantly the intensive labour that led to this manuscript, were simultaneously a fight against odds and bad things, an initiation for research and all the political game behind it, and a spiritual journey when discovering and learning new stuff.

I'll start by warmly thanking Pierre KELSEN, without who nothing of this would have even been possible. He took a huge risk out of his usual environment and comfort zone for supervising this Thesis. Yves LE TRAON was basically the architect of this work, by suggesting very early some of the research directions followed in this Thesis. I owe him a lot for letting me participate to his SerVal Team. More importantly, he basically let me doing almost whatever I wanted during the second part of my Ph.D. time, which resulted in many trips, collaborations, and fruitful ideas. After all these years, I warmly thank them for letting me achieve my vision.

Benoît COMBEMALE influenced this thesis in a special way. More than a scientific mentor, he supported me when I encountered difficulties, both in my professional and personal life. Back in 2010 when I started to work on Kermeta, he kindly answered my (sometimes stupid) questions without being bored despite his busy agenda. It was a pleasure and an honour to have him judge my work and to attend *tant bien que mal* my defense (oh! yes, these trips back to Rennes will always be remembered). As he demonstrated during the almost-one-hour discussion during the defense Q& A session, he always has a very insightful viewpoint about all the core topics of my Ph.D. I really hope we can continue to work together.

I would also like to thank the other members of my jury. Nicolas NAVET always showed an interest in my work during our discussions. He also partially inspired the RT-Kermeta perspective thanks to his former background. Pierre-Yves SCHOBBENS is the professor kind I always liked, the "*force tranquille*": a huge amount of knowledge packed within a very humble man. I hope he didn't suffer much reading my semantics specification! I really hope we can pursue a common road together.

The journey was not always as lonely as one might think. Mostly in the LASSY team, then in the SerVal one, I am grateful to all Post-Docs, Ph.D. students or Research Associates who helped installing a good working environment, and contributed to overcome the bad feelings that sometimes occur in Research. In LASSY, Núno AMALIO walked with me a few steps along my journey and made me discover Z. Christian GLODT, Qin MA and Shahed PARNIAN were always present for discussing, sharing a coffee or just chatting in the corridor or outside the building. Later on at SerVal, Kevin ALLIX, Alexandre BARTEL, Donia EL KATEB and Christopher HÉNARD made me enjoy lunches and the epic SerVal coffee breaks for which I was always delighted to sometimes find *croissants* when one of them succeeded at getting a paper accepted. I also thank Iram RUBAB, Assad MOAWED and later on, Li LI, for the always interesting discussions at the office. This list cannot end without my office mate Marwane EL KHARBILI, with who I shared more than a Ph.D., a real *tranche de vie*: we were sharing a past, we then shared a destiny. More than an office mate, he was a companion of my pain and complaints, and was always there for "going to take a coffee" (and never actually *having* a coffee) with me when I wasn't in the mood of working. Our many discussions, scientific or not, made me appreciate him beyond what we were sharing at that time. He also opened the path towards successfully graduating and convinced me that it was doable, after all.

Some Post-Docs played a crucial role in my Ph.D. Starting with Yehia Elrakaiby who offered me my first Conference publication. Gilles Perrouin was *the man from the shadow*: always a good word to comfort me, always an efficient help to solve problems, but never out of (sometimes very good) ideas to exploit, he is actually the first one who suggested me to work on the verification of Kermeta, and I took his advice. My debt to him cannot be evaluated and he is still largely influencing my professional life with his advice. Would we be colleagues in the future?

Some people show up in your life and change it so deeply that you can say there is a before and an after their meeting. I had to wait until the first one left the Lassy Lab. before we could actually work together: Lévi Lúcio introduced me to the field of Mde verification, supported me in my dark days at Lassy, was always kind to review (he's the first one to have read the semantics specification) my production and advise me all along my Ph.D. He listened to my sometimes crazy ideas and knew how to exploit them the best way to make something good of them. More than a colleague, he also became a true friend. I cannot forget the visit to Montréal, and before that to Cascais inside his family. Me deepest recognition and friendship goes to him. I can only hope Little Valentin will follow the footsteps of his parents!

I have not enough words to thank Francisco (Paco) Durán, Professor at the University of Málaga (Spain). A kind man with an unlimited knowledge of Maude, *rewriting and all that*, very humble, very simple…He convinced me that I could make it while enjoying working with Maude ("the best language ever"), and was there every day, like a coach, to push me forward. I enjoyed each and every of our discussions and look forward to eat another *turrón* ice cream with him. Thanks to him, but also all the students in Lab 3.3. (Loli Burgueño, Javier Troya and Antonio Moreno Delgado), I spent in Málaga probably the best period of my Ph.D ever: I was working like a fool, but have enjoyed the sun, the beach and the never-sleeping city center. This period was a blessing, not only for my Ph.D but also for my psychological wellness. Paco also reconciled me with the *ethics* of professorship, showing me that it was possible to be one without completely sacrificing what I believe this job is.

I would like to thank the secretaries that always did a tremendous job at solving my administrative issues: Danièle Flammang at Lassy, Fabienne Schmitz at the Csc Department, and Laurent Bétry and Christine Kinet at the SnT.

This list would not be complete without the people that I met and worked with at Grenoble (France). David Merchat, Lionel Morel and Cyril Pachon were my student fellows during my Licence, Maîtrise and Dea back then, and started the adventure at Verimag with me. I learned a lot thanks to them, their solicitations, their experience. Liana Lazar Bozga, Anahita Akhavan enlighted our days by their smile, their kindness and their snacks. I am grateful to Chaker Nakhli, who switched with me his funding and graduated successfully before me, and Moez Krichen my office mate for so many years, who taught me a lot about life. I cannot forget all the top-notch professors of Verimag: Florence Maraninchi and Fabienne Lagnier Carrier, Laurent Mounier and Jean-Claude Fernandez who wwere my teachers and became colleagues, before teaching me again how to teach; Marius Bozga, always kind and willing to help; and of course Saddek Bensalem and especially Yassine Lakhnech, who forced me to learn formal verification on my own.

Enfin pour terminer, j'aimerais remercier ma famille, et en particulier mon père, Bachir, qui aura vécu une vie de sacrifices et de galères pour offrir à ses enfants la possibilité d'étudier du mieux qu'ils pouvaient. J'espère que ce titre de Docteur apaisera sa soif de comparaison avec les "enfants des autres". Les longues heures au téléphone avec mes soeurs, Malika et Dalila, m'auront aussi aidé à traverser cette épreuve qu'était la (les) thèse(s), mais aussi plus largement l'épreuve de la vie. Enfin pour terminer, à celle qui se sera pris de plein fouet la difficile épreuve de la confrontation dès son arrivée à Thionville, et qui aura supporté du mieux qu'elle pouvait ces années de travail acharné entrecoupées de nombreux voyages, celle qui aura vu de l'intérieur mes déprimes, mes changements d'humeur, mes joies, mes déceptions, Fatima, ma femme, pour qui je n'ai pas assez

de mots dans la langue française pour lui exprimer mon amour, mais aussi ma dette envers elle après toutes ces années si loin de tout. Peut-être que le silence qu'elle déteste tant est encore la meilleure réponse ?

This work was partially supported by the *Fonds National de la Recherche* in Luxembourg, but also from institutional fundings of my supervisors, who always allowed me to travel wherever I wanted.

Contents

List of Figures **viii**

List of Tables **ix**

1 Introduction **1**

1.1 Model-Driven Engineering . . . 1
1.2 Software Validation . . . 2
1.3 Towards the Formal Analysis of Model Transformations . . . 3
1.4 Contributions . . . 3
1.4.1 A Description Framework for Model Transformation Intents . . . 4
1.4.2 A Formal Specification of Kermeta . . . 4
1.4.3 KMV: a Proof-of-Concept Kermeta Model-Checker . . . 5
1.5 Thesis Outline . . . 5

I Models, Model Transformation & Model Transformation Verification **7**

2 Model-Driven Engineering: (Meta-)Models & Transformations **11**

2.1 What is a Model? . . . 11
2.1.1 Model Kinds . . . 12
2.1.2 Instantiation . . . 13
2.1.3 (Meta-)Metamodels . . . 15
2.1.4 Discussions . . . 16
2.2 Model Transformations . . . 17
2.2.1 Definitions . . . 17
2.2.2 Transformation Languages And Specifications . . . 18
2.2.3 Transformation Execution . . . 20
2.2.4 Model Transformation Classifications . . . 20
2.2.5 Conclusion . . . 23
2.3 Domain-Specific (Modelling) Languages . . . 23
2.3.1 DSML Features . . . 23
2.3.2 DSMLs as Languages: Basic Components . . . 26
2.4 Conclusion . . . 29

3 Formal Verification **31**

3.1 The Verification Problem . . . 31
3.1.1 Characteristics of Formal Verification Approaches . . . 32

3.1.2 Common Verification Techniques . . . 35
3.2 A Tridimensional Approach . . . 36
3.2.1 Transformations . . . 36
3.2.2 Properties . . . 37
3.2.3 Formal Verification Techniques . . . 43
3.3 Discussion . . . 45
3.3.1 Property Kind / Fv Technique (PK/FVT) . . . 45
3.3.2 Transformation / Fv Technique (T/FVT) . . . 47
3.3.3 Transformation / Property Kind (T/PK) . . . 47
3.3.4 Transformation Intent as the glue between dimensions . . . 48
3.4 Conclusion . . . 48

4 Characteristic Properties of Model Transformations Intents **49**
4.1 Overview & Motivation . . . 49
4.2 A Description Framework for Model Transformation Intents . . . 50
4.2.1 A Metamodel for Intents and their Properties . . . 50
4.2.2 A Metamodel for Model Transformation Validation Methods . . . 51
4.2.3 Usage Scenarios . . . 52
4.2.4 Outline . . . 54
4.3 The Intents Catalog . . . 54
4.4 Characteristic Properties . . . 57
4.4.1 Fundamental Property . . . 58
4.4.2 Property Classes . . . 59
4.5 Five Examples . . . 62
4.5.1 Query . . . 62
4.5.2 Refinement . . . 64
4.5.3 Translation . . . 66
4.5.4 Analysis . . . 69
4.5.5 Simulation . . . 71
4.6 The Power Window Case Study . . . 73
4.6.1 Formalism Transformation Graph and Process Model . . . 73
4.6.2 Description . . . 75
4.7 Identifying Transformation Intents within the Pwcs . . . 77
4.7.1 Translation . . . 77
4.7.2 Simulation . . . 79
4.7.3 Overview . . . 81
4.8 Discussion . . . 82
4.8.1 Towards an Intent Taxonomy . . . 82
4.8.2 Composing Intents . . . 84
4.9 Related Work . . . 84
4.9.1 Intents in Software Engineering . . . 84
4.9.2 Classifications of Model Transformations . . . 85
4.9.3 Classifications of Model Transformation Verification Approaches . . . 86
4.10 Summary . . . 86

II Formal Specification of Kermeta **87**

5 Kermeta in a Nutshell **91**
5.1 History . . . 91
5.2 Metamodelling: the Structural Language (SL) . . . 92
5.2.1 SL Meta-metamodel . . . 92
5.2.2 The FSM Metamodel . . . 94
5.3 Transformations: the Action Language (AL) . . . 95
5.3.1 AL Meta-metamodel . . . 95
5.3.2 The FSM Model & Behaviour . . . 96
5.4 Mathematical Background . . . 98
5.4.1 Functions . . . 99
5.4.2 Abstract Datatypes Specifications . . . 99

6 Structural Language **101**
6.1 Structural Semantics Overview . . . 101
6.2 Names, Types and Values . . . 102
6.2.1 Names . . . 102
6.2.2 Syntactic Types . . . 103
6.2.3 Semantic Values . . . 104
6.3 Metamodels . . . 104
6.3.1 Package . . . 105
6.3.2 Enumeration . . . 105
6.3.3 Class . . . 106
6.3.4 Property . . . 106
6.3.5 Operation . . . 107
6.3.6 Metamodel . . . 108
6.4 Models . . . 110
6.4.1 Accessible Features . . . 110
6.4.2 Model . . . 111
6.5 Conformance . . . 112
6.6 Discussions . . . 113
6.6.1 Design Choices . . . 113
6.6.2 Related Works . . . 114

7 Action Language **115**
7.1 Restrictions . . . 115
7.2 Definition . . . 115
7.2.1 Local Variable Declarations . . . 117
7.2.2 Syntax . . . 117
7.2.3 Control Flow . . . 118
7.2.4 Example . . . 119
7.3 Type-Checking System . . . 120
7.3.1 Expressions . . . 121
7.3.2 Statements . . . 122
7.4 Semantics . . . 124
7.4.1 Semantic Domain and Operations . . . 124
7.4.2 Configuration . . . 128
7.4.3 Semantic Rules . . . 129

7.5 Discussions . . . 131
7.5.1 Syntactic aspects & Restrictions . . . 131
7.5.2 Related Works . . . 132

III Formal Verification of Kermeta **133**

8 Maude In a Nutshell **137**
8.1 Equational & Rewriting Logic . . . 137
8.1.1 Functional Modules for Equational Theories . . . 137
8.1.2 System Modules for Rewrite Theories . . . 139
8.2 Object-Oriented Maude . . . 140
8.3 Example: The **SIMPLE** Language . . . 141
8.3.1 Syntax . . . 141
8.3.2 Semantics . . . 144

9 KMV: Verifying Kermeta with Maude **147**
9.1 Overview . . . 147
9.1.1 mOdCL & Maudeling . . . 147
9.1.2 Assumptions . . . 149
9.2 Specifying Kermeta's Structural Language . . . 150
9.2.1 Names, Types, and Values . . . 151
9.2.2 Metamodel . . . 153
9.2.3 Model . . . 157
9.2.4 Conformance . . . 159
9.3 Specifying Kermeta's Action Language . . . 160
9.3.1 Local Variables . . . 161
9.3.2 Statement Syntax . . . 161
9.3.3 Control Flow . . . 162
9.3.4 Configuration . . . 163
9.3.5 Semantics . . . 164
9.4 Simulating a Model Transformation . . . 169
9.5 The KMV Tool . . . 170
9.5.1 Eclipse Integration . . . 170
9.5.2 Limitations . . . 170

10 Conclusion **173**
10.1 Contributions Summary . . . 173
10.2 Perspectives & Future Work . . . 174
10.2.1 Description Framework . . . 174
10.2.2 Formal Semantics . . . 174
10.2.3 KMV . . . 176
10.2.4 Real-Time Kermeta . . . 177

A The Finite State Machine Example **179**
A.1 Metamodel and Model . . . 179
A.2 Kermeta Full Textual Definition . . . 180
A.3 Mathematical Representation . . . 181
A.3.1 The FSM Metamodel $\mathsf{MM}_{\mathsf{FSM}}$ . . . 181

A.3.2 The FSM Model $\mathsf{M}_{\mathsf{abc}}$ 182
A.3.3 Does $\mathsf{M}_{\mathsf{abc}}$ conform to $\mathsf{MM}_{\mathsf{FSM}}$? 182
A.4 Maude Representation 184

B Maude Specification for Kermeta Semantics **189**

C Summary of Publications **197**

Bibliography **199**

List of Figures

1.1 Software Engineering Evolution (from Kabore 2008) . . . 2
1.2 Mapping DSMLs to several, specialised verification domains: classical *versus* proposal approaches 3

2.1 Token *versus* Type Models . . . 12
2.2 Ontological *versus* Linguistic instantiation . . . 13
2.3 The OMG MDA Pyramid . . . 15
2.4 Formal construction for metamodel and model sets in MOF . . . 16
2.5 Model Transformation: the big picture . . . 18
2.6 Defining behavioural semantics for DSLs: translational *versus* operational . . . 22
2.7 Platform-Independant and Platform-Specific Models with Transformations . . . 25
2.8 Relations between DSML components: abstract/concrete syntax(es), semantic domain and semantic mapping . . . 27

3.1 Representation of the concrete semantics as a trajectory . . . 33
3.2 Representation of error zones . . . 33
3.3 Correct abstraction as an over-approximation . . . 33
3.4 Representation of Testing . . . 33
3.5 Erroneous abstraction . . . 33
3.6 Bounded Model-Checking . . . 33
3.7 False alarms . . . 33
3.8 Model Transformation Verification Tridimensional Approach . . . 37
3.9 The Tridimensional Classification revisited . . . 47

4.1 Intents as a classification mechanism for model transformations . . . 50
4.2 Metamodel for describing model transformation intents. . . . 51
4.3 Methods for validating model transformations. . . . 52
4.4 Identifying the intent of a model transformation . . . 53
4.5 Validating a model transformation with a specific intent . . . 53
4.6 Property Class Classification . . . 59
4.7 FTG/PM for the Power Window Case Study . . . 74
4.8 Safety Analysis FTG+PM Slice, with FTG on the left and PM on the right . . . 76
4.9 Example model for the *Environment DSL* language . . . 77
4.10 Example model for the *Encapsulated Petri Nets* language . . . 78
4.11 Syntactic property preservation example for the EnvToPN Power Window transformation . . . 79
4.12 Transformation Rules for simulating Petri Nets . . . 80

5.1 Kermeta as a Kernel for different standardised MDE languages. 92
5.2 Construction of Kermeta's Languages using weaving/promotion. 92
5.3 Simplified Kermeta's meta-metamodel 93
5.4 The Finite State Machine (FSM) DSL in Kermeta 94
5.5 Kermeta's Action Language Metamodel 97

6.1 Overview for the construction of the SL semantics 102

7.1 Kermeta's Action Language (from (Drey et al. 2009, §3.3)) 116
7.2 BNF for AL's Expressions and Statements 117
7.3 Simplification procedure illustrated on the *fire* operation 120
7.4 Functions computing default values for all collection types 125
7.5 Assignment of objects in a reference 127

8.1 The Bank Account Class Diagram and a simple Object Diagram. 140
8.2 Syntax for the SIMPLE 142
8.3 Semantics of the SIMPLE Language 144

9.1 Dependencies between mOdCL, Maudeling and KMV 148
9.2 Translation from Kermeta to Maude 150
9.3 Integration of KMV in Eclipse. 170

List of Tables

3.1 Classification of Contributions according to Property Kinds 42
3.2 Classification of verification techniques. 46

4.1 Query Intent Characterisation 63
4.2 Refinement Intent Characterisation 65
4.3 Translation Intent Characterisation 68
4.4 Analysis Intent Characterisation 70
4.5 Simulation Intent 72
4.6 Intents of transformations present in the Pwcs. 81
4.7 Model transformation examples from the Pwcs falling under the *Simulation* intent 82
4.8 Model transformation examples from the Pwcs falling under the *Translation* intent 83

7.1 Common representation of traditional `while`/`if-then-else` statements with `CondStmt`. 119

1 Introduction

1.1 Model-Driven Engineering

Over the past five decades, software researchers and developers have been constantly facing new challenges, forcing them to find new innovative solutions to lower development and maintenance costs, to improve software performance and reliability, and to better master software evolution. Each time a limit was reached, the software engineering community created new *paradigms*, imposing a new vision on how software applications and systems were managed and developed, which in turn helped mastering development costs and delays; they proposed new *methodologies* for designing and modelling these paradigms; and they developed new programming languages, better suited to their needs and alleviating better abstractions than their predecessors. Figure 1.1 shows how quickly paradigms shifted from procedural, modular conception in early '80 to models and model transformations nowadays.

Advances in programming languages and development platforms during the past two decades have significantly raised the software abstraction level at disposal for developers. However, it becomes harder to master the rapidly growing complexity of software: for example, Thales, a French multinational company delivering critical systems for aerospace, defense and transportation, noticed that the systems size they deliver to their clients is multiplied by a factor of 5 or 10 every five years (S. André 2004)! In these conditions, it becomes difficult to deliver such applications in a reasonable delay, with reasonable costs. One of the major source of this situation resides in the very nature of languages and platforms: they often have a computer-oriented focus that alleviates the solution space rather than capturing abstractions of the problem domain (D. C. Schmidt 2006).

Model-Driven Engineering (MDE) emerged in early 2000 as a promising approach to overcome third-generation general-purpose programming languages limitations. MDE promotes models and model transformations as first-class citizens for all the facets of the software activity. Among all MDE approaches, Domain-Specific Modelling (DSM) consists in focusing on the problem domain, be it technical- or business-oriented, rather than the possible technical solutions. DSM is supported by Domain-Specific (Modelling) Languages (DSMLs) that capture domain concepts, giving modellers the feeling to work with their usual notions. Models are then manipulated through model transformations to translate, analyse, generate code or documentations, extract relevant information, etc. However, to achieve this level of automation and reuse, DSMLs need to be precisely defined in order to enable machine manipulation, without sacrificing user readability.

After now more than a decade of intensive research, MDE tool support is reaching a maturity level that enables its use in many different application targets. Among others, some industrial fields have highly competitive markets, which requires mastering the software complexity by providing a higher automation level for repetitive tasks whenever possible. More importantly, these industrial applications like aeronautics, automotive or space-oriented applications must obey strict security requirements for ensuring that embedded softwares are correctly implemented.

Some industrial sectors already adopted tools and practices that share a lot with MDE new paradigms (for example, Airbus, the European aeronautics consortium, develops part of their embedded software using Scade,

	Before 1980	**1980 – 1995**	**1995 – 2000**	**2000 – Now**
Concepts	Procedure / Module	Objects / Classes	Framework / Patterns	(Meta-)model / Transformation
Language	Pascal, Fortran, C, ...	SmallTalk, Eiffel, C++, ...	Java, Ada, ...	ATL, QVT, Kermeta, ...
Methodology	Merise (1970) Yourdan (1970)	SSADM (1990) Booch (1992) OUM (1995)	UML (1996)	UML 2.0 (2000) MOF (2001)
Paradigm	Procedural Refinement	Object Messaging Object Composition		Model Transformation

Figure 1.1 – Software Engineering Evolution (from Kabore 2008)

a high-level synchronous language that automatically generates low-level code). However, to beneficiate from a better impact in such highly competitive industries, MDE has to grow to a full engineering discipline with accurate development methodologies, and tool support that encompasses all steps of the software development lifecycle.

1.2 Software Validation

The Software Engineering discipline traditionally distinguishes between two different concerns (Society 2004). On the one hand, *validation* is the process of building the *right system*, in a way that it meets the different stakeholders' desiderata. On the other hand, *verification* aims at building the *system right*, in a way that it complies with its specification, in order to eliminate, or at least sufficiently reduce, possible error flaws. While validation is made difficult by the natural changes occuring along a project, verification can be automatised to a large extent. Several techniques or approaches are nowadays available for determining if a system satisfies a given set of requirements. Among all of them, we will distinguish between two conceptually different ones that are commonly encountered in the context of these industrial developments, namely *testing* and *formal verification*.

Testing is performed on an actual implementation of a system Basically, the idea consists of feeding the system with some *inputs*, then, determining, by observing the resulting *outputs*, if the system behaves as expected. The main disadvantage of testing relies in the so-called "*coverage*" problem: supply adequate inputs to properly explore all possible execution paths is very hard, especially when subtle errors occuring only in highly exceptional cases become hard to reproduce. Several techniques try to overcome the limitations of testing, by automating the production of adequate inputs, by reaching a better ratio of coverage, and by automating the input/output comparisons. Among other popular approaches, we can cite model-based testing (Pretschner 2005; Selim, Cordy, and Dingel 2012b), and mutation testing (Jia and Harman 2010).

For safety-critical software however, testing is insuffcent, since certification standards are higher. Indeed, all computer scientists have experienced costly bugs in embedded software: everybody is familiar with the overflowed computation of the Ariane V maiden flight (Lions 1999); or the unit error of the Mars Orbiter (Stephenson et al. 1999), among others. Failures of this kind of systems may have huge financial and human losses.

In this context, but also every time a deeper confidence within software systems is required, more powerful techniques are necessary. Instead of working directly on the system at hand, formal verification techniques usually operate on a *model* of the system, and consist of mathematically proving the correctness of a system with respect to a set of requirements, called in this context *properties*. The main disadvantage of formal verification is the so-called *combinatorial* (or *state*) *explosion* problem: exhaustively exploring all possible paths is generally impossible, especially for infinite systems. Popular approaches for formal verification are model-checking (E. M. Clarke, Grumberg, and Peled 1999), theorem-proving (Duffy 1991), bounded satisfiability checking (Jackson 2011) and abstract interpretation (Boulanger 2011).

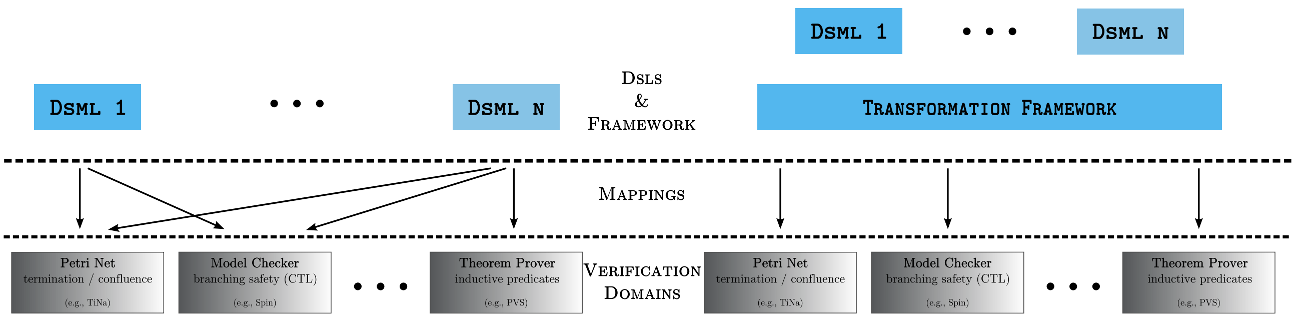

DSML 1
DSML N
DSMLs & Framework
DSML 1
DSML N
Transformation Framework
Mappings
Petri Net termination / confluence (e.g., TiNa)
Model Checker branching safety (CTL) (e.g., Spin)
Theorem Prover inductive predicates (e.g., PVS)
Verification Domains
Petri Net termination / confluence (e.g., TiNa)
Model Checker branching safety (CTL) (e.g., Spin)
Theorem Prover inductive predicates (e.g., PVS)

Figure 1.2 – Mapping DSMLs to several, specialised verification domains. On the left, the classical approach defines one mapping per DSML per domain; on the right, our approach defined only one mapping per domain at the level of the transformation framework.

1.3 Towards the Formal Analysis of Model Transformations

A natural idea would consist of adapting formal analysis techniques developed in the context of software engineering in general, to the specific area of MDE, especially for sensitive application domains. However, it is not as easy as one could expect.

The analysis of model transformations is complicated by the fact that models are richer structures than the ones traditionally manipulated by programming languages, ranging from simple data structures, to full syntactic representation of transformations themselves in the context of higher-order transformations (Tisi et al. 2009). Since MDE practitioners do not generally possess a strong knowledge in formal analysis, but have at the same time to work on critical industrial applications, a practical methodological guide could help them precisely target the tasks they have to perform in order to prove the correctness of their transformations.

Development and analysis time are sometimes difficult to conciliate: the former usually proceeds in MDE using "agile" methods, i.e. with iterative and incremental evolutions, constantly adapting to new requirements, a process facilitated by the transformations that automate part of the development; whereas formal analysis usually requires to have at disposal the entire application before applying verification techniques. It is possible to help, at some extent, transformation designers during the development with targeted static analysis techniques (by, for example, detecting that a transformation rule will never be applicable, or that a variable is never initialised), but more powerful techniques, especially when addressing semantic properties, requires transformations to be fully defined. It is then important to adopt development methodologies that help reducing the development costs, and at the same time allow to capitalise the knowledge and experience in formal analysis.

1.4 Contributions

In this Thesis, we contribute to the problem of formal analysis of model transformations regarding this double perspective:

1. For helping MDE practitioners ensuring the correctness of their model transformations, we propose a *Description Framework* that helps relating which properties are relevant to which kind of transformations.

2. For helping reducing development costs and capitalising formal analysis knowledge and experience, we propose define bridges towards verification domains directly at the level of the transformation framework instead. We demonstrate the feasability of this idea by applying it to Kermeta, a popular MOF-compliant object-oriented metamodeling framework, offering model-checking and theorem-proving capabilities.

Figure 1.2 compares the classical approach in the literature with our proposal. It is not realistic to hope to achieve formal analysis of any kind of property by using only one approach or tool (the so-called "*one-fits-*

all" approach): all verification techniques for reasonably interesting properties suffer from the general issue of undecidability and state explosion. When dealing with various DSLs, the classical approach expects that engineers will define several mappings for each DSL, by means of transformations, as depicted on the left of Figure 1.2: each mapping will target a dedicated verification domain that covers a limited, but specialised, area that will try to improve analysis results by using dedicated algorithms and data structures. What we propose is to generalise this approach at the level of transformation frameworks: as depicted on the right of Figure 1.2, only one bridge for each verification domain is required, defined once and for all. This way, any DSL defined within a transformation framework directly benefits from all these mappings for free. For this approach to be effective, a reference semantics of the transformation framework language is required, i.e. a semantics defined independently of the targeted verification domains. This semantics is used to precisely identify which abstractions are necessary for which verification domain. For example, loops are a common source for non-termination: for formally ensuring termination, one can focus on reentrant rules or recursive operations, and absract away from the manipulated data. Of course, this approach comes at a price: it is no longer possible to finely tune the mappings towards verification domains with the specific information contained in DSLs; rather, it shifts the effort into defining the necessary abstractions one have to define to overcome undecidability, which can be precisely documented using the reference semantics.

The remainder of the Section details our contributions: the next Section elaborates on the Description Framework; Sections 1.4.2 and 1.4.3 details how we apply the previous idea for the Kermeta transformation framework with a bridge into Maude, a rewriting system offering model-checking and theorem-proving capabilities.

1.4.1 A Description Framework for Model Transformation Intents

Several aspects of model transformation have been thoroughly investigated in the literature: for example, how to build model transformation languages or to apply model transformations in different contexts (Sánchez Cuadrado, Guerra, and de Lara 2011; Syriani and Vangheluwe 2010a). However, very few research has been conducted on the different intents, or purposes, that model transformation can typically serve in MDE. In particular, how they can be leveraged for development and validation activities is a clear lack in the literature.

In this Thesis, we propose the notion of *intent* for capturing the purpose of a transformation and its expected goals. We present a description framework for model transformations intents that allows the construction of a model transformation catalogue through the identification of properties that an intent must or may possess, and any condition that support or conflict with an intent. For instance, the *Translation* intent describes model transformations aimed at translating the meaning of a model in a source language in terms of concepts of another target language, whose result can then be used to achieve several tasks that are difficult, if not impossible, to perform on the originals. Thus, for a transformation to be considered as a valid realisation of the *Translation* intent, it should produce an output model that maintains a semantic relationship with the input.

We expect to propose an useful guide for MDE practitioners and researchers. For instance, it would help engineers identify the model transformation intent that best matches a particular MDE development goal, and consequently facilitate the subsequent model transformation development, by providing a comprehensive explanation of the properties a model transformation has to satisfy. This work is a first step towards a *validation engineering* in MDE.

1.4.2 A Formal Specification of Kermeta

For addressing the necessity of multiple verification domains, we define in this Thesis a fully-fledged formal semantics of a relevant subset of Kermeta. Such a semantics can serve as a reference, independent of Kermeta's current implementations (historically in Java, now ported to Scala), so that users can "play" and reason about the language independently of the execution platform, and propose more advanced analysis tools without being forced to deal with the entire platform.

The formalisation is defined at the transformation framework's level: any DSL whose specification fits the addressed Kermeta subset receives *de facto* a formal underground without any extra effort. This radically contrasts with traditional *ad hoc* approaches for equipping DSLs with formal semantics, that depend on the chosen target execution as well as the target verification domains. The proposed formalisation does not rely on any tool-oriented syntax or formalism, relieving the reader from learning new languages just for the purpose of understanding Kermeta's semantics. Rather, the formalisation relies solely on mathematics and computer science concepts: *set theory* captures the meaning of the metamodels' structures; and *rewriting rules* capture the meaning of Kermeta's dynamic aspects (i.e. those language constructions used for expressing MDE transformations) by means of a *structural operational semantics*.

By choosing an adequate target verification domain, it becomes possible to tailor the semantics to a particular analysis by defining appropriate, documented abstractions helping to overcome verification undecidability. Starting from the fully-fledged reference definition, one consciously defines which parts are irrelevant for a given analysis, and consequently abstracted away, instead of leaving them implicit. For example, performing shape or escape analysis would focus on Kermeta operation calls sites and model topology, abstracting from precise attribute values; whereas seeking for model-checking capabilities requires to represent as precisely as possible data structures manipulated during the transformations.

1.4.3 KMV: a Proof-of-Concept Kermeta Model-Checker

As an example of formal verification for Kermeta, we propose in this Thesis KMV, a proof-of-concept implementation of Kermeta's semantics that enables model-checking as well as theorem-proving.

Choosing Maude as a target verification domain pursues three objectives. First, it shows that the mathematical formalisation can effectively be concretely implemented in a tool, proving that mathematical formalisation are not just an exercise, but is rather concretely anchored in software engineering practice. Algebraic specifications and rewriting, the two core components of Maude, give the possibility to translate the formalisation into a semantic target that offers constructions close to the mathematical tools, reducing the semantic bridge between the specification and the implementation. Second, Maude can execute such specifications, providing an alternative execution platform to Kermeta; but more interestingly, Maude provides model-checking as well as theorem-proving capabilities that are interesting to explore in the context of transformation languages like Kermeta. We illustrate the capabilities of such a translation with small, yet representative examples covering different kinds of properties.

1.5 Thesis Outline

This Thesis is organised into three Parts. Part I provides a background on MDE and the formal verification of model transformations, and details our methodological framework for model transformations verification. This Part consists of three chapters:

Chapter 2: MDE: (Meta-)Models & Transformations. This Chapter constitutes a background on MDE and DSMLs, as well as a quick review of the existing model transformation tools.

Chapter 3: Formal Verification. This Chapter introduces the Verification Problem by stating what distinguishes Formal Verification from other validation approaches, and reviews classical approaches in the domain. It also surveys the usage of Formal Verification in the domain of model transformations, providing a comprehensive panorama of existing contributions and research trends.

Chapter 4: Model Transformations Intents Description Framework. This Chapter describes our approach for describing model transformation intents properties. It presents an Intent Catalog of 21 common transformation intents, and presents a high-level formalisation of key characteristic properties of transformation intents. The framework is applied on a real-life case study consisting of a chain of more than 30 model transformations, aiming at producing hardware code for a Power Window, starting from high-level requirements.

The other Parts focus on the verification of Kermeta transformations. As an enabler for Formal Verification, Part II addresses the Formal Semantics of (a subset of) Kermeta.

Chapter 5: Kermeta in a Nutshell. This Chapter is an introduction to Kermeta, the model transformation language studied in this Thesis. It briefly explains how the language was designed, and presents the Structural and Action Languages, core constituents of Kermeta.

Chapter 6: Structural Language. This Chapter formalises the Structural Language, used for defining metamodels and models used within DSLs. The formalisation uses set theory.

Chapter 7: Action Language. This Chapter formalises the Action Language, used for specifying model transformations in general. In particular for our work, this Language allows the definition of DSLs behavioural semantics. The formalisation is expressed in terms of Structural Operational Semantics.

Finally, Part III addresses the Formal Verification of Kermeta transformation using Maude, by presenting our proof-of-concept tool, KMV, and demonstrating its use on some examples.

Chapter 8: Maude in a Nutshell. This Chapter is an introduction to Maude, the target domain used for verification. It briefly explains how Maude specifications are built using Equational and Rewriting Logics.

Chapter 9: KMV: Verifying Kermeta with Maude. This Chapter explains how Kermeta's formal semantics is implemented in Maude. This provides an alternative execution engine for Kermeta transformations, but also a platform for analysing such transformations.

The Thesis finishes by a general Conclusion that summarises the Thesis content and sketches some perspectives; Appendices gather fully detailed material used within the Thesis, as well as a Publication Summary.

Chapter 10: Conclusion. This Chapter summarises the main contributions and results, and outlines possible future research directions.

Appendix A: The Finite State Machine Example. This Appendix summarises the code for the Finite State Machine example in the languages used in the Thesis: first using MDE techniques, then with Kermeta, and finally with Maude.

Appendix C: Publication Summary. This Appendix summarises our publications.

Part I

Models, Model Transformation & Model Transformation Verification

This Thesis is at the crossroad of two research fields: Model-Driven Engineering (MDE) and Formal Verification. The goal of this Part is to explore each field and try to reconcile them.

Chapter 2 explores the core MDE notions, models and model transformations, to provide the necessary background for understanding our work. It covers the fundamental concepts, but also provides an insight on the MDE techniques and tools.

Chapter 3 explores the domain of Formal Verification applied to MDE. After recalling background notions around Formal Verification, it proposes a comprehensive state-of-the-art based on a tridimensional approach for studying the field. This Chapter is based on a paper (Amrani et al. 2012b) presented at the Workshop on Verification of Model Transformations (VOLT).

In Chapter 4, we try to reconcile MDE with Formal Verification by proposing a so-called *Description Framework* that consists of a methodology for supporting model transformations designers through the task of ensuring the correctness of their transformations. This Chapter is based on a preliminary study (Amrani et al. 2012a) presented at the Workshop on Analysis of Model Transformations (AMT), and was later extended and then recently submitted as a Journal paper (Amrani et al. 2013).

2 Model-Driven Engineering: (Meta-)Models & Transformations

During the last decade, software engineering radically changed. On the one hand, the engineering practice evolved in order to improve the design and deployment of software, but also its maintenance over the years. Those changes became necessary to respond to the stakeholders demands, including shorter development delays, but also better evolution of software functionalities over time. On the other hand, new platforms, with new challenges and specificities, appeared recently: smartphones or graphical tablets, embedding more and more elaborated processors, and consequently, software; but also more critical applications in the automotive or aeronautics industries, among others, make reliability become crucial.

Model-Driven Engineering (MDE) seems to be a promising approach for reducing development time and costs, without sacrificing the quality and reliability of software. MDE promotes models and model transformations as first class citizens for all the facets of the software activity. Among all existing approaches following the MDE principle, Domain-Specific Modelling is an approach that consists in focusing on a narrowed technical or business domain for promoting automation and reuse. In this thesis, we study Kermeta[1], a metamodelling framework designed to support Domain-Specific Modelling in an object-oriented fashion.

This Chapter clarifies the concepts, the terminology and the foundations of MDE core artifacts, with the objective to provide the necessary background knowledge to understand how our formalisation for Kermeta is built: Models and Model Transformations are discussed in Sections 2.1 and 2.2 respectively; Section 2.3 explores Domain-Specific Modelling both from a Software Engineering viewpoint, but also from a mathematical viewpoint to explain our formalisation methodology.

This Chapter illustrates MDE concepts and Domain-Specific Modelling with several examples. Although we tried to keep things sufficiently simple, we assume the reader is familiar with at least two languages, very standard and commonly used in MDE, namely the Object Management Group (OMG) languages UML (Universal Modeling Language) (Object Management Group 2011a) and MOF (Meta-Object Facility) Object Management Group (2006). Since MOF is the underlying formalism for representing structures in Kermeta, it is thoroughly discussed in Part II.

2.1 What is a Model?

Etymologically speaking, the word "model" derives from "*modulus*": it can be translated as *measure*, *rule*, *pattern*, or *example to be followed*. Not suprisingly, this notion is exploited by many scientific disciplines, like physics, chemistry, biology, but also by less fundamental sciences like meteorology. Although the respective meaning of "model" in these discipline can vary, they all share a common characteristics: a model is a *sound abstraction* of an original (or, a subject) that depends on the science matter. Being an abstraction means that a model retains a relevant selection of the original's properties, or characteristics, which have to be understood both regarding the original's context, and the user's concerns. Being sound means that a model can be safely used in place of the original for predicting, or infering things about the original (Stachowiak 1973).

[1] http://www.kermeta.org

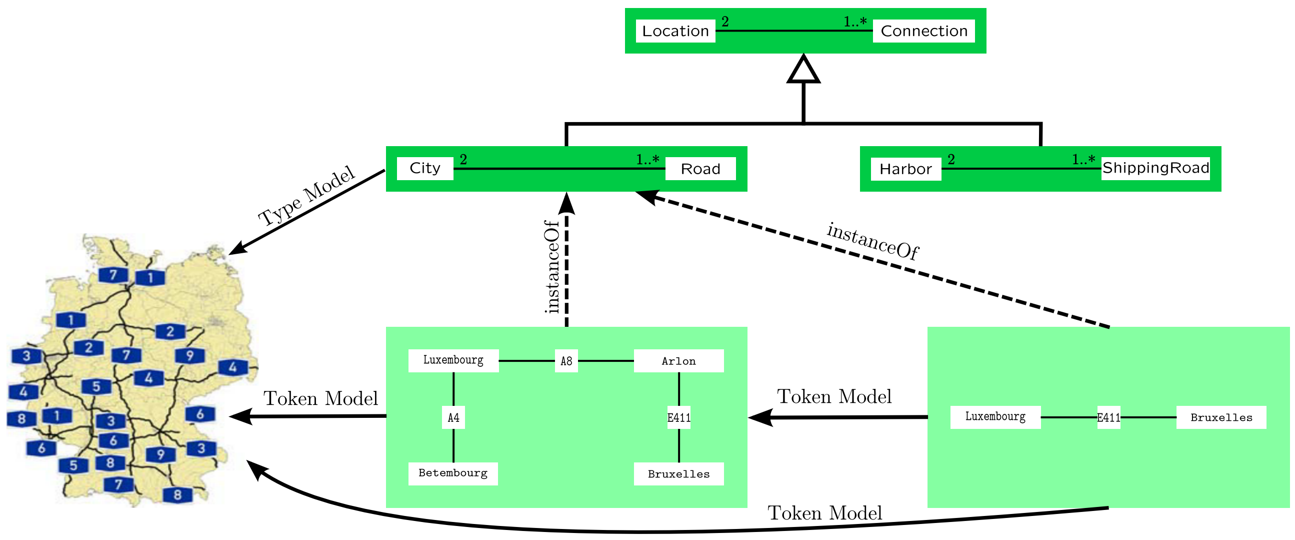


Figure 2.1 – Token *versus* Type Models (adapted from T. Kühne 2006) Moussa ▸*Add Subject Domain / Models*◂

As it is generally the case in science, the precise meaning of the term "model" in the Computer science and engineering discipline highly depends on the particular field of its use: for instance, image reasoning and software analysis both use models, but the respective models' relationships to the original, their use as well as the kind of reasoning, predictions and inferences engineers can perform differ radically.

No real consensus emerged over the last decade in the MDE community for precisely defining what a model is. However, the experience from academics, but more importantly from industrial practitioners, established a common sense that this Section explains, based on the work of Atkinson & Kühne (Atkinson and T. Kühne 2002a,b; T. Kühne 2006): we first explore the relationship between a model and what it models; then study the cornerstone notion of instantiation and the notion of metamodel. This allows us to propose an approach for the formalisation of metamodelling, which serves as a basis for Kermeta's Structural Language in next Part.

2.1.1 Model Kinds

A model is basically a *representation* of a subject, but it is not the subject: a model always operates an *abstraction* regarding its original, and it becomes relevant only when paired with this original. This relationship between a subject and a model is crucial in MDE, and can be of two kinds: a model is either a *token model* or a *type model*. This distinction is illustrated in Figure 2.1 with the classical example of a road map, which basically depicts cities interconnected through roads (T. Kühne 2006).

Token models are models that *directly* represent originals in a one-to-one fashion, by capturing *singular* aspects of the original's elements. At the bottom of Figure 2.1, we find two models (in light green) for representing the map on the left. The model in the middle associates one model element to exactly one city or road, and links things together with respect to the map. Notice that at this point, some details of the reality are abstracted away: e.g., the model does not retain how many legs a road is composed of. The model in the bottom right represents the same map, but with even fewer details: here, only countries' capitals are retained, and only the longest roads are retained in the model to link capitals together.

Type models are model that *classify*, or *conceptualise*, the original's elements in a many-to-one fashion, condensing complex original's features into concise descriptions, thus capturing *universal* aspects of the original's elements. In the dark green models on top of Figure 2.1, we only find two model elements, *City* and

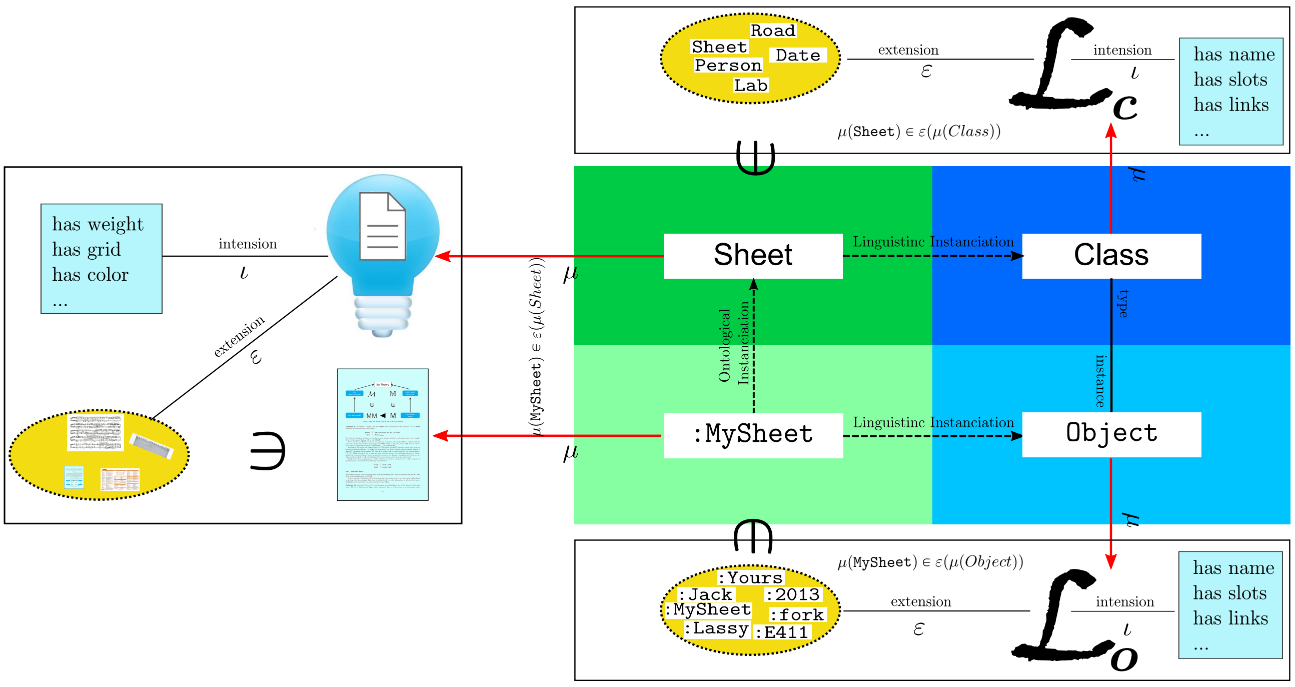


Figure 2.2 – Ontological *versus* Linguistic instantiation (adapted from (T. Kühne 2006))

Road, which classify the actual elements represented by the map by stating the universal nature of maps: roads link two cities, and cities are possibly connected through several roads.

The *classification* step, required for models to qualify as a type model, induces a specific relationship: many *elements* are mapped to one *concept* (e.g., *Bettembourg*, *Luxembourg* and so on are mapped to *City*). This relationship, between elements and concepts, should not be confused with *generalisation*, another very common relationship in MDE. Generalisation relates several concepts to one single (super-)concept and thus deals with type models elements, instead of talking about the "real" objects. As an example, Figure 2.1 shows another type model depicting maritim roads: *ShippingRoads* connect *Harbors*. Maritim and terrestrial roads are conceptually very similar if we forget about (or abstract away from) their support: it makes sense to generalise them by considering *Connections* between *Locations*, as represented on the top of Figure 2.1, using the classical UML plain arrow.

MDE makes use of token and type models under different names: roughly speaking, as the UML-based syntax already suggests, token models and type models respectively correspond to object and class diagrams. A token model is an *instance of* a type model if they relate to the same subject domain, the former as a representation and the latter as a classification.

2.1.2 Instantiation

Token and Type models have a relationship with the artifacts from the subject domain. Instantiation is, on the contrary, a relationship internal to models. We explain now the fundamental distinction between *ontological* and *linguistic* instantiation, since it is a core point in formalising Kermeta's Structural Language.

Figure 2.2 presents a possible modelling of concepts related to paper sheets, as it can be needed in a printing company. On the very left on white background, the objects belonging to the expertise domain, sheets, are described with respect to two levels: the conceptual level of sheet at the top, symbolised by a sheet surrounded by a lamp; and the actual level of objects at the bottom, with (an image of) an actual sheet, for example, a witness sheet extracted from this PhD manuscript. The conceptual level can be thought of as a set covering all possible sheets in the world, past and future. Therefore, it can be defined *intentionally*, using a bunch of predicates indicating whether an object from the reality can be considered as a sheet; or *extensionally*, by enumerating all objects considered as a sheet (which is impossible for infinite sets, as it is for sheets). These descriptions are captured with two functions ι and ε. Our witness sheet object then naturally belongs to the extension of the sheet concept.

In the green background, this expertise domain is modelled for computer usage, i.e. for being able to reason about sheets (e.g., assembling sheets for creating a book, flipping or rotating a sheet, etc.) Both levels are modelled (using here the classical UML representation for class and object diagrams): the class *Sheet* models the idea of sheet at a conceptual level; the object `:MySheet` models the witness object from the domain (note that we use two different fonts for denoting classes and objects). As we did in Figure 2.1, both models *represent* objects from the subject domain of the white background box on the left: the μ functions capture the association between models to their representatives. However, they qualify as models because they perform an abstraction step regarding their originals: for instance, the class does not retain those sheet features not relevant for book assembling, like the weight of the paper; and the object is unable to distinguish between perfect copies of the same sheet to be assembled.

In the blue background, the model domain is in turn modelled by elements that describe the *language* used for modelling the subject domain. In other words, elements from the blue bakcground have as a subject domain the elements of the green background: as the intuition already suggested, elements from the top layer correspond to *Class*es whereas those from the bottom layer correspond to *Object*s. As language elements, artifacts depicted in the blue background also have a precise meaning, i.e. the one traditionally attached to languages, as depicted in the white areas surrounding the green and blue backgrounds. For instance, the *Object* class represents a language (fragment) $\mathcal{L}_{Object}$ (hence, the μ function). As a language, $\mathcal{L}_{Object}$ describes the set of acceptable sentences valid for *Object*, i.e. recognised as an UML *Object*: this can be described extensionally (all acceptable sentence of the language recognised as *Object*) and intentionally (with a few characteristics like slots, names, links and so on, just as the Object Management Group (2006) has formalised it for MOF). The same obviously stands for *Class*.

So :MySheet represents the actual witness sheet within a computer: it is at the same time an *instance of Sheet* and an *instance of Object*. However, this instantiation relationship has a different nature:

Ontological instantiation The element `:MySheet` is an *ontological* instance of *Sheet* because it is a specific incarnation of the general concept of sheet: `:MySheet` refers to an element belonging to the extension of the concept referred to by *Sheet*, which is formally expressed by:

$$\mu(\texttt{:MySheet}) \in \varepsilon(\mu(\mathit{Sheet}))$$

Linguistic instantiation *Sheet* and `:MySheet` are *linguistic* instantiation of *Class* and *Object* respectively. This fact is further enforced by the use of the corresponding UML dedicated syntax for classes and objects. Then, *Sheet* and `:MySheet` belong to the extension of the language (fragment) meant by *Class* and *Object* respectively, which is formally expressed by

$$\texttt{:MySheet} \in \varepsilon(\mu(\mathit{Object}))$$

Although very close, these instantiation relationships are different, as showed in the previous formulæ: with ontological instantiation, a "detour" *via* the subject domain is required, forcing us to use the object represented by :MySheet (i.e., using $\mu(:\text{MySheet})$) in the set membership test; whereas with linguistic instantiation, :MySheet appears as itself for the membership test, because only the *form* of elements is relevant as opposed to their *content*, or *meaning* for the ontological case. This distinction is made even clearer if we notice that elements from the blue background are not related at all with those elements from the subject domain (white background on the left): as a matter of fact, a linguistic model is never a model of its subject model's subject.

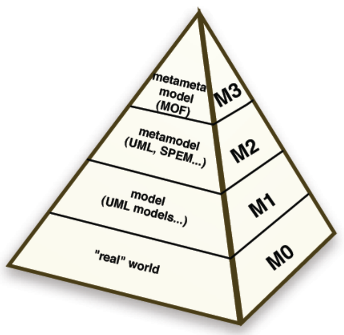


Figure 2.3 – The OMG MDA Pyramid (from (Thirioux et al. 2007))

2.1.3 (Meta-)Metamodels

With the considerations above, we can now clarify the notion of *metamodel* largely used in MDE. As the etymology of the prefix "meta" suggests, a metamodel is a model of a model (Object Management Group 2003, 2004). However, not all models of models can qualify as metamodels. For example in Figure 2.1, the model at the bottom right is also a model of the model in the middle, but can hardly qualify as metamodel: it has the same relationship to the subject domain, whereas a metamodel is expected to operate some "detachement" from its subject. Therefore, only models that are type models of other models can qualify as metamodels, thus involving an instantiation relationship.

It seems then natural, in an MDE approach, to also use a model to properly define a metamodel. This former model is then known as a *meta-metamodel*, and the overall approach is captured by the so-called *metamodeling pyramid* of Figure 2.3, as initially proposed by the Object Management Group (2003): the M0 level corresponds to the "real" world, what we referred to by "*subject domain*" in Section 2.1.1; the M1 level corresponds to the *model* layer, i.e. to *token* models representing artifacts from M0; the M2 level corresponds to the *metamodel* layer, i.e. *type* models of M1; and finally the M3 level corresponds to the *meta-metamodel* layer, i.e. a model for defining (syntactically) metamodels. This pyramid presents two particular features:

- The qualification of M3 as a meta-metamodeling layer is improper with regards to the different instantiations we distinguished previously. As one could expect, the prefix "meta", applied twice, should correspond to the application of the same relationship twice, but this is not the case here: the relationship between M1 and M2 is *ontological* because M2-artefacts are type models of M1-artefacts (cf. the relation between `:MySheet` and *Sheet* in Figure 2.2), but the relationship between M2 and M3 is *linguistic*, because the meta-metamodel describes M2-artefacts' syntax (just like the relation between *Sheet* and *Class* in Figure 2.2). One can still consider M3 as a *meta-metamodel* if only the instantiation relationship is considered independently of its nature.

- The pyramid contains a hidden layer, known as the *meta-circularity* property of MDE meta-metamodels: the (unique) model in M3 is in fact defined in terms of itself, enjoying again a linguistic relationship with its syntax. This is particularly useful for reflexivity purposes: all metamodels are syntactically perceived as being (linguistic) instances of the same meta-metamodel, including the meta-metamodel itself, which becomes a "regular" M2 citizen.

The first remark has a concrete conceptual consequence from a semantic point of view. Since M3-models capture the syntax of M2-models, providing a formal semantics for M3 is sufficient to capture the *linguistic* meaning of any M2-model, independently of what it *ontologically* represents. This approach is somehow natural and unavoidable: the actual meaning of M2-models, in all their diversity, cannot be formally captured once and for all, since they model various subject domains; but it is enough to reason on this M2-model which *represents*

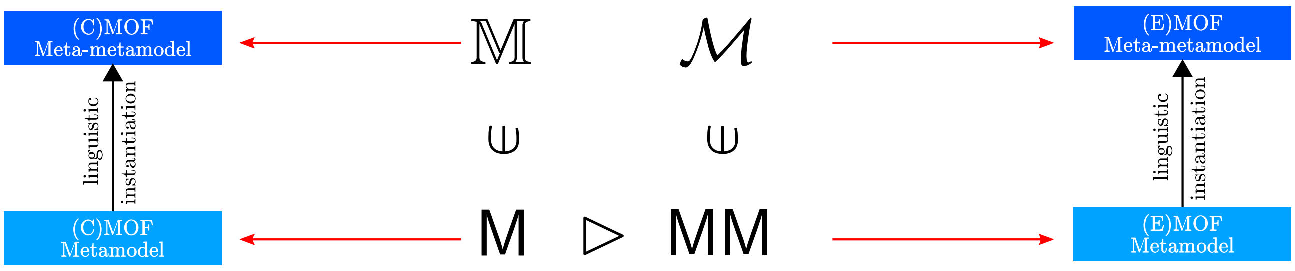


Figure 2.4 – Formal construction for metamodel and model sets for Mof. Languges $\mathcal{L}_o$ and $\mathcal{L}_c$ of Figure 2.2 are renamed $\mathcal{M}$ and $\mathbb{M}$ respectively. Instantiation is now more clear: *linguistic* is captured by set membership $\in$; *ontological* by a specific relationship $\triangleright$. Red arrow still denotes μ functions for representation.

what is necessary to know about the domain, and what will be computationally manipulated. Defining the "right" model for a domain is therefore subsequent to design choices, guided by the model's purpose and future uses, but this is not a exact science. As a direct consequence, it is impossible to infer properties on models that are not somehow "included" within the model.

The second remark has direct consequences from an engineering point of view. Since the meta-metamodel can be treated exactly as any other metamodel, it favors reuse and reduces development costs of metamodeling frameworks using the well-known *bootstrap* technique: once a first core version of a metamodeling framework is mature and stable enough, treatments on the meta-metamodel can be handled using the framework itself. It also facilitates metamodel exchange and compatibility, if metamodel bridges (Deltombe, Le Goaer, and Barbier 2012; Wimmer and Kramler 2005) are defined between already existing meta-metamodels[2].

2.1.4 Discussions

We discussed in this Section *models*, the first key artifact in Mde. We explained what a model is, and identified two relationships:

- an *extra*-model relationship, i.e. a relationship between a model and its subject, where token models correspond to the Uml notion of "object diagrams" and type models to Uml "class diagrams";
- an *intra*-model relationship, i.e. a relationship between models corresponding to the Uml notion of instantiation, and distinguished between ontological and linguistic instantiations.

The second relationship shows that from a linguistic viewpoint, two languages are required to describe both levels of models: in Figure 2.2, $\mathcal{L}_o$ captures the meaning of object diagrams, or token models; whereas $\mathcal{L}_c$ captures the meaning of class diagrams, or type models. This is not surprising: Mof itself is organised similarly. Metamodels' syntax is captured by eMof, and models' syntax by cMof.

We follow the same perspective for our formalisation of Kermeta's Structural Language. The following Definition introduces the notations we will use from now on for these notions:

Definition 2.1 ((Meta-)models — Conformance)**.** *$\mathcal{M}$ and $\mathbb{M}$ are the sets of all metamodels and models respectively, as defined by the (*Mof*) meta-metamodel. For a given model $\mathsf{M} \in \mathbb{M}$ and a given metamodel $\mathsf{MM} \in \mathcal{M}$, we note $\mathsf{M} \triangleright \mathsf{MM}$ if* M conforms to MM. *The set of all models conforming to MM is noted $\mathbb{L}(\mathsf{MM})$.*

Figure 2.4 illustrates this Definition by relating each set to its corresponding Mof metamodel, and by presisely distinguishing between each type of instantiation. In fact, $\mathcal{M}$ and $\mathbb{M}$ describe the *form*, or the *syntax*, of metamodels and models respectively: a metamodel $\mathsf{MM} \in \mathcal{M}$ basically consists in declarations that bind metamodel elements' names to (syntactic) types and other information with respect to a particular topology defined by the

[2]Bridges between technical spaces are discussed in Section 4.5.3, as a particular case of a *Translation* between two metamodels.

MOF meta-metamodel; a model $\mathsf{M} \in \mathbb{M}$ consists of a collections of objects (also called class instances) that possess a type, i.e. its referring class, and a state, represented by the values of its properties (i.e. MOF's attributes and references).

The *ontological* instantiation is captured by the *conformance* relationship $\triangleright$, as depicted in Figure 2.2 for our Sheet example: this relationship relates two models whose originals are in the same domain, but living at two different logical levels (i.e. the conceptual and the actual ones). Since computers do not have access to the things they model, a "real" ontological test cannot be mechanised; instead, a syntactic conformance predicate is defined to figure out wether a model can be regarded as an (ontological) instance of a type model.

MOF meta-circularity property becomes now clearer: as any other language, metamodels' *syntax* can be described by a metamodel, i.e. there exists a particular metamodel MMM, conforming to $\mathcal{M}$, that exactly captures metamodels' syntax, i.e. the basic elements constituting metamodels (such as packages, classes and so on) as well as their relationship (e.g., the subpackage relation cannot be cyclic).

2.2 Model Transformations

The previous Section provided a discussion about *models*, the first central notion of MDE. Models capture various data structures, spanning from elementary ones like sets, to arbitrarily complex ones. But they are inherently static, in the sense that they just capture concepts and their relationships.

This Section discusses *model transformations*, the second central notion of MDE, which is computational: model transformations allow the computerised manipulation of models in various ways. As one can expect, this area is also very complex and can be declined in many different flavours. We provide in this Section a panorama of this area, starting from a working definition that highlights the core components of model transformation. We then discuss each component from our perspective of semantics specification and formal verification.

2.2.1 Definitions

Historically, one of the first definitions was proposed by the OMG, in straight line with the Model-Driven Architecture view. The Object Management Group (2003) perceives transformations as "*the process of converting one model to another model of the same system*". Immediately after, the system-centric view was enlarged by A. G. Kleppe, Warmer, and Bast (2003): "*a model transformation is the automatic generation of a target model from a source model, according to a transformation definition*". This definition shifts from the system-centric view in order to consider general source/target models, insisting on the fact that transformations are mostly perceived as *directed* and *automatic* (i.e. without users' intervention) manipulation of models. Tratt (2005) describes a transformation as "*a program that mutates one model into another*", insisting on the computational aspect of transformations. More recently, two contributions widened the perspective with two important aspects: Mens and Van Gorp (2006) proposed to see transformations as "*the automatic generation of one or* multiple *target models from one or* multiple *source models, according to a transformation description*", whereas Syriani (2011) re-introduced the crucial importance of the specific intention behind transformations by defining transformations as "*the automatic manipulation of a model with a specific intention*".

We propose a broader definition that clearly embeds the dual nature of model transformation, distinguishing its specification from its execution, and places the transformation's intention at its core:

Definition 2.2 (Informal Definition from Amrani et al. (2012b)). *A* transformation *is* the **automatic** manipulation, conforming to a **specification**, of **(a) source model(s)** to produce **(a) target model(s)** according to a **specific intent**.

Figure 2.5 depicts the elements involved in a model transformation. An input model, conforming to a source metamodel, is transformed into an output model, itself conforming to a target metamodel, by executing a trans-

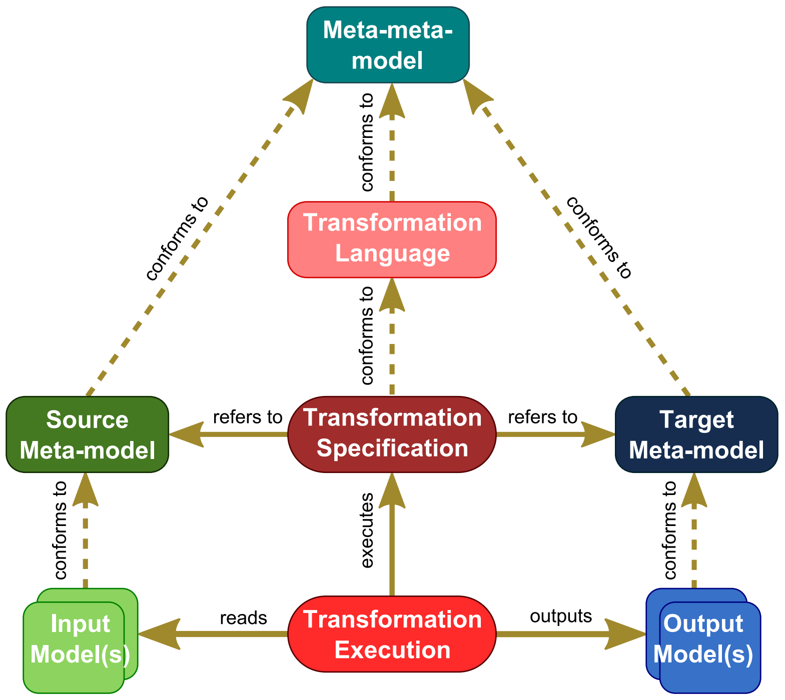


Figure 2.5 – Model Transformation: the big picture (adapted from (Syriani 2011))

formation specification. A transformation specification is defined in terms of the source and target metamodels, whereas its execution operates on the model level. Both source and target metamodels, as well as the transformation specification, are themselves models, conforming to their respective metamodels: for metamodels, this is the classical notion of meta-metamodel; for transformations, it actually refers to the transformation language, or metamodel, which allows a sound transformation specification. Of course, some transformations may manipulate several source and/or target (meta-)models. For the purpose of this thesis, we only consider in this document legal transformations, i.e. transformations executing on conforming input models $\mathsf{M}_\mathsf{i}^1, \ldots, \mathsf{M}_\mathsf{i}^n$, and outputting models $\mathsf{M}_\mathsf{o}^1, \ldots, \mathsf{M}_\mathsf{o}^m$ eventually not conforming to their respective metamodels.

The rest of this Section discusses both levels for Model Transformations, namely *specification* and *execution*, and reviews the existing classifications and their limitations from the verification viewpoint.

2.2.2 Transformation Languages And Specifications

When specifying a transformation between source and target metamodels, a transformation designer has to follow the Transformation Language's syntax. The following Definition captures transformation specification's required components:

Definition 2.3 (Transformation Specification)**.** *A transformation specification is a triple* $\tau = ((\mathsf{MM}_\mathsf{i}^\mathsf{k})_{\mathsf{k}\in[1..n]}, (\mathsf{MM}_\mathsf{o}^\mathsf{k})_{\mathsf{k}\in[1..m]}, \mathsf{spec})$ *where* $(\mathsf{MM}_\mathsf{i}^\mathsf{k})_{\mathsf{k}\in[1..n]}$ *and* $(\mathsf{MM}_\mathsf{o}^\mathsf{k})_{\mathsf{k}\in[1..m]}$ *are indexed sets of input and output metamodels, respectively, and* $\mathsf{spec} \in \mathcal{L}_\tau$ *is a well-formed transformation specification written in a transformation language* $\mathcal{L}_\tau$*.*

As previously noticed by Bézivin et al. (2006), transformations have a dual nature:

- If considered as a *model transformation*, we emphasise a particular manipulation of source and target metamodels, as described by spec;
- If considered as a *transformation model*, we emphasise the linguistic nature of spec, i.e. the relationship between spec and its defining language $\mathcal{L}_\tau$, understood as a computation whose expression relies on a particular Model of Computation.

Since here, nothing constrains the nature of the metamodels involved and their manipulation, one can easily consider transformations of transformations, known as *Higher Order Transformations* (HOTs) (Tisi et al. 2009): in such case, metamodels represent transformations syntax.

How do model transformation languages allow designers to create transformations? Over the years, many languages as well as many transformation frameworks emerged, with various model transformation purposes and targets. We quickly review a categorisation of computing paradigms for model transformations, as previously stated by Czarnecki and Helsen (2006):

Programming-Based This category encompasses several ones proposed originally by the authors, in which practices already well-known in General-Purpose Programming Languages are applied, or adapted, for designing transformations.

Visitor-Based This approach adapts the well-known design pattern (Gamma et al. 1995) for models: it consists in traversing a tree-based internal representation of a model and then outputting the corresponding code into a text stream. A good example of this approach is Jamda (Boocock n.d.).

Template-Based This approach adapts the well-known program transformation technique (Visser 2005) for models: it consists of metacode to access the source model information, and performs code selection based on metavariables to store intermediate information, then expands them at runtime, when actual values are known. This approach is very common for code generation, like AndroMDA (Bohlen et al. n.d.) and MetaEdit+ (Kelly and Tolvanen 2008).

Direct Manipulation This approach offers an internal model representation, completed with an API to manipulate it, which is directly accessible by the transformation designers. APIs are usually object-oriented, and users have to define their own libraries for enabling reuse.

Programming-Based approaches are generally well-suited when target metamodels are intented to be manipulated through text, as so-called "*model-to-text* (M2T)" transformations. Notice however that M2T can be considered as a special case of general metamodelling: tools like XText (Bettini 2013) already bridge both approaches, by easing the definition of a textual representation of a metamodel, or the parsing of a textual representation for creating a corresponding model.

Operational This category, also known as *meta-programming*, is similar to the previous one, but usually offers more dedicated support for handling model manipulation. Generally textual, it consists in enriching the meta-modeling formalism with facilities for expressing computations, which is actually in the spirit of what MOF naturally proposes. Obviously, Kermeta belongs to this category, but other frameworks exist as well, e.g. Epsilon (Kolovos et al. 2012).

Declarative This category is usually based on rewriting, and contrasts with the previous one by letting the designer specify *what* is expected from the transformation instead of describing *how* to realise it, and is generally equipped with a visual syntax (although most transformation engines actually use a hybrid syntax, where visual and textual syntaxes are mixed more or less appropriately). Two different approaches exist in this category:

Relational Transformations are defined based on mathematical relations among source and target metamodels elements, using constraints. Due to its nature, it makes this approach sometimes not executable (since sometimes, the constraints have no solution). Logic and Constraint Programming are the natural semantic targets for this approach.

Graph Transformation-Based This approach represents models as graphs, and benefits from the well-established category theory (Rozenberg 1997) for dealing with transformations. However, some native MDE constructs, like inheritance and containment (Jurack and Taentzer 2010), or multiplicities (), have been integrated only recently. Examples of such transformation engines are AGG (Taentzer

2000), ATL ((ATL) n.d.), AtoM^3 (de Lara and Vangheluwe 2002), QVT (Object Management Group 2008), or VIATRA (Varró and Balogh 2007), among others.

This categorisation is never perfect: hybrid approaches are common in model transformation frameworks that combine advantages from several sides.

2.2.3 Transformation Execution

From the runtime viewpoint, a transformation execution is not different from any other program running on a computer. Therefore, transformation executions can be mathematically modelled as a general Turing Machine executions.

Definition 2.4 (Transformation Execution)**.** *The* execution*, or* semantics*, of a model transformation specification t is given by a transition system* $\mathsf{TS}_\mathsf{t} = (\mathsf{S}, \mathsf{I}, \longrightarrow)$ *where* S *is a set of* execution states*;* $\mathsf{I} \subseteq \mathsf{S}$ *is a set of* initial *states, called* input models*; and* $\longrightarrow \subseteq \mathsf{S} \times \mathsf{S}$ *is the transition relation over* S*.*

The precise definition of S and $\longrightarrow$ obviously depends on the transformation language. S includes at least the models involved within the transformation, but often also extra information: for example for meta-programming languages, features like control flow and local variables are necessary to keep track of the execution logic. Similarly, $\longrightarrow$ is easy to define for graph-based transformation languages, since it follows the semantics attached to rewriting rules (cf. Rozenberg 1997, for more details).

If a transformation execution is not different from any other executable artifact, what exactly differentiates model transformation? Darlington and Burstall (1976) introduced in 1976 the closely related notion of *program transformation*: the purpose was to transform a program into another program that was expected to preserve the semantics. Originally designed for coping with complexity involved by recursive programs, the methodology became popular and gained maturity over the years, if we judge by the number of transformation systems developed since then (cf. Partsch and Steingbrüggen (1983) for a comparative survey of the tools, and Visser (2005) for a survey of the techniques and applications).

The difference between program and model transformation is not clear-cut: some model transformation approaches are very close to actual programming, although models usually manipulate semantically rich data. Syriani (2011) distinguishes three differenciating characteristics:

- Models are more diverse than program, and can equally represent rich concepts like DSLs as well as programs, and model-driven environments sometimes represent programs in the same form as models;
- While programming languages are syntactically defined using grammars, leading to abstract syntax *trees* as their underlying formalism, models are usually richer and require *graphs* as their underlying formalism;
- Models provide higher-level constructs for structuring data, such as concept generalisation (or, equivalently in programming languages, inheritance) and powerful associations (such as aggregation and containment in UML) that are rarely present as is in programming languages.

Program or model transformation systems can be equally used for dealing with program or model transformation, since both are theoretically Turing-complete. But as noticed above, model transformation seems to be a broader field: it can always manipulate, in the same framework, richer and more diverse artifacts, including programs.

2.2.4 Model Transformation Classifications

From a verification viewpoint, it is important to classify model transformations in order to identify precisely which properties have to be checked to guarantee correctness. We review the existing classifications available in the literature, and show their limitations. We then propose a more suitable classification (that we will elaborate in the next Chapter) for analysing the existing work on formal verification of model transformations.

2.2.4.1 Features

A first level of classification is to highlight how a transformation is built, i.e. which features compose model transformations in general. We summarise the features proposed by Czarnecki and Helsen (2006), relevant for the purpose of model transformation formalisation and verification.

Transformation Units are the basic building blocks used to specify how computations are performed during the transformation.

Scheduling specify how transformation units are combined to perform the computation, either implicitly, in which case the transformation designer has no direct control; or explicitly, using a large set of possibilities, ranging from partially user-controlled schedulers to explicitly modelled DSLs for specifying schedule flow.

For example, most of the graph-based tools are based on rewriting rules specifying which patterns have to be found in the model, and which pattern have to replace them; whereas in Kermeta, statements are used to describe one step of the computation.

2.2.4.2 Forms

A second level of classification consists in focusing on the form of the transformation, i.e. in which ways it is related to the metamodels for its specification and the models for its execution. For this purpose, Mens and Van Gorp (2006) provided a multi-dimensional taxonomy: we briefly summarise here the interesting aspects of their classification:

Heterogeneity between the source and target metamodel: if they are the same, the transformation is *endogeneous* and expresses a rephrasing intention; otherwise, it is *exogeneous* and conveys a translation intention.

Abstraction Level related to the detail level involved into models: if the target metamodel adds or reduces the detail level, the transformation is *vertical*; otherwise, if the abstraction level remains unchanged, it is *horizontal*.

Model Arity regarding the number of source (respectively, target) models as input (resp., output) of the transformation.

Source Model Conservation related to the treatment of the source model: if the transformation directly alters the source model, it is *destructive*, or *in-place*; if another independent model is outputted, it is *conservative*, or *out-place*.

This classification is useful from a syntactic viewpoint: it gives information about the number of (meta-) models involved and their relative abstraction level. However, it is nearly impossible to figure out, from a verification viewpoint, which properties are interesting to be checked.

2.2.4.3 Intents

We introduced in Definition 2.2 the fact that a transformation follows an *intent*. We now show how important this notion is for the Formal Verification activity.

Let us consider a transformation that defines the semantics of a Domain-Specific Language like Finite State Machines (FSM). As depicted in Figure 2.6, such a transformation can have two forms:

- either *operational*, meaning that the semantics is expressed directly on the FSM's concepts[3], just like a mathematical definiton would do (e.g., if from the current state s, the automaton reads a letter v

[3]As we mentioned earlier, an operational semantics is specified *on the basis* of the DSL concepts, but usually requires extra information to keep track of data not immediately present in the DSL metamodel, but nevertheless required during the execution.

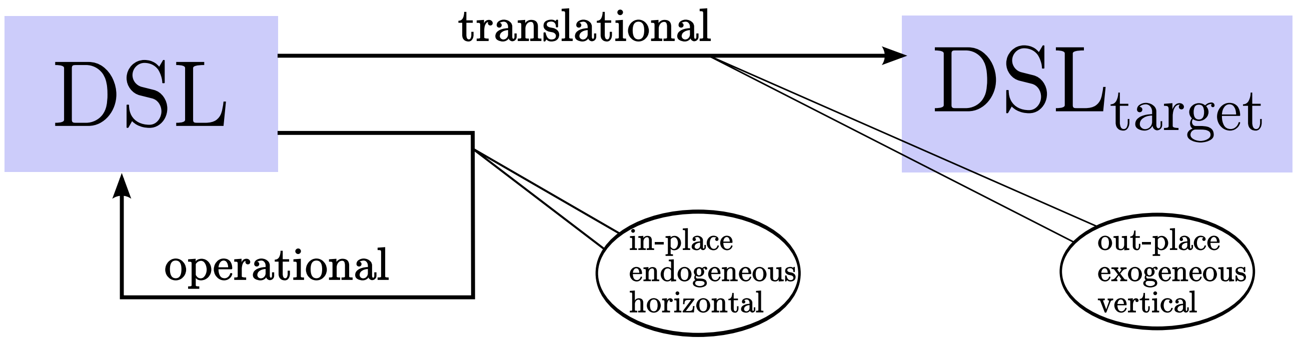


Figure 2.6 – Two ways for defining DSLs semantics: a *translational* transformation is out-place, exogeneous and vertical; an *operational* one is in-place, endogeneous and horizontal.

corresponding to a transition outgoing from s to s', then the current state is updated to s'), then the corresponding transformation is in-place, endogeneous and horizontal;

- or *translational*, meaning that the semantics is expressed by translating the FSM's concepts into another domain, like Petri Nets, where execution becomes possible, then the transformation is out-place, exogeneous and vertical.

The previous example shows that for one purpose, two possible transformations can be designed, each one leading to the opposite classification of the other. From a verification viewpoint however, properties of interest attached to a transformation should conceptually be the same (e.g. here, proving the correctness). This is not surprising, since the previous classification mainly focuses on the transformation's *form*, whereas from a verification viewpoint, the *meaning*, or the *purpose* of the transformation is more important in order to figure out which properties should be established.

We contributed to a community effort that aims at identifying a set of transformation intents that appear repeatedly in most MDE efforts, to build a *Model Transformation Intent Catalog* that identifies and describes transformation intents and the properties they may or must possess. This Catalog has several potential uses:

Requirements analysis for transformations The catalog facilitates the description of transformation requirements, i.e., of what a transformation is supposed to do. Improved requirements can improve reuse, because they may make it easier to locate suitable transformations among a set of existing ones and reuse them.

Identification of properties, certification methods, and languages The catalog may help transformation developers become aware of properties a transformation must possess, how these properties can be certified, and which transformation language is known to best support their needs (*i.e.,* if the used certification methods are language dependent).

Model transformation language design The catalog may provide some useful input for designers of domain-specific transformation languages. For instance, it may be appropriate to design dedicated languages for specific intents for efficiency or readability. The properties and certification methods associated with an intent may provide useful information about requirements of a transformation language used for an intent.

In Chapter 3, we will discuss two of them, namely *Simulation* and *Translation*, to analyse which properties these particular intents should possess. These properties will be formally checked with our tool in Part III.

2.2.5 Conclusion

As the second key notion in Mde, model transformation allows the manipulation of models in various ways. We discussed in this Section the core components of model transformations, and provided a comparison with classical programming. After having compared existing classifications for model transformations, we proposed a novel approach highlighting transformations intents, which is more suitable from the verification point of view.

2.3 Domain-Specific (Modelling) Languages

A common way to tackle the increasing complexity of current software systems consists in applying the "divide-and-conquer" approach: by dividing the design activity into several areas of concerns, and focusing each one on a specific aspect of the system, it becomes possible not only to raise the abstraction level of the produced specifications, with the immediate benefit of raising the confidence attached to them, but also to make them closer to each expert's domains, which facilitate the control of the produced artefacts, and sometimes even delegating their creations to these experts.

Within Mde, Domain-Specific Modelling (Dsm) becomes a key methodology for the effective and successful specification of such systems. This methodology makes systematic use of Domain-Specific Modelling Languages (Dsmls, or Dsls for short) to represent the various artifacts of a system, in terms of models. The idea is simple: focusing designers' efforts on the variable parts of the design (e.g., capturing the intricacies of a new insurance product), while the underlying machinery takes care of the repetitive, error-prone, and well-known processes that make things work properly within the whole system.

In this Section, we further investigate two facets of Dsml. First, we explore the building blocks of Dsmls, i.e. what is in the name of Domain-Specific Modelling Languages? Second, we review the core ingredients of Dsmls, from the viewpoint of languages: what are the basic components of a (computer, formal) language, and how are they used in the context of Mde?

2.3.1 Dsml Features

This Section helps understanding the notions behind the concept of Dsml, most of them already contained in the concept's name itself: what is a *domain*, in which way can a language become *specific* to a domain and why is this desirable, and what are the expectations for using such languages.

2.3.1.1 (Subject) Domain & Reuse

Dsls derive their expressive power from the fact that they directly represent concepts from the *subject* (or, equivalently, *expertise*) domain. Consequently, Dsls enable experts as well as developers to work efficiently by manipulating what they know best, often using symbols themselves imported from this domain. The idea of Dsls is by far not new, and not specific to the Mde field: Simos (1995) already provided an useful definition:

> We define a *software domain* as an abstraction that groups a set of software systems or functional areas within systems according to a domain definition shared by a community of stakeholders. A *domain* can be represented by a domain model, which serves as a basis for engineering components (or assets) intended for use in multiple applications within the domain. The *domain model* provides a formalized language for specifying the intended range of applicability of the assets[4]. (Simos 1995)

The author clearly separates what he calls the domain *as the real world* and the domain *as a set of systems*, which directly refers to the traditional distinction of domains in software engineering:

[4]The emphasising is from us

Functional Domain covers the notions from the real world directly related to the business at hand (e.g., bank, human resources, insurance, or health), which need to be abstracted and represented within the computer to be able to reason about;

Technical Domain covers the notions meaningful only for software applications and systems development that require specific tasks among (sub-)systems (e.g., distribution, concurrency, deployment, persistence, transactions and so on) for which stakeholders' knowledge is not necessary.

In this respect, Dsls recall the notion of *program family*, early introduced by Parnas (1977): common features from the domain are captured once and for all within the Dsl whereas variable parts can be expressed by the modeller using high-level constructs. This distinction promotes reuse of models at early stages of application design, as defined by Biggerstaff (1998):

> Software reuse is the reapplication of a variety of kinds of knowledge about one system to another similar system in order to reduce the effort of development and maintenance of that other system. This reused knowledge includes artifacts such as domain knowledge, development experience, design decisions, architectural structures, requirements, design, code, documentation, and so forth (Biggerstaff 1998).

2.3.1.2 Raising the Abstraction Level

Since computer scientists have begun programming computers, they faced the need to increase the abstraction level of programs. The history of programming languages, the primary interface for interacting with computers, has consistently offered new paradigms for writing and designing programs.

First generation programming languages were operating at the level of the Cpu, directly manipulating the operation codes used by a particular processor; second generation languages, although specific to a particular architecture, offered a first layer of abstraction by manipulating mnemonics and macros, allowing programmers to abstract from the internal binary representation of instructions; third generation languages, with their large scale of abstraction levels and programming paradigms, offered to programmers the possibility to specify computations independently from the low-level machine architectures, gaining a substantial expressive power. Among others, object orientation considers software systems as a large bunch of communicating objects, that enable programmers to tackle problems and to design systems that were considered impossible a few decades ago.

Mde/Mdd can be seen as the natural continuation of this effort to propose programmers the best abstraction level they need for each aspect of a large system: functional specification for users, designers, system architects; but also persistence, maintenance, documentation, and so on. If object orientation enabled Mde underlying paradigms, Mde operates in fact a radical rupture by going one step further: each aspect of interest is modeled at the required abstraction level.

Dsls epitomise this abstraction level aspect: Dsls are languages whose model elements directly represent concepts from the subject domain world, not the code world. Furthermore, Dsls usually enable representations richer than text: tables and matrices, but also graphical diagrams potentially using dedicated symbols from the domain. It then becomes possible to propose users richer interfaces for manipulating models: multiple views over the system can be kept synchronised, allowing users with different roles to work on different aspects of an application; and information hiding becomes a key capacity for displaying adequate information, hiding low-level or unnecessary details. With concepts from the domain, adequately represented with domain symbols, Dsls widen the number of users and involve even the domain experts who are usually reticent to deal with automated applications.

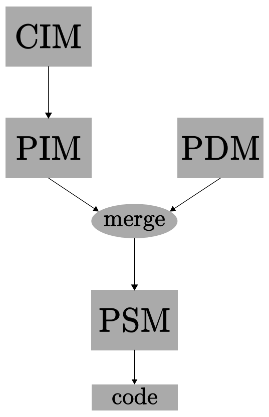


Figure 2.7 – Platform-Independant and Platform-Specific Models with Transformations (from Object Management Group 2003)

2.3.1.3 Narrowing Design Space

As the name indicates, DSLs operate within a narrowed area of interest: they are domain-specific rather than general-purpose. Since the modelling language is aware of the domain specificities, the modelling tasks are easier, and further analysis or code generation from models are simplified by the fact that they work on a smaller focus.

The approach supported by DSLs radically contrasts with other existing MDE approaches, among which the Model-Driven Architecture (MDA), an OMG proposal launched in 2001 to help standardise model definitions, and favor model exchanges and compatibility (Object Management Group 2003). The MDA consists of the following points (A. G. Kleppe, Warmer, and Bast 2003):

- It builds on UML (Object Management Group 2004), an already standardised and well-accepted notation, widely used in object-oriented systems. In an effort to harmonise notations and clean the UML internal structure, they proposed MOF (Object Management Group 2006) for coping with the plethora of model definitions and languages;

- It proposes a pyramidal construction of models as sketched in Fig. 2.3: artifacts populating the level M0 represents the actual system; those in the M1 level model the M0 ones; artifacts belonging to the M2 level are metamodels, allowing the definition of M1 models, and finally, the unique artifact at the M3 level is MOF itself, considered as meta-circularly defined as a model itself[5];

- Along with this pyramid, it enforces a particular vision of software systems development seen as a process with the following step: requirements are collected in a Computation Independent Model (CIM), independently of how the system will be ultimately implemented; then a Platform Independent Model (PIM) describes the design and analysis of all system parts, independently of any technical considerations about the final execution platforms and their embedded technologies; these are then refined into Platform Specific Models (PSM) and combined with Platform Description Models (PDM) to finally use model transformations for reaching the specific code running on the platform.

[5]Surprisingly, this claim was considered very innovative because it was seen as being able to reduce the development costs of frameworks and editors. Unfortunately, this is known from a very long time in other computer engineering fields: in compiler theory, *bootstrapping* is a common technique for developing a language's compiler with the language itself; in rewriting theory, theories can often (syntactically) described themselves (e.g., BNF grammars, or even Maude theories). These techniques do not prevent from paying an initial effort to implement the necessary machinery to make things work before being able to treat the language or the theory within the framework itself. As we already explained in detail in Section 2.1, the term *meta-metamodel* for the M3-layer is improper: the instantiation relationship between M1 and M2 on the one hand, and M2 and M3 and the other do not share the same nature: the former is *ontological* while the latter is *linguistic* (T. Kühne 2006), which is why the meta-metamodel can be syntactically treated as any other metamodel.

Although genuinely adopting Mde principles, the Mda approach is questionable in its fundamental philosophy. First, Uml, which was designed to address every stage of software development (from high-level specification to low-level implementation details), can clearly not achieve the abstraction goal developers need to properly address their numerous problems: an "universal" language is rarely efficient and flexible enough for every possible need without further adaptations (e.g., with profiles and stereotypes as discussed by Atkinson and T. Kühne 2007). Second, the Mda philosophy itself, consisting of separating computational aspects, functional features and platform-specific details, is difficult to achieve in practice: this leads to many intermediate models engineers have difficulties to understand because they are partially generated, and capture things they do not completely master; and these difficulties are enforced by the fact that all model levels need to be kept synchronised, which is hard to realise when modifications occurring at lower-levels have to be reflected back in higher levels.

Dsls adopt a different approach. Instead of creating models with a general-purpose language like Uml, the idea consists in specialising the language itself to focus only on a particular domain, and code generators are the ones that embed platform specificities to reach the final code directly from those high-level models: the generators are specific to a platform, not the models, and one generator is used for each specific platform.

2.3.1.4 Achieving Better Automation for Reducing Costs

The introduction of Dsmls, either in research projects or companies, clearly differentiates two different roles: developers in charge of developing the Dsml solution, and users of this solution in charge of creating models. Using Dsmls then decouples the model creation and modelling framework development activities, which is beneficial in several aspects.

On the one hand, *model creation* becomes more focused on the task at hand, allowing modelers to forget about the low-level coding and development tasks to "make things work". Since models directly reflect the expertise domain's concepts, it is easier to catch modelling errors at earlier stages, specifically because modelling languages can often include correctness rules that make ill-formed models difficult, or even impossible, to create. On the other hand, *modelling framework infrastructure* is harder than creating models, but this is the key point of Dsml solutions: by allowing model creation at a higher abstraction level, and automating all subsequent tasks (like persisting relevant information, checking model consistency, executing models, generating code, etc.), a significant productivity increase is usually observed, compared to pure code solution development. The decoupling also allows faster reaction to changes, and favor so-called "agile" development processes: models can be changed, maintained, or adapted to new requirements more easily because they do not directly impact the underlying infrastructure (unless, of course, modifications of the metamodel become necessary); and new functionalities to the Dsml framework can be added with a limited impact on existing models (Kelly and Tolvanen 2008).

Building a Dsml is nevertheless costly, even if this Mde technique includes several methodological advances for software development that could help reducing these costs. Kelly and Tolvanen (2008) studied the economic aspects of introducing Dsml in a company's life, and presented some evidence regarding the productivity gains, the cost reduction of the complete development of a software solution, and the maintenance costs after solution deployment. However, these results are considered anecdotal, because they are difficult to assess, reproduce, or even verify. The Mde methodology is still young, but some academic studies, more independant than the previous one, show that the benefits are real, even if it is not always at the expected level (cf. Barišic et al. 2014, for a review of existing studies).

2.3.2 Dsmls as Languages: Basic Components

We have seen so far that Dsmls epitomise the Mde methodology, but focus on a particular expertise domain. Dsmls take the opposite philosophy of what is intended with General-Purpose Languages: by focusing on a particular domain, they discard by nature the possibility of reuse in areas that go beyond their domain of interest.

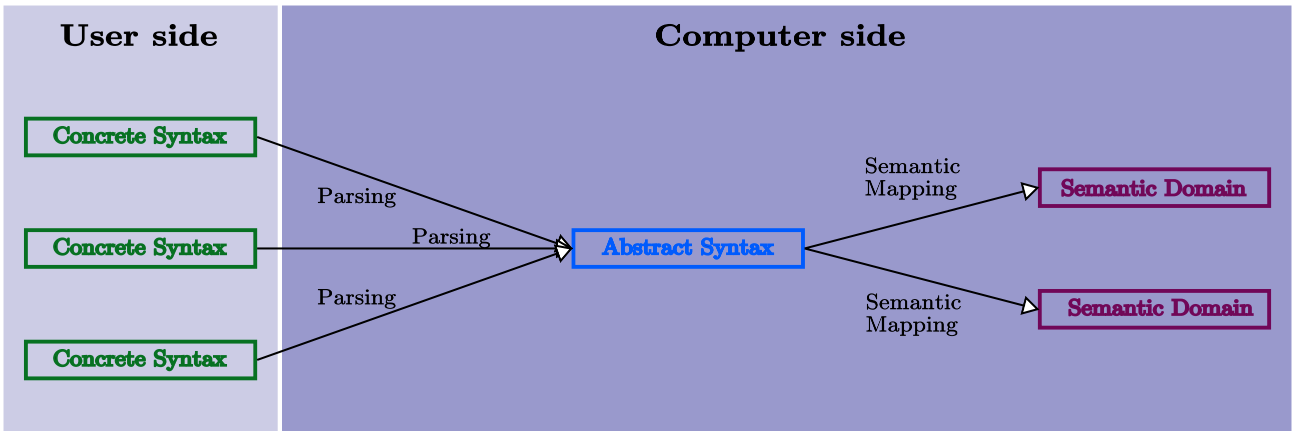


Figure 2.8 – Relations between DSML components: the *abstract syntax* is the central component and corresponds to a metamodel; *concrete syntaxes* allow end-users to manipulate the language, they are parsed to produce a conforming model; the *semantic domain*(s) will provide meaning(s) for the metamodel constructions through semantic mapping.

However, thanks to this restriction, DSMLs allow domain experts to achieve a better degree of automation when getting the final result without knowing all the low-level details.

However, from a theoretical point of view, DSMLs are still languages: it should be possible to study their essence with the classical tools available from research and practice on formal languages. Figure 2.8 relates the core components of every language (Harel and Rumpe 2004):

- the *form* of the language is described by an *abstract syntax*, which is an internal representation of the DSL concepts, and one (or several) *concrete syntax*(es), which serves as an interface with users to allow them manipulate the language;
- the *meaning* of the language is traditionally expressed using a *semantic domain* (or several ones, depending on the uses), into which abstract syntax concepts find their meaning through a *semantic mapping*.

We review each concept, and discuss existing approaches for defining DSLs' semantics, in contrast to the approach of this thesis. A concrete example of a simple DSL is given in Chapter 5, within the context of Kermeta.

2.3.2.1 DSL Syntaxes

In the context of MDE, an *abstract syntax* is always a metamodel, conforming to its meta-metamodel. The role of the abstract syntax is to model the concepts of the language and their relationships, as well as the structuring rules that constrain the metamodel elements and their combinations, in order to respect the domain's rule.

On the other hand, a *concrete syntax* has many forms, usually classified as *textual* or *visual* (although "pure" visual syntaxes are rare since they often include textual parts). A concrete syntax associates to model elements, or their combinations, a concrete representation that is manipulable by a user (which can already exist in the expertise domain). Dedicated tools offer nowadays the capability of assisting DSL designers when choosing a concrete syntax: manipulating textual representations is usually the easiest, since it already benefits from advances in the domain of compilation; however, dedicated tools capable of working at the level of metamodels, like XTEXT, help designers in generating grammars directly from the metamodels and, of course, in parsing strings according to these grammars. Managing visual syntaxes is perceived as more difficult, but several advances already provide trustable tools for managing such artifacts. For example, Costagliola et al. (2002) proposed VLCC, a graphical system for the automatic generation of visual environments; whereas Ráth, Ökrös, and Varró (2010) built a tool for synchronising abstract and concrete syntaxes, which is very convenient for following "live" execution of models.

From a theoretical viewpoint, it is interesting to compare how metamodelling and traditional compilation theory approach the syntax problem. When comparing the expressive power of textual grammars like BNFs and graphs underlying the metamodelling activity, the latter is strictly more powerful (A. Kleppe 2009): as a matter of fact, BNF's underlying mathematical structures are *trees*, whereas models are *graphs*. But this comparison is questionable in practice: these structures are never used blindly, but usually comes with constraints that change the general picture.

- Despite the metamodel's natural representation with graphs, many meta-metamodels impose structural constraints. For example, MOF has such a constraint, forcing to contain any metamodel element in a so-called *root* element, making the overall metamodel look more like a tree than a graph[6].
- If we compare the plain usage of both formalisms, grammars and metamodels are equally expressive if we consider the associated "*constraint language*": graphs alone are strictly more expressive than trees; but metamodels associated with their constraint language, such as the OCL standard of the OMG, are strictly equivalent to grammars associated with traditional type-checking techniques.

This again comes with no surprise: using metamodels would not bring us the capability of describing data structures that could not have been described with already existing programming language; furthermore, many metamodels are actually persisted using textual representations (and among all possible candidates, the most popular still remains XML in Eclipse-based MOF representations) without loss of information (Alanen and Porres 2003).

2.3.2.2 DSL Behaviour

When the concepts relevant for a DSL are defined, it becomes necessary to attach to the DSL a *behaviour* for it to be useful. This behaviour will obviously be defined, in the context of MDE, using transformations. Historically, DSML's behavioural aspects were not addressed at the beginning, but received proper attention only lately. Several overviews exist in the literature (Bryant et al. 2011; Combemale, Crégut, et al. 2009), that all share a classification along two dual approaches, namely *translational* and *operational* semantics, as already depicted in Figure 2.6. We present the rationale behind each approach, and emphasise their differences.

Translational Semantics is by far the most popular approach: it is *exogeneous*, meaning that it consists in translating the DSL metamodel and behaviour into another metamodel that serves as a target framework for execution. This technique has an obvious advantage: the DSL directly benefits from the target framework's additional capabilities, like testing or formal analysis, which is especially useful for big and complicated DSLs for which a dedicated tool is difficult to build. But this comes at a price: DSL designers need to learn the target language to define the semantics, and to be able to perform such tasks; furthermore, due to the potential difference of abstraction level between both metamodels, mapping results obtained in the target framework back to the original DSL metamodel can become tricky.

Depending on which transformation framework is considered formal or not, there exist several semantic targets in the literature complying with the translational approach. Abstract State Machines is an important formalism (Börger and Stärk 2003) that comes in two flavours: semantic anchoring (Chen, Sztipanovits, and Neema 2005) on the one hand, and semantic mapping, hooking and meta-hooking (Gargantini, Riccobene, and Scandurra 2009) on the other hand. Petri Nets are another target formalism that deals very well with state-based DSL semantics, which is exemplified by de Lara and Vangheluwe (2010). Graph-Based Transformations and Rewriting Logics share common verification capabilities like reachability analysis and model-checking. Among others, Rivera and Vallecillo (2007), later extended with real-time

[6]Graph-Based Transformation frameworks are a notable exception, since they truely use plain graphs to represent both metamodels (the so-called *typed graphs*) and models. However, as already explained in Section 2.2.2, some native MDE constructions like multiplicities or containments were addressed very recently and require special care.

behaviour by Riveira (Rivera, Durán, and Vallecillo 2010), formalise DSL behaviour using Maude (Clavel et al. 2007); whereas Arendt et al. (2010) with Henshin on the one hand, and citedeLaraVangheluwe2002 with AToM^3 on the other hand, are popular tools based on graph transformation enabling the definition of DSL semantics in a visual manner.

Operational Semantics is the second approach, popularised by the emergence of graph-based transformation engines: it is *endogeneous*, meaning that the semantics is expressed on the basis of the DSL constructs, showing how a conforming model evolves over time. This approach has a straightforward advantage: since DSL concepts are directly manipulated, the semantics is usually perceived as easier to define, especially when the DSL comes with a convenient concrete syntax familiar to the DSL designers. However, when it comes to performing further tasks on the DSL, this approach completely relies on the transformation engine's capabilities.

Kermeta clearly belongs to this category: a designer fills MOF operations bodies for performing the necessary changes to model accordingly to the DSL intended semantics. We just mention an interesting overview by Combemale, Crégut, et al. (2009), and provide an in-depth discussion of the related work in Chapter 7, where Kermeta's Action Language is formalised.

We do not insist more on DSML semantics: our main contribution is a semantic specification for Kermeta, which provides a real-life example on how to proceed for this matter.

2.4 Conclusion

This Chapter reviewed the central notions of MDE, namely models and model transformations for the purpose of formally specifying Kermeta, a model transformation framework. We explored the relationships of models and what they represent, emphasising the crucial difference between linguistic and ontologic instantiation, at the core of the mathematical definition of Kermeta's Structural Language. We reviewed existing techniques and tools for model transformations, as well as existing classifications, to notice that from a verification perspective, they fail at providing a concrete way to design a general method for ensuring transformation correctness. Finally, we provided an overview of one particular MDE technique, DSLs, both from an engineering and a mathematical viewpoint. This founded the motivation of our work: by providing a formal semantics of the transformation language itself, any DSL whose behaviour is expressed in this transformation language acquires automatically a formal counterpart.

The following Chapter studies the second dimension of our Thesis, namely Formal Verification, with the underlying goal of achieving a systematic methodology for ensuring model transformation correctness.

3 Formal Verification

After a decade of theoretical developments, and the substantial progress regarding technical advances and tool assistance, Mde gained the maturity for being used in various domains, spanning from small home-made process improvements within companies for automating redundant tasks, to complete product lines allowing to manage the software products' variability (Weiss and Robert Lai 1999). More recently, Mde has been applied to safety-critical and embedded applications and systems In these domains, misconceptions or failures may have dramatic consequences, either in terms of market loss (e.g., the famous Intel Pentium bug) or in term of human lives (e.g., the Ariane crash). This will irremediably call for better techniques for ensuring the correctness of model transformations which, in turn, will largely benefit to the entire practice of Mde.

Several techniques have correctness in mind, at different levels. The simplest one starts with developers' habits, when they debug their own production. For Mde, this is still unsufficiently explored, even if noticeable progress has been done recently (see for example). Testing and formal analysis are common techniques for validating software artifacts. In particular, formal analysis (or methods) covers any technique based on mathematics for the purpose of specification, development, or verification of software (or hardware) systems.

This Chapter provides a background in the domain of formal verification to help position our contribution. Note however that this Thesis does not contribute to the domain of verification itself (e.g. by defining a new logic for a new class of properties, or new datastructures for improving model-checking algorithms), but rather makes an experimental application of pre-existing verification tools in the context of the Kermeta framework. Section 3.1 properly defines what we understand by *formal verification*, and defines some concepts and terminology. Section 3.2 presents a state-of-the-art of the practice of formal verification in the context of model transformation. Maude, as a specification and a verification tool, is extensively covered in Part III. Overall, this Chapter contributes by identifying the key ingredients of model transformation verification, and by an extensive review of the literature that provides an up-to-date snapshot of the current practice in the domain.

3.1 The Verification Problem

In Computer Science, a *decision problem* is a problem meant to be solved using mechanised ways (i.e., using a computer), and whose answer is boolean, i.e. either `Yes` or `No`. Decision problems can be classified according to their "*solvability*": if it is possible to build an algorithm to solve the problem in finite time, the problem is said *decidable*, and *undecidable* otherwise; and *semidecidable* if such an algorithm exists for only one case (typically, for answering `Yes`), that is not guaranteed to terminate for the other case. The well-known *Halting Problem*, which consists in determining if a given program executed on a given set of data will terminate or not, is a classical example of undecidable problems (P. Cousot and R. Cousot 2010).

Shortly stated, the *Verification Problem* is a decision problem that consists in answering (non-trivial) questions about all possible executions of a computation. For reaching a more precise definition, we first need to precise the exact meaning of the involved concepts: what are a *computation execution* and a *question* in the context of the verification problem.

Execution The execution of a program, a transformation, or any piece of software or hardware material, can be influenced by several factors, internal or external. In particular, the environment can be a source of large variations: it can be a human interacting with the program, or the physical world whose data are captured by sensors and captors. We call *concrete semantics* a model of an execution under all possible environments. It basically captures the values of all possible variables involved in the program's execution, either internal, i.e. for maintaining information during the execution; or external, i.e. input/output variables used for communication with the environment. A possibility to represent this concrete semantics consists of reporting how these variable values evolve over time: Figure 3.1 shows such a description, where the values are summarised in a variable vector $x(t)$, and are represented as a trajectory over time. As one can expect, this is mainly a mathematical construction, represented here conceptually, rather than a representation of what is happening in the reality: as already highlighted, this construction is often infinite even for very simple programs.

Property In the context of verification, a property captures a set of variable values that characterise *erroneous states* (or *forbidden zones*), i.e. which states are considered, during the computation, as not desirable because they can cause damages to the system or its environment. Figure 3.2 depicts these states using red areas.

The *Verification Problem* can now be precisely stated: we want to *prove* that independently of the given input variables, all executions, represented by their trajectories, never cross the forbidden zones representing the erroneous states. Mathematically speaking, this proof establishes that the intersection of the set of states for the concrete semantics and the set of erroneous states is always empty. As we expected, this problem is generally undecidable (at least for "interesting" questions): it is not always possible to answer this question completely automatically, i.e. without any human intervention, using finite computer resources, without any uncertainty.

In the rest of this Section, we discuss some key characteristics of verification approaches, then review classical verification techniques.

3.1.1 Characteristics of Formal Verification Approaches

Formal verification techniques have several characteristics that makes them reliable, but difficult to perform on real-life programs. We contrast and illustrate these characteristics by opposition to other techniques, not considered as verification *per se*. A formal verification technique is

Offline (or *static*): applying the technique should not require actually executing the analysed program. This implies that the analysed program is treated as an input of the analysis, rather than instrumenting it to extract relevant data for its correctness.

Exhaustive (or *full covering*): a verification technique covers all possible execution paths without exception, which allows to certify the absence of errors under all possible circumstances.

Sound (or *correctly abstracting*): if the technique concludes to the absence of errors, then it implies there are actually no errors (at least, for the range of properties checked, not in the absolute).

Exactitude (or *preciseness*): the technique delivers a boolean answer (in compliance with the definition of a decision problem).

Since computing the concrete semantics is not feasible in practice, it becomes necessary to operate an *approximation*: instead of computing each and every step of the execution, several states are amalgamated together to reduce their number. This is usually referred to as an *abstraction*, leading to the computation of an *abstract semantics*. To ensure the validity of the results, the abstract semantics is an over-approximation: this way, proving that properties hold on the abstract semantics mathematically ensures that the same properties hold on

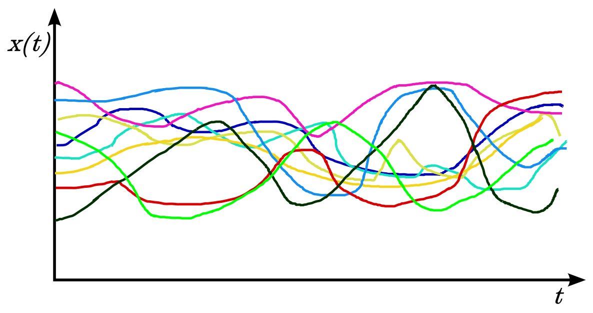


Figure 3.1 – Representation of a program's concrete semantics: all values are captured by a single vector $x(t)$ evolving over time.

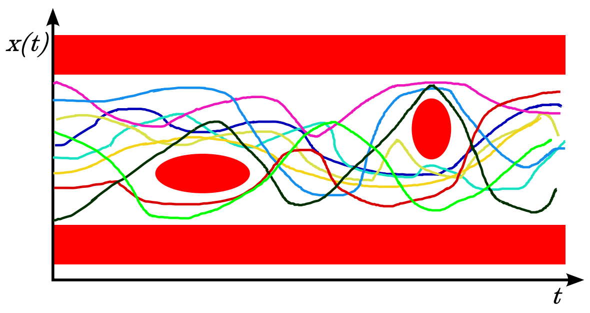


Figure 3.2 – Error zones, in red, are variable values that are forbidden, i.e. that represent bad, non-desired behaviour.

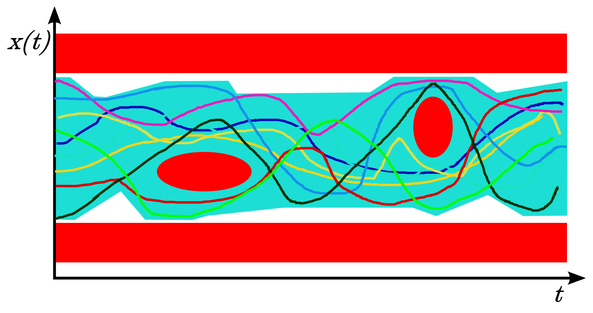


Figure 3.3 – Correct Abstraction (light green area) as an over-approximation of the concrete semantics, i.e. covering all possible execution paths.

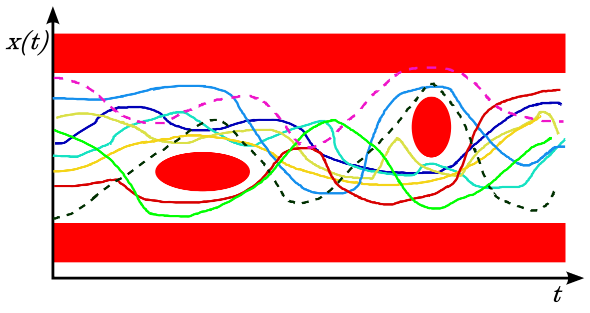


Figure 3.4 – Representation of *Testing*: plain lines are the ones tested; since not all paths are covered, one can miss erroneous scenarios (in dashed pink).

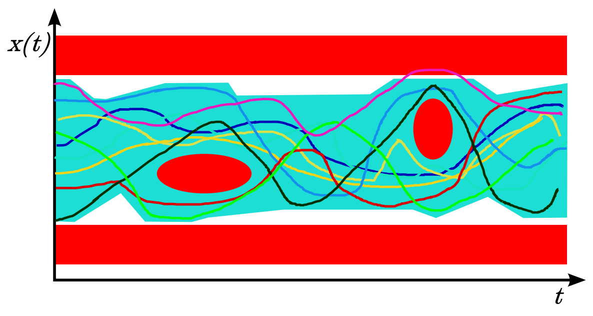


Figure 3.5 – Erroneous abstraction: not being an over-approximation, the erroneous part of the pink path is not detected.

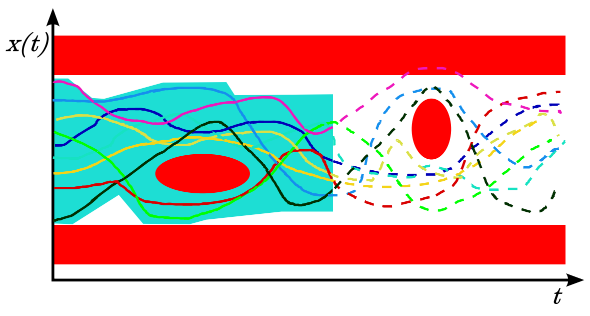


Figure 3.6 – Bounded Model-Checking checks all paths, but only up to a certain time, so that late errors (in pink path) are not detected.

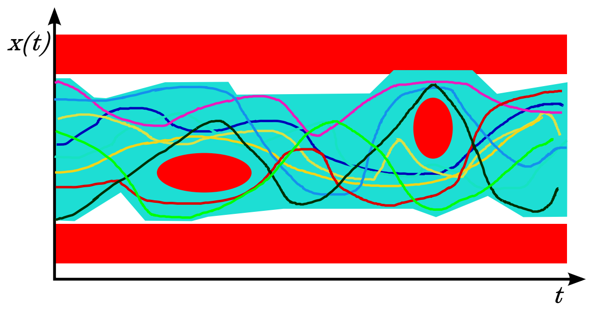


Figure 3.7 – *False alarms* can occur when coarse-grained abstractions are used: detected errors do not correspond to an concrete execution path.

the concrete one (since the abstract semantics covers more states than the concrete one). To still stay tractable, an abstraction should balance between *computability*, i.e. the capacity to model it in order to reason about the program, and *accuracy*, i.e. not abstracting too many details in order to preserve the satisfiability of the properties one is interested in. Figure 3.3 shows a possible abstraction in green: since it encompasses all possible trajectories, if the green zone does not intersect the error zones in red, so does the actual execution.

From a practical viewpoint, the previous four properties represent an idealistic situation that is sometimes difficult to reach. In order to propose tractable analysis, tools have to reach a balance between these properties, and sometimes even break them to propose "lightweight" and "quicker" analysis. We review some classical techniques and link them with the previous properties.

- Figure 3.4 illustrates the testing approach: testing relaxes exhaustivity, and explores only a few possible paths for assessing that they do not cross the red zones. As a consequence, if an error happens in a path not explored for any reason (e.g., not captured by test cases, or not selected at all), it is not detected.

- Figure 3.5 depicts an unsound abstraction, which is therefore incorrect. This case happens when an abstraction is defined as not being an actual superset of all possible executions: an error can occur in those path fragments not captured by the approximation.

- Figure 3.6 depicts another unsound abstraction, but for a different reason: it effectively covers all possible executions, but only up to a certain time. This technique assumes the so-called *small scope hypothesis*, stating that if there is an error, it is likely to appear very early in the execution. By doing so, decisions procedures are considerably accelerated, since only a limited prefix is explored, at the expense of potentially missing all late errors. This technique is also known as *bounded* model-checking, and used e.g. in Alloy Jackson 2011.

- Another possibility is to relax the decision problem's *answer*. For example, semi-decidable algorithms combined with soundness lead to decision procedures that ensure the safety of a system if no errors are detected, but can loop indefinitely or return erratic answers. This kind of techniques is problematic because no indication for fixing the errors are given.

  Another path is *probabilistic analysis*: instead of a boolean answer, a correctness score is returned that indicates to which extent the set of properties hold.

- A last possibility is to require *human assistance* to overcome the complexity. This is often the case for theorem-provers like Coq (Bertot and Castéran 2004) or Isabelle/HOL (Nipkow, Paulson, and Wenzel 2013): the user guides the proof towards the goal when automated procedures fail.

Formal verification techniques suffer from well-known issues that hinder their use and their results. We discuss some of them in the light of their adaptation to model transformation verification.

Explosion problem When not efficiently balanced, the exhaustivity requirement induces that execution representations can become so large that it hinders the analysis itself: completely exploring all possible states can take unreasonable time. For complex transformations, one can expect the same problem.

False Alarms Approximating executions leads to the simple consequence that more behaviours are represented than the actual ones, sometimes even executions that do not occur in the reality. Sometimes, this leads to detecting errors called *false alarms*, i.e. errors that have no counterpart in the concrete program execution. Eliminating such alarms is a crucial challenge for enabling formal verification to be used in more diverse contexts. This situation is depicted in Figure 3.7: the green light zone corresponding to the approximation crosses the top red zone, although there is no erroneous trajectory corresponding to an actual execution path.

Backward Traceability When a property is violated, verification techniques usually produce so-called *counterexamples*, expressed in terms of the abstract semantics. Relating counterexamples back to the concrete semantics would help interpreting the results more appropriately in the formalism familiar to the programmer. However, abstractions are often difficult to reverse, with the consequence that the programmer becomes forced to understand the abstract semantics to hope for correcting the discovered errors. This is, together with the explosion problem, one of the main challenges in formal verification.

Heavy Mathematical background The requirements for formal verification techniques, namely exhaustivity and soundness, imply that their formal background is sometimes seen as difficult to learn, and develop. This also explains why this kind of analysis techniques were mostly used in contexts where the costs of deploying them counterbalances their investment returns: hardware systems, safety-critical and embedded systems and applications were for decades the main target for formal verification, because a failure in these applications could cost millions, or worst, be life-threatening.

These issues are intrinsically connected to the nature of formal verification and the requirements that were presented earlier. However, to enable a wide adaptation of these techniques to Mde, it is crucial to study how to leverage them and adapt them to the context of model transformations. In particular, transformations defining Dsls behavioural semantics carry the same issue, since they represent the Dsl execution. However, because of the higher level of abstraction, it should be possible to build execution approximations that would allow their analysis. The second challenge is the versatility of Mde compared to traditional software development: typically, building models and transformations is an agile process that calls for many development rounds, each of them being potentially submitted to analysis. This typically breaks the usual way verification is made, since the full definition of the execution is necessary before exhaustively exploring it for errors.

3.1.2 Common Verification Techniques

This Section briefly reviews common verification techniques by explaining their key conceptual differences. As recalled earlier, the verification problem has three components: an *abstract semantics* that approximates the "real" concrete semantics; a specification of the properties of interest; and a way to check the latter against the former.

The goal of a verification tool is to automate the checking process, given the other components. The main variation points for verification techniques are then the following: which (sometimes implicit) *form* has the approximation of the program, and which *role* has the user in building it; and the *flexibility*, or the range of properties, the technique is able to handle. Since only practical considerations are important for us, rather than the theoretical considerations, we only briefly review the main verification techniques in light of the previously mentioned variation points (P. Cousot and R. Cousot 2010).

Static Analysis consists in a predefined approximation, generally automatically pre-computed over the analysed entity. Sometimes, the user can parametrise the analysis in order to focus on specific properties. This technique is nowadays well mastered, and accompanies very often development tools for providing assistance and computing relevant information from a program to help developers better understand their programs and correct simple mistakes. Different forms of static analysis have been popularised during the last decades.

Some techniques are very old, because they were studied for optimising compilation of programs (e.g., live variables or dead code detection, for optimising registry allocation and code generation). Other techniques emerged with new programming paradigms like object orientation (e.g., inheritance hierarchy for tracking fields/methods redefinition and overloading; graph calls for better understanding objects interactions, escape and shape analysis for improving data representation).

Model-Checking consists in approximating programs into finite dynamic structures (most of the time, labelled transition systems), and providing a formal logic for expressing properties of interest independently from the program. Finite structures obviously result from the fact that some details are lost during the process: for example, for detecting deadlocks, it is often enough to forget about the actual values of variables and focus solely on the interactions executed in parallel.

The user has to provide the finite representation; the model-checker basically automates the checking. Most of the time however, this finite representation is automatically extracted from the program itself, using static analysis techniques, which relieves the user from this burden. A model-checker usually provides a violation trace if the property does not hold: this trace indicates a scenario leading to a violation, which helps the user figuring out what went wrong and where. However, due to the usual semantic gap between program languages and model-checker representations, it is often a real challenge to trace violation paths back to the original program.

Theorem-Proving consists of specifying a program by means of inductive properties satisfying verification conditions. Basically, properties of interest have the form of predicates whose truth is checked against the inductive properties representing the program.

The user has to provide such a specification; the theorem-prover basically automates the proof burden, i.e. the fact that these properties are indeed inductive. However, due to the undecidability issue, the theorem prover sometimes needs guidance to fully discharge the proof. The specification can be partially discovered directly from the program by using static analysis techniques.

Abstract Interpretation somehow generalises the previous approaches by allowing any kind of approximation to be defined. The user then has then to prove that the proposed approximation is sound, and the abstract interpreter automates the verification process for the properties. By using predefined abstract domains, the approximation can eventually be automatically computed, but this restricts the range of properties the abstract interpreter is capable of checking.

3.2 A Tridimensional Approach

This Section proposes a classification of Model Transformations' verifiable properties: it brings an interesting snapshot of the current state-of-the-art in this area, thus enabling reasoning about evolution and trends of this research domain. In order to provide a comprehensive interpretation, we propose a tridimensional approach, as depicted in Figure 3.8, that allows locating each contribution in the literature regarding the constitutive characteristics for model transformation verification: the *transformation* involved; the *property kinds*; and the *verification technique*. These dimensions are closely related, but clearly independent. This work allows a better understanding of the expected properties for a particular transformation, and facilitates the identification of suitable tools and techniques for enabling their verification.

3.2.1 Transformations

Since the *Transformation* dimension is already thoroughly reviewed in Section 2.2, we only recall the main elements for the purpose of this approach:

Definition We reviewed the definitions existing in the literature, and proposed a broader and more complete one in Definition 2.2, that puts more emphasis on transformations' *intent*;

Languages We proposed a general definition of both levels a transformation operates at, namely the *specification* and the *execution* levels (cf. Definitions 2.3 and 2.4) and explored the various transformation languages;

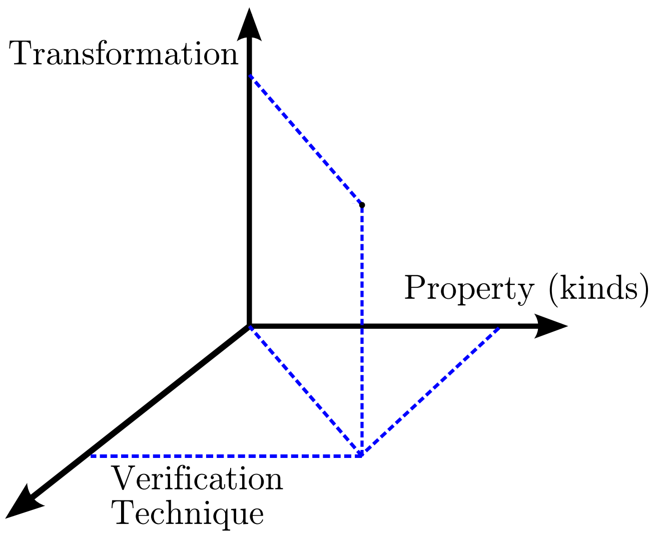


Figure 3.8 – Model Transformation Verification Tridimensional Approach

Classifications We reviewed existing classifications for model transformations: one is based on transformation's *features* (Czarnecki and Helsen 2006), the other is based on the transformation's *form* (Mens and Van Gorp 2006). However, both fail at providing an efficient way to associate transformations properties according to their intents.

3.2.2 Properties

Expressed in a particular Transformation Language TL, model transformation specifications relate source and target metamodel(s) and execute on models. Considering only conforming models for transformations to be valid is not enough: due to the large number of models transformations can be applied on, one has to ensure their validity by carefully analysing their properties to provide modelers a high degree of trustability when they use automated transformations to perform their daily tasks.

This Section explores properties one may be interested in for delivering proper and valid transformations. Following the dual nature of transformations, we identified two classes of properties: the first class in Sec. 3.2.2.1 relates to the computational nature of transformations and targets properties of TLs; whereas the second class in Sec. 3.2.2.2 deals with the modelling nature where models plays a prominent role. Table 3.1 summarises the reviewed papers according to the property kinds identified hereby.

3.2.2.1 Transformation Models: Language-Related Properties

From a *computational* perspective, a transformation specification conforms to a *transformation language* (cf. Fig. 2.5), which can possess properties on its own. In the MDE context, two properties are interesting at execution time: *termination*, which guarantees the existence of target model(s); and *determinism*, which ensures uniqueness. These properties qualify the execution of transformations written in such languages. Another property, namely *typing*, relates to design time: it ensures the well-formedness of transformation specification, and can be seen as the TL's *static semantics*.

Because they hold at the TL's level, these properties directly impact the execution and design of all transformations. Therefore, formally proving them cannot be done by relying on one particular transformation's specifics. TLs adopt one of the following strategies for proving execution time properties hold: either the TL is kept as general (and powerful) as possible, making these properties undecidable, but the transformation framework provides capabilities for checking *sufficient conditions* ensuring them to hold on a particular transformation; or these properties are ensured *by construction* by the TL, generally by sacrificing its expressive power.

3.2.2.1.1 Termination Termination directly refers to Turing's halting problem, which is known to be undecidable for sufficiently expressive, i.e. Turing-complete, TLs: Graph-Based Transformations (GBTs) have already been proven to not terminate in the general case (Plump 1998); whereas MPLs often use loops and (potentially recursive) operation calls, enabling them to simulate Turing machines.

Termination Criteria A large literature for GBT already exists, which makes exhauste coverage beyond this paper's scope. H. Ehrig, K. Ehrig, et al. (2005), introduce layers with injective matches in the rewriting rules that separate deletion from non-deletion steps. Varró, Varró-Gyapai, et al. (2006) reduce a transformation to a Petri Net, where exhaustion of tokens within the net's places ensures termination, because the system cannot continue any more. Bruggink (2008) addressed a more general approach by detecting in the rewriting rules infinite creating chains that are at the source of infinite rewritings. Küster (2006) proposes termination criteria with the same base idea, but on graph transformations with control conditions.

Termination criteria for MPLs directly benefit from the large and active literature on imperative and object-oriented programming languages. They usually rely on abstract interpretations built on top of low-level programming artefacts (like pointers, variables and call stacks). For example, Spoto, Hill, and Payet (2006) detect the finiteness of variable pointers length; and Berdine et al. (2006) use separation logics for detecting effective progress within loops. Since these techniques are always over-approximations of the TL's semantics, they are sound but not complete, and can potentially raise false positives.

Expressiveness Reduction Reducing expressivity regarding termination generally means avoiding constructs that may be the source of (unbounded) recursion. For example, DSLTrans (Barroca et al. 2010) uses layered transformation rules and an in-place style: rules within a layer are executed until they cannot match anymore, which occurs because models contain a finite amount of nodes that are deleted in the process, preventing recursions and forbidding loops syntactically.

3.2.2.1.2 Determinism Determinism ensures that transformations always produce the same result. Generally, this property is only considered up to the interactions with the environment or the users. Considering this, MPLs are considered deterministic since they directly describe the sequence of computations required for the transformation.

Determinism Criteria Determinism directly refers to the notion of *confluence* (often called the *Church-Rosser*, or *diamond*, property) for GBTLs, which has also been proven undecidable (Plump 2005). Confluence and termination are linked by Newman's lemma (Newman 1942), stating that confluence coincides with local confluence if the system terminates. This offers a practical method to prove it by using the so-called *critical pairs*. The GBT community is very active and already published several results. Heckel, Küster, and Taentzer (2002) formally proved the (local) confluence for Typed Attributed Graph Grammars, and Küster (2006) for graph transformations with control conditions. Lambers, H. Ehrig, and Orejas (2006) improved the efficiency of critical pairs detection algorithms for transformations with injective matches, but without addressing pairs of deleting rules. More recently, Biermann (2011) extended the result to EMF (Eclipse Modelling Framework), thus preserving containment semantics within the transformations. In another area, Grønmo, Runde, and Møller-Pedersen (2011) addresses the conformance issue for aspects, i.e. ensuring that whatever order aspects are woven, it always leads to the same result.

Expressiveness Reduction Reducing expressivity regarding confluence means either suppressing the possibility of applying multiple rules over the same model, or providing a way to control it. In DSLTrans for example (Barroca et al. 2010), the Tl controls non-determinism occuring within one layer by amalgamating the results before executing the next layer. This ensures confluence at each layer's execution, and thus for a transformation.

3.2.2.1.3 Typing A crucial challenge for transformation specification is the detection of syntactic errors early in the specification process, to inform designers as early as possible and avoid unnecessary execution that will irremediably fail. This property primarily targets visual modelling languages, since textual modelling already benefits from experience gathered for building Ides for Gpls, where a type system (usually static) reports errors by tagging the concerned lines. All syntactic errors cannot be detected, but a framework possessing this feature will considerably ease the designers' work.

To achieve this goal, tools must rely on an explicit modelling of transformations (Bézivin et al. 2006). T. Kühne et al. (2009) studied the available alternatives for this task and their implications: either using a dedicated metamodel as a basis for deriving a specialised transformation language, or directly using the original metamodel and then modulating the conformance checkings accordingly, for deriving such a language. Studying the second alternative, they proposed the Ram process (Relaxation, Augmentation, Modification) that allows the semi-automatic generation of transformation specification languages. On the other hand, Levendovszky, Lengyel, and Mészáros (2009) explored in the other alternative by proposing an approach based on Domain-Specific Design Patterns together with a relaxed conformance relation to allow the use of model fragments instead of plain regular models.

3.2.2.2 Model Transformations: Transformation-Related Properties

From a *modelling* perspective, a transformation refers to source/target models (cf. Fig. 2.5) for which dedicated properties need to be ensured for the transformation to behave correctly. Of course, the properties of interest range over a large scale of nature, precision, and complexity.

This section provides a comprehensive overview of properties of interest based on their kinds: properties involving transformations's source and/or target models were historically the first to be considered; then, syntax-guided properties, which relate models' syntaxes (i.e. their metamodels) provide a first level of analysis; and finally, properties involving the underlying semantics of models are discussed.

3.2.2.2.1 On the Source/Target Model(s) A first level of property verification concerns the source and/or target model(s) a transformation refers to. The *conformance* property is historically one of the first addressed formally, because it is generally required by transformations to work properly. Transformations admitting several models as source and target require other kinds of properties, either required for transformations or simply desirable.

Conformance & Model Typing Conformance ensures that a model is valid w.r.t. its metamodel, and is required for a transformation to run properly. Usually, *structural* conformance, involving only the model, is distinguished from *constrained* conformance, which is an extended property that includes structural constraints, otherwise referred to as metamodels' *static semantics* or *well-formedness rules* (see e.g. Boronat 2007). Nowadays, this property is well understood and automatically checked within modeling frameworks for input models. However, proving that a transformation always outputs conform target model(s) is not trivial, especially when using Turing-complete frameworks. Most of the time, an existing procedure for checking conformance is programatically executed after the transformation terminates. Model Typing (J. Steel and Jézéquel 2007) extends the notion of type beyond classes, by defining a *subtyping* relation on models. This enables better reuse for modelers: transformations defined for particular models also work for any sub-model.

N-Ary Transformations Properties Unsurprisingly, transformations operating on several models at the same time, e.g. model composition, merging, or weaving, require dedicated properties to be checked.

Concerning *merging*, Chechik, Nejati, and Sabetzadeh (2011) follow an interesting research line: they enunciate several properties merge operators should possess: *completeness* means no information is lost during the merge; *non-redundancy* ensures that the merge does not duplicate redundant information spread over source models; *minimality* ensures that the merge produces models with information solely originating from sources; *totality* ensures that the merge can actually be computed on any pair of models; *idempotency*, which ensures that a model merged with itself produces an identical model. These properties are not always desirable at the same time: for example, completeness and minimality become irrelevant for merging involving conflict resolution. Beyond merging, they can potentially characterise other transformations, not necessarily involving several source models.

Concerning *aspect weaving*, Katz (2006) identifies temporal logics to characterise properties of aspects: an aspect is *spectative* if it only changes variable within this aspect without modifying other system variables or the control flow; it is *regulative* if it affects the control flow, either by restricting or delaying an operation; it is *invasive* if it changes system variables. Static analysis techniques enriched with dataflow information or richer type systems are generally used to detect these properties. Despite their textual programming orientation, these properties should apply equally in MDE. Molderez et al. (2010) present DELMDSoC, a language for Multi-Dimensional Separation of Concerns implemented in AGG (Taentzer 2000). Ultimately, this framework will allow the detection of conflicts between aspects by model-checking, typically when multiple advices must be executed on the same joinpoints.

3.2.2.2.2 Model Syntax Relations A “lightweight” approach for ensuring that a transformation behaves correctly is to look at the outputted artefacts: a transformation could be considered as correct if certain model elements, or structures, of the input model are transformed into other elements or structures of the output model. Of course, the fact that these structures are a correct match has to be defined by the user who knows how the transformation is supposed to behave. This kind of property is called *model syntax relations*: they relate the *shape* of an (set of) input model(s) with the shape of an (set of) output model(s), leaving the actual semantics of input and output models implicit, and manipulating it symbolically through these structures.

Akehurst, Kent, and Patrascoiu (2003) formally introduce a set of structural relations between the metamodels of the abstract syntax, concrete syntax and semantic domain of a fragment of UML, by creating an intermediate structure that relates the elements of both metamodels as well as the elements of the intermediate structure itself. Although only apply it to an academic example, the proposed technique appears to be sufficiently well founded to be applied in a more generalised case. (Narayanan and Karsai 2008b) also define a language for defining structural correspondences between metamodels that take into consideration the attributes of an entity in the metamodel. In particular they apply their approach to verifying the transformation of UML activity diagrams into the CSP (Communicating Sequential Process) formalism. They also point out that Triple Graph Grammars (TGGs) (Schürr and Klar 2008), the formalism used for defining model transformations, could also be used to encode structural relation properties between two metamodels. Lúcio, Barroca, and Amaral (2010) formally define a property language to express structural relations between two language's metamodels and propose a symbolic technique to verify those relations hold, given an input and an output metamodel, and a transformation.

3.2.2.2.3 Model Semantics Relations Beyond structural relationships between source and target models, it may be interesting to relate their meaning, which implies to dispose of at least a partial explicit representation of the models' semantics, or a way of computing it.

An example of such a semantics relation property could the fact that a statechart is transformed into a *bisimilar* statechart. In this case the relation between the semantics of these two behavioral models is *bisimulation* – which is a strong variant of a *simulation* relation – where both systems are able to simulate each other from an observational point of view. In order to prove automatically such a relation is enforced by a transformation it is necessary to build, explicitly or implicitly, the labelled transition systems corresponding the semantics of both the input and the output model.

Narayanan and Karsai (2008a) show how to verify that a transformation outputs a statechart bisimilar to an input statechart. Combemale, Crégut, et al. (2009) formally prove the weak simulation of xSPEM models transformed into Petri Nets, in the context of the definition of translational semantics of Domain-Specific Languages, thus enabling trustable verification of properties on the target domain.

The fact that two systems are able to simulate each other pertains to the observational behavior of those systems. One may wish to enforce a relation between the actual states of the behavioral input and output models. Varró and Pataricza (2003) are able to prove that CTL (Computation Tree Logic) properties are preserved when transforming statecharts into Petri Nets. Several contributions addressed in the recent years the formal verification of temporal properties. The idea consists in representing metamodels, models and transformations in an external formal framework already equipped with verification capabilities, generally delegated to a dedicated tool. Among others, Gargantini, Riccobene, and Scandurra (2010) use Abstract State Machines within the AsMeta framework to perform CTL verification using SPIN; Rivera, Durán, and Vallecillo (2009) use the Maude rewriting system and its embedded LTL model-checker to verify semantic properties of Domain-Specific Languages.

An interesting subset of CTL are *safety* properties, which are expressed as invariants over the reachable states of the system. The idea is that certain conditions can never be violated during execution. In this sense, Becker et al. (2006) are able to prove that safety properties (expressed as *invariants*) are preserved during the evolution of a model representing the optimal positioning of a set of public transportation shuttles running on common tracks. Given the evolution of the model is achieved by transformation, the safety properties will enforce that the shuttles do not crash into each other during operation. Padberg, Gajewsky, and Ermel (1997) introduce a morphism that allows preserving invariants of an Algebraic Petri Net when the net is structurally modified. Although this work was not explicitly created for the purpose of model transformation verification, it could be used to generate a set of model transformations that would preserve invariants in Algebraic Petri Nets by construction.

Models may have a *structural* semantics, rather than a *behavioral* semantics. This is the case of UML *class diagrams*, which semantics is given by the *instanceOf* relation. In this case, although the behavioral properties mentioned above do not apply, relations between the structural semantics of input and output models may still be established. Massoni, Gheyi, and Borba (2005) present a set of refactoring transformations that preserve the semantics of UML class diagrams.

3.2.2.3 Summary

Table 3.1 classifies the literature contributions we reviewed according to the property classes they are targeting, based on the dual nature of model transformations. This research emphasised two levels property classes are operating at: at the level of *transformation languages*, the property classes correspond to those of any computational language; at the level of *model transformations*, we distinguished three classes: *source/target* properties are only interested in the models involved in a transformation, without any consideration about the computation itself; whereas the transformation is taken into account with *syntactic* and *semantic relations*.

Not suprisingly, termination and determinism are broadly explored for graph-based approaches, since they are recurrent issues for this kind of model transformation frameworks. The first property class, namely source/target, represents a lightweight possibility for ensuring transformation correctness, without delving into the specific details of a transformation. Obviously, semantic properties are harder to cover, and therefore less explored, often by simply reusing, or readapting existing techniques for general purpose programming languages.

Language-Related		Transformation-Related	
Termination	H. Ehrig, K. Ehrig, et al. (2005) Varró, Varró-Gyapai, et al. (2006) Bruggink (2008) Küster (2006) Spoto, Hill, and Payet (2006) Berdine et al. (2006) Barroca et al. (2010)	**Source/Target**	Chechik, Nejati, and Sabetzadeh (2011) Katz (2006) Molderez et al. (2010) Boronat (2007)
Determinism	Küster (2006) Barroca et al. (2010) Lambers, H. Ehrig, and Orejas (2006) Biermann (2011) Grønmo, Runde, and Møller-Pedersen (2011)	**Syntactic Rel.**	Akehurst, Kent, and Patrascoiu (2003) Narayanan and Karsai (2008b) Schürr and Klar (2008) Lúcio, Barroca, and Amaral (2010)
Typing	T. Kühne et al. (2009) Levendovszky, Lengyel, and Mészáros (2009)	**Semantic Rel.**	Narayanan and Karsai (2008a) Varró and Pataricza (2003) Becker et al. (2006) Padberg, Gajewsky, and Ermel (1997) Massoni, Gheyi, and Borba (2005)

Table 3.1 – Classification of Contributions according to Property Kinds

3.2.3 Formal Verification Techniques

In this section, we discuss FV techniques proposed in the literature to prove MT properties implemented in various TLs. Table 3.2 captures our classification of MT verification techniques, which fall into one of three major types: *Type I* FV techniques guarantee certain properties for all transformations of a TL, i.e. they are *transformation independent and input independent.* Techniques of *Type II* prove properties for a specific transformation when executed on any input model, i.e. they are *transformation dependent and input independent.* Techniques of *Type III* prove properties for a specific transformation when executed on one instance model, i.e. they are *transformation dependent and input dependent.* When a FV technique is transformation independent, it implies that no assumption is made on the specific source model: this explains why, in Table 3.2, the cell representing FV techniques that are transformation-independent and input-dependent is empty.

Although applicable to any transformation, Type I verification techniques are the most challenging to implement since they require expertise and knowledge in formal methods. Type III verification techniques are the easiest to implement and are considered "light-weight" techniques since they do not verify the transformation *per se*; they verify one transformation execution. Across all the three types of verification techniques, the approaches used often take the form of model checking, formal proofs or static analysis.

Different properties discussed in section 3.2.2 were verified in the literature using different techniques from the three types. Some properties (e.g. termination) were proved only once by construction of the model transformation or the TL. Proving such properties required less automation and more hand-written mathematical proofs, although some studies used theorem provers to partially automate the verification process. Type I verification techniques were used to prove such properties. Other properties (e.g. model typing and relations between input/output models) were proved repeatedly for different transformations and for different inputs. Proving such properties required more automated and efficient verification techniques from Type II and Type III techniques. In the following subsections, we discuss examples of each type of verification technique from the literature.

3.2.3.1 Type I: Transformation-Independent and Input-Independent

Properties that hold independently of the transformations expressed in a particular TL are normally proved once and for all. Performing tool-assisted proofs is cumbersome: it requires to reflect the semantics of the underlying TL directly in the FV tool, which is a heavy task. For example, formally proving termination for GBT with a theorem-prover requires to express the semantics of pattern-matching in the theorem-prover's language. Therefore, these kinds of proofs are usually presented mathematically. Barroca et al. (2010) prove termination and confluence of DSLTrans inductively, following the layered syntactic construction of the language's transformations. H. Ehrig, K. Ehrig, et al. (2005) address termination of GBTs inductively, by proving termination of deleting and non-deleting layers separetely. Küster (2006) proposes sufficient conditions for termination and confluence of GBT with control conditions, by formalising the potential sources of problems within the theory of graph rewriting.

Another proof strategy consists of taking advantage of existing machinery in a particular formalism. For example, several techniques for proving termination exist for Petri Nets. The challenge is then to provide a correct translation from the metamodeling framework and the TL into the Petri Nets technical space. Varró, Varró-Gyapai, et al. (2006) prove termination by translating GBT rules into Petri Nets and abstracting from the instance models (i.e. the technique becomes input-independent); the proof then uses the Petri Nets algebraic structure to identify a sufficient criterion for termination. Padberg, Gajewsky, and Ermel (1997) prove that safety properties, expressed through invariants over Algebraic Petri Nets, transfer to refined Nets if specific modifications of the Nets are followed, thus guaranteeing the preservation of these safety properties. Massoni, Gheyi, and Borba (2005) checks the validity of refactoring rules over UML class diagrams by translating everything into Alloy to discover inconsistencies in the rules, taking advantage of the instance semantics of Alloy.

3.2.3.2 Type II: Transformation-Dependent and Input-Independent

For this type of verification, classical tool-assisted techniques are generally used: model-checking, static analysis and theorem-proving.

Model-Checking Rensink, À. Schmidt, and Varró (2004) compared two approaches for the model-checking of GBTs. The first approach used the CheckVML Tool to transform a GBT system to a Promela model, further verified using SPIN. The second approach used the Groove Tool to simulate GBT rules and build a state space of graphs for model-checking. The second approach was found more suited for dynamic and symmetric problems. Lúcio, Barroca, and Amaral (2010) implemented a model-checker for the DSLTrans Tool that builds a state space for a transformation where each state is a possible combination of the transformation rules of a given layer, combined with all states of the previous ones. The generated state space is then used to prove if properties hold for all input models of the transformation. Varró and Pataricza (2003) used model checking to prove that dynamic consistency properties were preserved in a model transformation from statecharts to Petri Nets.

Static Analysis Becker et al. (2006) proposed a static analysis technique to check whether a model transformation (formalized as graph rewriting) preserved constraints expressed as (conditional) forbidden patterns in the output model. The study proved that the structural adaptation does not transform a safe system state to an unsafe one by verifying that the backward application of each rule to each forbidden pattern cannot result in a safe state.

Theorem Proving Asztalos, Lengyel, and Levendovszky (2010) proposed deduction rules that can be applied to model transformation rules (formalized as graph rewriting) to prove or disprove a property. The deduction rules were implemented as a verification framework in Visual Modeling and Transformation System and were used to verify a refactoring transformation on business process models. R. F. Paige, Brooke, and Ostroff (2007) compared two approaches for the verification of model conformance checking and multi-view consistency checking (MVCC): with PVS, a popular theorem prover based on set theory; and with Eiffel, an object-oriented language. Nevertheless, performing MVCC checking requires actually executing the generated Eiffel code. Giese, Glesner, et al. (2006) proposed formalizing Model-to-Code transformations using TGGs in Fujaba, further verified within Isabelle/HOL.

3.2.3.3 Type III: Transformation-Dependent and Input-Dependent

Using Traceability Links To prove that a specific transformation preserved certain properties for a specific input model, some studies proved that input-output relationships are maintained for a transformation instance. Narayanan and Karsai (2008b) used GReAT for both structural and semantic relationships between source and target models: they generate crosslinks between source and target models to check structural correspondence between source and target models. Narayanan and Karsai (2008a) check state reachability in a transformation between StateCharts to Extended Hybrid Automata, by checking the existence of a bisimulation with the help of crosslinks between source and corresponding target models.

Using Petri Net Analysis de Lara and Vangheluwe (2010) formalized the operational semantics of a visual TL using graph rewriting. The transformations and the manipulated models were transformed into Petri Nets to benefit from existing FV techniques. The study further proposed extending graph rewriting rules with timing information and transforming them into timed Petri Nets for formal verification.

Using SAT Solvers Anastasakis, Bordbar, and Küster (2007) used Alloy for simulation and assertion checking. Source and target metamodels, as well as transformations, are represented as Alloy models. The Alloy Analyzer then generates possible instances of the source metamodel and the transformation; the Analyzer is then used to check if the corresponding target model satisfies assertions. If no instance is found, it reveals inconsistencies in the transformation specification.

3.2.3.4 Summary

Table 3.2 classifies the literature contributions according to the last dimension, namely the formal verification technique employed for ensuring transformation correctness. This research emphasised two cross-cutting levels: whether a verification technique depends on the *transformation* at hand, or applies to all possible transformation expressible within the transformation framework; and whether it depends on a particular *input model*, or for all possible conforming input models. This classification offers a graduated evaluation of the effort needed for the verification process: the more specific it is (i.e. specific to a transformation *and* an input model), the easier and "lightweight" the technique can be performed. At one extreme, transformation-dependent and input-dependent techniques are very close to testing, but with the fundamental difference that they are still offline.

Another result of this research is the large spectrum of techniques already implemented for verifying model transformations. With the notable exception of Abstract Interpretation, all other techniques are at least implemented by one contribution, the most represented being model-checking. Beyond its relative popularity, model-checking is an attractive technique for two reasons: it is fully automatic, and the state explosion problem can be sometimes circumvented within specific contexts, when the models involved in a transformation are sufficiently small.

3.3 Discussion

Our tridimensional classification captures all variation points influencing the activity of formally verifying model transformations. We discuss in this Section the relations between pairs of dimensions (as depicted in Figure 3.9), and revisit the central role played by the notion of model transformation *intent*.

3.3.1 Property Kind / Fv Technique (PK/FVT)

This relation is the best explored within this paper, and directly relates to the contributions made by the Computer-Aided Verification community: on the one hand, we distinguished two property kinds that follow the dual nature of model transformations (Bézivin et al. 2006) to obtain *language-related* and *transformation-related* properties; on the other hand, we identified three different types of Fv techniques that depend or not on the transformation and the input.

From the literature, we showed that language-related properties, such as termination or determinism, are often proved mathematically: when it is possible to establish such properties for all transformations expressible in a given Tl, the proof is discharged mathematically once and for all; otherwise, sufficient criteria (also mathematically proved) are integrated into Tls to help transformation designers establish those kind of properties for each transformation.

Classical Fv techniques, such as static analysis, model-checking, theorem-proving, are mostly employed for transformation-related properties that are related to the transformation's semantics, and that have to be verified on all possible inputs. We note that abstract interpretation is largely absent from the reviewed contributions. Two explanations can justify this fact: the underlying difficulty of the mathematical underground, and the lack of general-purpose tools.

It is sometimes interesting to prove some properties of interest on a specific input only, for example when the transformation is used on a limited number of input models that need to be deployed within an application (e.g. a transformation expressing the behaviour of a Dsl). In this context, the classical techniques remain applicable but lightweight approaches become also interesting because they are generally easier to deploy.

An interesting trend in the literature is the use of *model syntactic correspondences*. Although the idea already exists for programming languages, it reaches another level of complexity for model manipulations. Capturing

	Transformation Independent	**Transformation Dependent**
Input Independent	Barroca et al. (2010) Padberg, Gajewsky, and Ermel (1997) Massoni, Gheyi, and Borba (2005) Stenzel, Moebius, and Reif (2011) H. Ehrig, K. Ehrig, et al. (2005) Varró, Varró-Gyapai, et al. (2006) Küster (2006)	Rensink, À. Schmidt, and Varró (2004) Lúcio, Barroca, and Amaral (2010) Becker et al. (2006) Asztalos, Lengyel, and Levendovszky (2010) R. F. Paige, Brooke, and Ostroff (2007) Giese, Glesner, et al. (2006) Varró and Pataricza (2003)
Input Dependent		Narayanan and Karsai (2008a) de Lara and Vangheluwe (2010) Narayanan and Karsai (2008b) Anastasakis, Bordbar, and Küster (2007)

Table 3.2 – Classification of verification techniques.

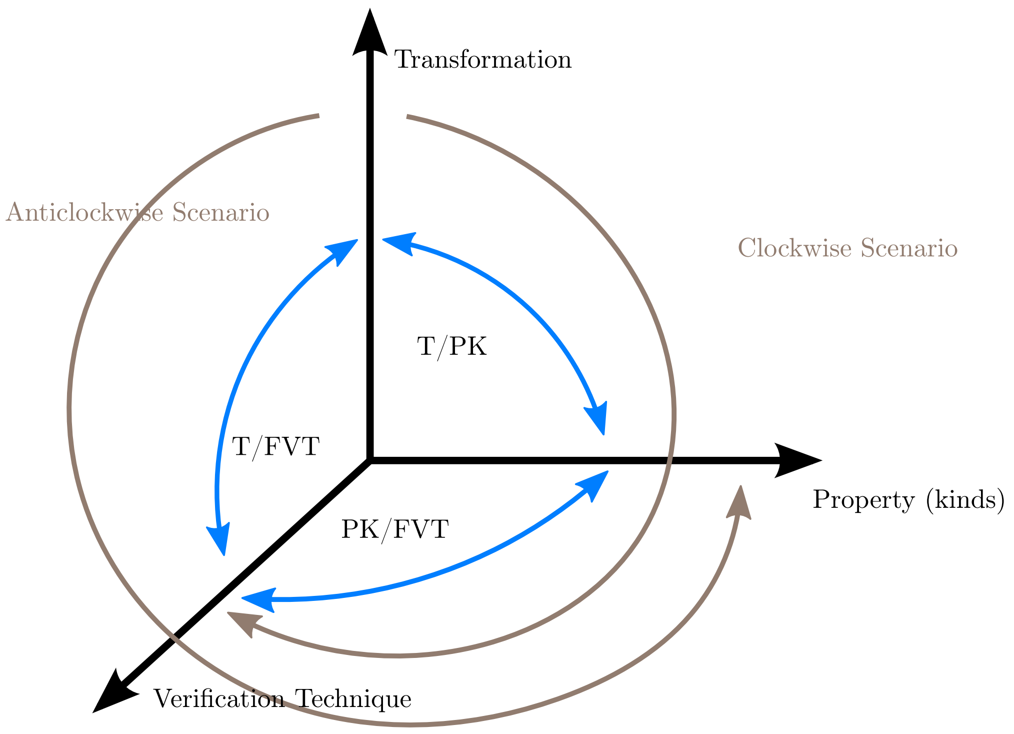


Figure 3.9 – Closing the loop: the relation between each pair of transformation dimension; and the two possible scenarios for model transformation verification.

the similarity of the input and output models through patterns or contracts expressed on each side avoids diving into the complexity of the semantic layer. This trend has interesting results for structural models, but can only provide a quick check for more complex models that integrate behaviour.

3.3.2 Transformation / Fv Technique (T/FVT)

This relation remains largely unexplored in our work as well as in the literature. It seems natural that the underlying paradigm of TLs would influence the spectrum of Fv techniques that are usable for proving some property kinds. The respective research communities (of programming languages on the one hand, and of verification on the other hand) could provide interesting knowledge about the core principles governing this relation, and to which extent it is possible to adapt this knowledge to the manipulation of models.

From the literature, we noticed however an interesting trend. Instead of developing specific techniques for model transformations, some contributions took the opposite approach: they express the full semantics of a TL within a general-purpose programming language already equipped with analysis capabilities (such semantic mappings exist for example for GBT in simple/double pushout style in Maude (Rivera, Guerra, et al. 2009). This approach is interesting because it is often perceived as easier than developing an analysis engine from scratch. However, it limits the spectrum of verifiable property kinds to those already handled by the selected programming language.

3.3.3 Transformation / Property Kind (T/PK)

We showed on a simple example that the current classifications for model transformation are not sufficient to derive the properties one needs to prove to ensure transformation correctness. What really matters is the *intent* of a transformation: by capturing the *purpose* of a transformation instead of the *form* of its expression, one can define precisely what is the appropriate notion of *correctness* attached to this transformation. We extracted several kinds of properties of interest from the contributions found in the literature; however, a more systematic study of both dimensions and how they relate to each others will enable a better engineering of model transformation verification.

3.3.4 Transformation Intent as the glue between dimensions

How can a transformation designer be guided through the process of formally verifying model transformations, especially when those transformations are used in sensitive applications like safety-critical and embedded systems? We have already discussed the drawbacks of first selecting a transformation engine natively equipped with predefined verification capabilities, which corresponds to the anticlockwise scenario in Figure 3.9. In our opinion, another possibility is more desirable. Instead of limiting *a priori* the set of verification techniques a designer can employ, it can be more interesting to select the appropriate technique(s) corresponding to the needs of each transformations. As already noticed e.g. by Rahim and Whittle (2013), the notion of model transformation *correctness* is tighly related to the *kind* of transformation one is attempting to verify. We believe that the concept of the engineering *intent* of a model transformation can act as a "glue" between our three dimensions: by identifying the intent of a transformation, it becomes possible to precisely circumscribe the properties that matter for proving that transformation's correctness, and then to select the most appropriate verification technique(s) for discharging this proof.

3.4 Conclusion

This Chapter explored the second central notion for this Thesis, namely Formal Verification, in two stages: first, it reviewed the main characteristics and the most popular techniques of Formal Verification; and then surveyed the literature for its application in the domain of Model Transformation. For this purpose, we introduced a powerful tridimensional approach based on the components involved in the process of transformation verification: the transformation, the properties and the verification techniques. This approach allowed us to capture an interesting snapshot of the current engineering practice in this domain, and proved to be popular (for example, Calegari and Szasz (2013) almost copied our approach).

Starting from a popular model transformation task, namely providing semantics to a DSL, we showed that existing classifications of model transformations lack to classify their *purpose*, or *intent*. In the following Chapter, we explore further this novel notion and its implication for the purpose of model transformation verification, as a possible answer to the clockwise scenario suggested in Figure 3.9.

4 Characteristic Properties of Model Transformations Intents

In MDE, *models* or software abstractions comprise the basic building blocks in the software development process and such models are manipulated by model transformations. Thus, model transformations are considered the *heart and soul* of MDE (Sendall and Kozaczynski 2003) and can be used for a variety of purposes such as the generation or synchronization of models on different levels of abstraction, the creation of different views on a system, and the automation of model evolution tasks (Czarnecki and Helsen 2006).

Although several aspects of model transformations have been thoroughly investigated in the literature (e.g., model transformation languages and applications of model transformations), minimal research has been conducted on requirements and specifications for model transformations in general, and on the different *intents* or purposes that model transformations can typically serve in MDE and how they can be leveraged for development and validation activities.

We showed on the previous Chapter that classifying model transformations by their form is not enough for extracting the properties of interest needed to prove the validity of transformations. In this Chapter, we introduce the new notion of model transformation *intent* (Amrani et al. 2012a) that serves as a "semantic" classification of model transformation, i.e. targeting the *purpose*, or *goal*, of a model transformation, in contrast to the existing "syntactic" classifications (Czarnecki and Helsen 2006; Mens and Van Gorp 2006) more interested in the *form* of transformations.

This Chapter starts by motivating the introduction of our *Description Framework* as a methodological approach for studying and engineering model transformations. Section 4.1 provides an overview of the Description Framework as well as a motivation from a research and engineering perspectives. Section 4.2 goes a step ahead by presenting two conceptual metamodels, making things more operational, and explaining through several scenarios how the Framework can be exploited. The rest of the Chapter then details all components of the Description Framework articulated around a real-life case study.

4.1 Overview & Motivation

The Description Framework allows the construction of a model transformation intent catalogue through the identification of properties that an intent must or may possess, and any conditions that support or conflict with an intent. For instance, a *Translation* model transformation intent can describe a model transformation whose purpose is to prepare a model M_1 for some kind of analysis. Thus, for a model transformation to be considered a valid realisation of the translation intent for analysis, it should produce an output model M_2 that when analysed, yields analysis results that "carry over" to M_1.

As illustrated in Fig. 4.1, intents are used to group transformations with the same goal and to associate with them so-called *characteristic intent properties*, such as termination, type correctness, traceability, or the preservation of structural or semantic aspects. A characteristic intent property can be thought of as a template that can be concretised into a *transformation property*, i.e., a concrete property pertaining to a specific transformation. The resulting link between transformations and transformation properties then facilitates validation

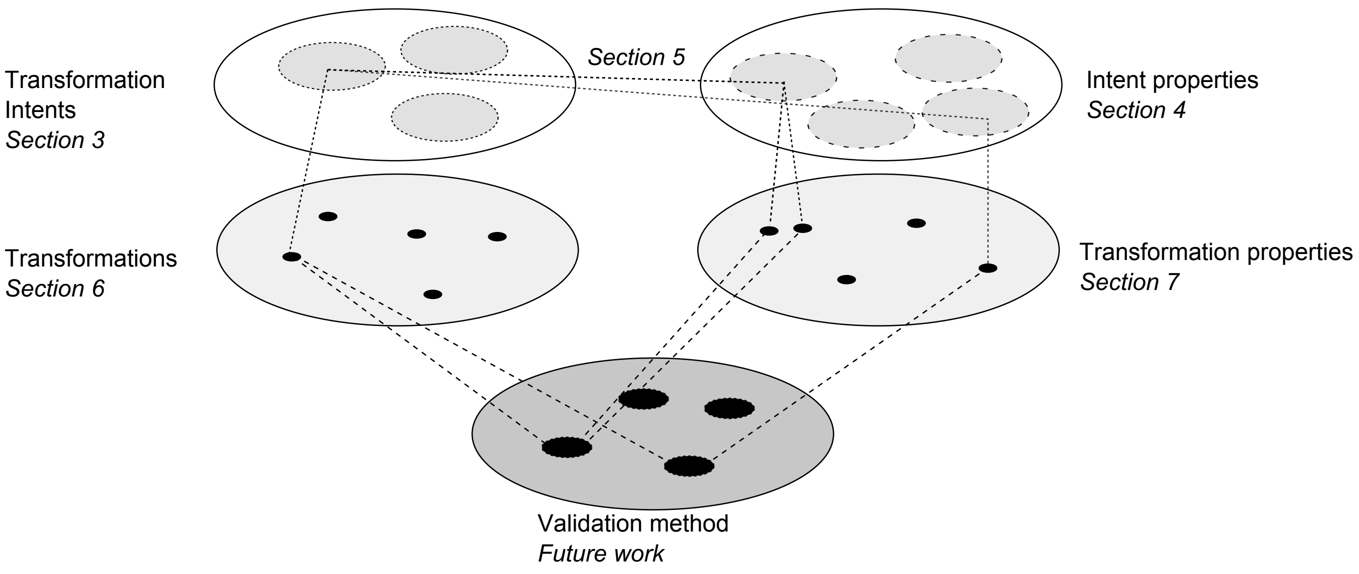


Figure 4.1 – Intents as a classification mechanism for model transformations

of transformations via appropriate validation methods.

The use of the framework is illustrated by presenting an initial catalogue of 21 common model transformation intents and discussing five of them (Query, Refinement, Translation, Analysis and Simulation) in more detail. Moreover, a case study involving the use of model transformations for the development of the control software for a power window in the automotive industry is described and for some of these transformations their intents and transformation properties are identified.

This work on model transformation intents should be useful to MDE practitioners and researchers. For instance, it would help engineers identify the model transformation intent that best matches a particular MDE development goal and facilitate the subsequent model transformation development or reuse by explicating the properties that a model transformation has to satisfy. Moreover, the notion of model transformation intent would also provide useful input for researchers interested in the specification and analysis of model transformations by clarifying how to best describe what a transformation is doing and which kinds of model transformation analyses might be most useful. Finally, the notion of model transformation intent can be used to classify model transformations into different *domains* that can be leveraged for the development of domain-specific model transformation languages and tools dedicated to express transformations of specific intents due to the language features or the kinds of analyses that they support.

4.2 A Description Framework for Model Transformation Intents

The ideas informally depicted in Figure 4.1 are now more formally described by means of the two metamodels of Figures 4.2 and 4.3. After describing them, we explain how they can concretely be used by transformation designers or researchers.

4.2.1 A Metamodel for Intents and their Properties

In Figure 4.2 a ModelTransformationIntent is described in a manner similar to object-oriented design patterns (Gamma et al. 1995). An intent has a name, and is more precisely described using description and useContext. The description informally conveys the general idea behind the intent whereas the useContext presents precise scenarios where the intent is used. One or several examples refer to sample transformations, possibly from the literature, having this intent. A set of preconditions describes any necessary conditions that

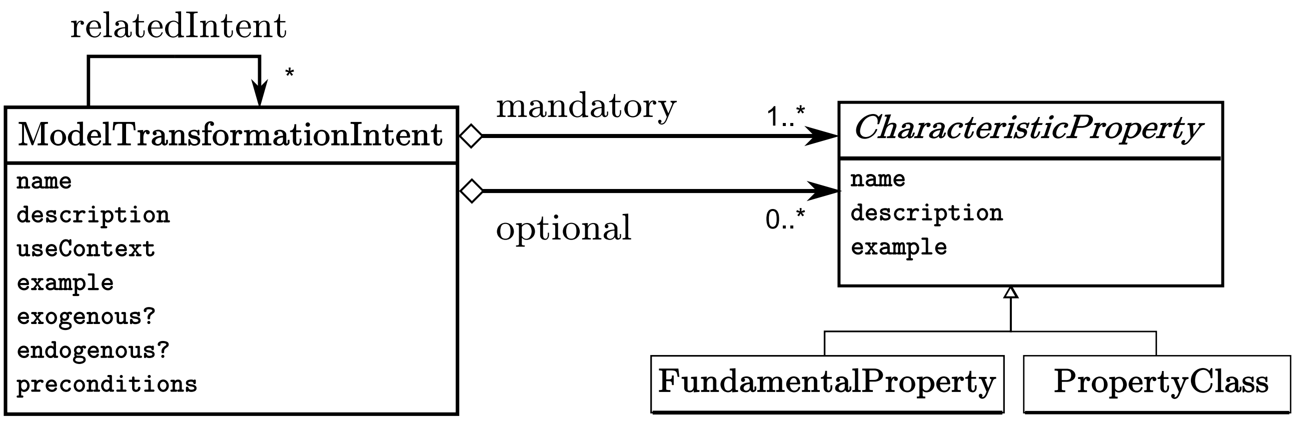


Figure 4.2 – Metamodel for describing model transformation intents.

need to be satisfied for transformations to possess this intent. Boolean attributes exogenous? and endogenous? indicate whether transformations with this intent can have different or the same metamodel. An intent can have several relatedIntents with which it shares similarities regarding the nature and purpose of their model manipulation; related intents are expected to have similar CharacteristicPropertys.

A CharacteristicProperty (also called *intent property*) of an intent is a property common to all transformations with that intent. Characteristic properties can be thought of as templates with "holes" for either the specifics of a transformation (e.g., its specification or just aspects of it, e.g., the target metamodel) or of the property to be expressed (e.g., a postcondition the output model has to satisfy). Characteristic properties can thus refer to aspects of the execution of the transformation, or to the result produced. The size and number of holes makes some intent properties more abstract than others. Section 4.4 presents several characteristic properties including "termination", "type correctness", and "determinism" which are concrete; more abstract intent properties include the "Structural Relation Property" which allows the expression of conditions over pairs of input and output models; intent property "Semantic Relational" additionally considers their semantics; properties requiring the preservation of aspects of structure or semantics arise as special cases of these two.

The mapping between ModelTransformationIntent and CharacteristicProperty is split into two different parts: mandatory and optional properties. The mandatory property set describes *necessary* properties for a transformation to have a particular intent. Note, however, that this set is *not sufficient*, i.e., it is very common that related intents share their mandatory properties. In such cases, the intents' remaining attributes have to be consulted for disambiguation. The optional property set collects properties that transformations with a specific intent may, but do not need to, have.

4.2.2 A Metamodel for Model Transformation Validation Methods

If the transformation is part of the development of a safety-critical application, validation[1] or even formal verification may be desired. Partial classifications of formal verification techniques for model transformations have already been proposed in (Amrani et al. 2012b; Calegari and Szasz 2013) where is highlighted the impact of two factors on the suitability of a given verification technique (cf. Section 3.2): the *model transformation language paradigm*, i.e. if the model transformation language is, e.g., declarative, meta-programmed, or hybrid (Czarnecki and Helsen 2006), and the *model transformation form*, i.e., how the transformation is syntactically specified (Czarnecki and Helsen 2006).

The process of filling the holes of an intent property is called *concretisation* and yields a TransformationProperty, i.e., a fully fleshed out property pertaining to a specific transformation which can be used for transfor-

[1]We use the term *validation* to refer to all formal, semi-formal, and informal activities aimed at collecting evidence for the correctness of a model transformation with, e.g., testing and formal verification as prominent special cases.

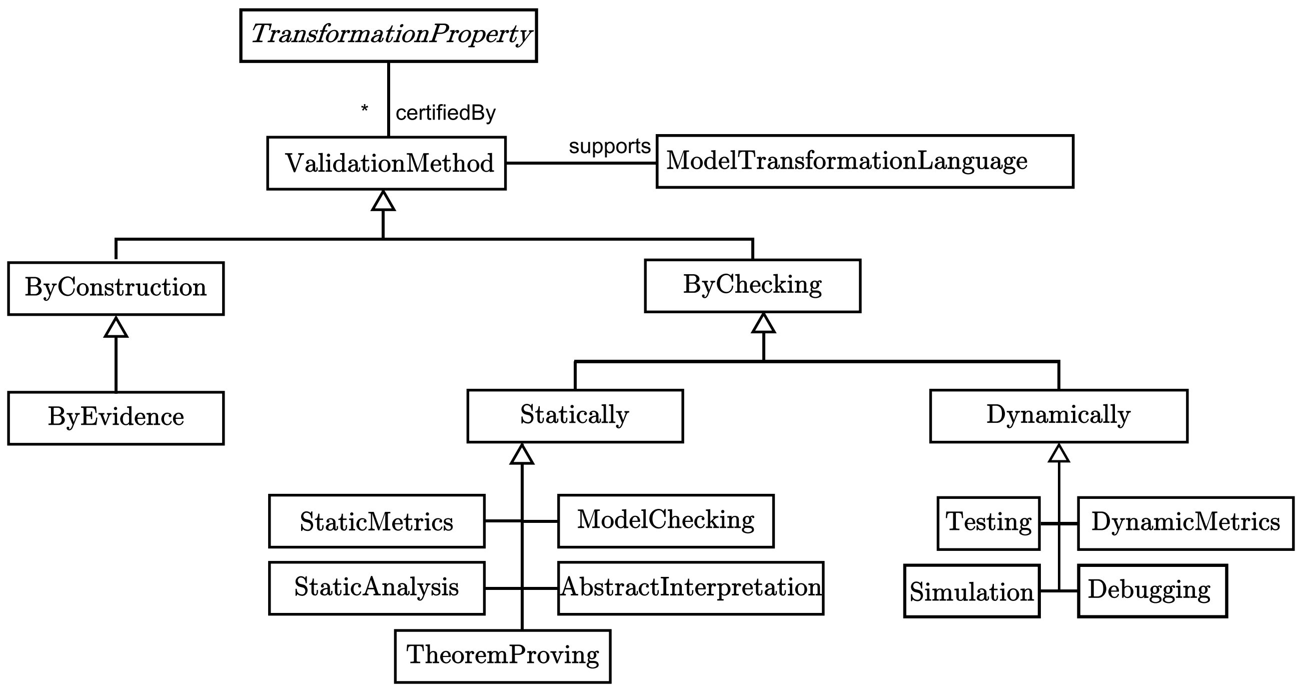


Figure 4.3 – Methods for validating model transformations.

mation validation. In comparison to Amrani et al. (2012b) and Calegari and Szasz (2013), Figure 4.3 collects and organises ValidationMethods (extracted from Adrion, Branstad, and Cherniavsky 1982; E. M. Clarke and Wing 1996; D'Silva, Kroening, and Weissenbacher 2008) for validating a transformation with respect to a transformation property. We distinguish between two validation categories: ByConstruction and ByChecking. ByConstruction means that the property is implied by the way the transformation language is constructed and operates. Techniques that allow transformation-independent and input-independent validation of transformations (i.e., properties that are shown to hold for all transformations of the language and for all input models) are often ByConstruction: for instance, using a mathematical proof one might be able to show termination or determinism for a model transformation expressed as a graph rewriting system for all transformations and inputs (cf. Section 3.2.3.1 for details). Other formal properties are either Statically or Dynamically validated with formal techniques. *Dynamic* techniques require executing the transformation being validated (e.g., Testing or DynamicMetrics) whereas *static* techniques include abstraction-based techniques such as AbstractInterpretation, TheoremProving, ModelChecking or any StaticAnalysis with a specific scope (e.g., identifying unfireable rules). For many of these categories, concrete examples of approaches from the research literature can be found in Section 3.2.

4.2.3 Usage Scenarios

We presented a mapping between transformation intents and characteristic properties, intended to be concretised to enable transformation validation through various techniques. This proposal can be helpful for practitioners and researchers for supporting the following activities. This Section presents several scenarios where the Description Framework is useful: for *identifying* the intent of model transformation and for *validating* model transformations against their properties of interest (both described through an Activity Diagram Object Management Group 2011a) and more generally, for *favouring* model transformation research.

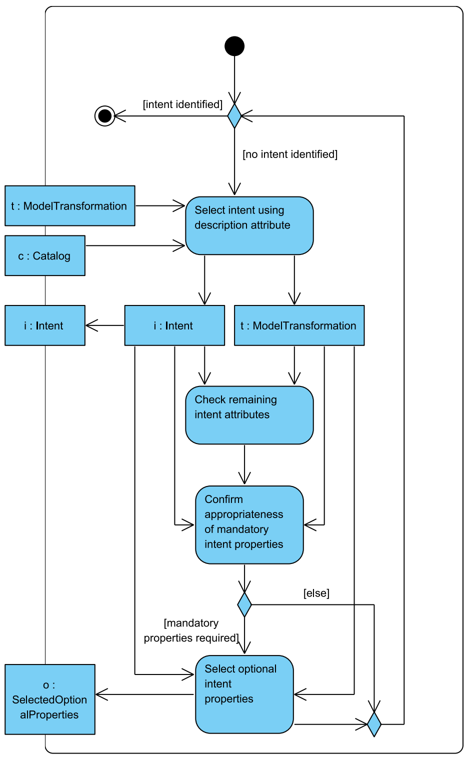

[intent identified]
[no intent identified]
t : ModelTransformation
c : Catalog
Select intent using description attribute
i : Intent
i : Intent
t : ModelTransformation
Check remaining intent attributes
Confirm appropriateness of mandatory intent properties
[else]
[mandatory properties required]
o : SelectedOptionalProperties
Select optional intent properties

Figure 4.4 – Identifying the intent of a model transformation

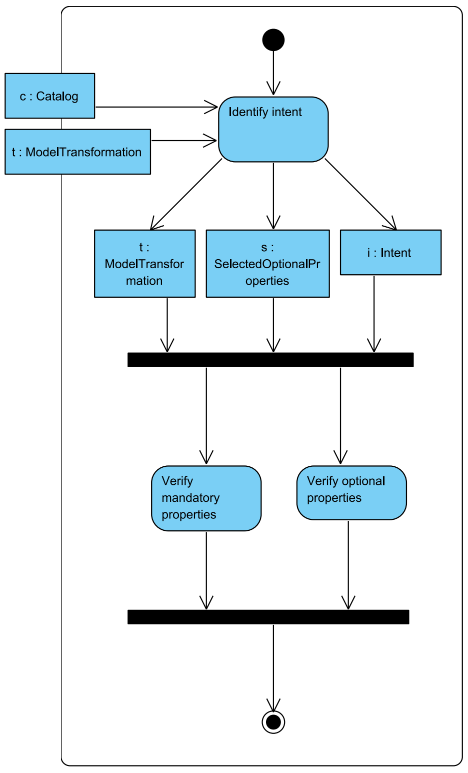

c : Catalog
t : ModelTransformation
Identify intent
t : ModelTransformation
s : SelectedOptionalProperties
i : Intent
Verify mandatory properties
Verify optional properties

Figure 4.5 – Validating a model transformation with a specific intent

4.2.3.1 Intent Identification

Given an existing transformation, our intent catalogue can be used to determine the intent of that transformation together with any relevant optional intent properties as depicted in Figure 4.4. Should the transformation not match any intent in the catalogue sufficiently well, our framework could be used to describe the new intent and add it to the catalogue. Knowing the transformation's intent may facilitate the documentation, maintenance, validation, or reuse of the transformation. If the transformation has not been implemented yet, intent identification may still be possible using, e.g., requirements documents or interviews with MDE engineers. In this case, knowing which intent the transformation possesses may facilitate its implementation.

4.2.3.2 Model Transformation Validation

For validating a given transformation with respect to a specific intent, the mandatory intent properties and, to the appropriate extent, the optional intent properties, need to be concretised into transformation properties pertaining to the given transformation. Validation succeeds if the transformation satisfies all transformation properties. This process is summarized in Figure 4.5.

4.2.3.3 Model Transformation Research

Our work is relevant to researchers interested in the specification and analysis of model transformations, since it describes and formalises properties that transformations may have to possess. Allowing for these, and perhaps other properties to be expressed in a uniform, elegant specification language for model transformations would be of interest, as would be the development of effective analysis and validation techniques and tools for model transformations.

Some intents may occur so frequently and require so much development effort, that the development of an "intent-specific" (e.g., domain-specific) transformation language may be helpful. The new language may be a subset of an existing one obtained by removing certain constructs (e.g., constructs that introduce non-termination), or a completely new language employing paradigms and features that optimally support the efficient construction of transformations with a specific intent. Should these transformations be part of the development of safety-critical software, designing the transformation language in such a way that the proof of transformation properties is facilitated (e.g., a transformation language without possibly non-terminating constructs will only allow the construction of terminating transformations) could further increase productivity. Consequently, our work may also stimulate more research into the design, implementation and analysis of domain-specific model transformation languages.

4.2.4 Outline

Section 4.3 presents a non-exhaustive catalogue of 21 common transformation intents. The description of each intent is rather short, using only a small part of our framework of Section 4.2. Section 4.4 presents high-level formalisations of some key characteristic intent properties. The list of properties is also not meant to be exhaustive. Section 4.5 uses the full framework from Section 4.2 to provide detailed descriptions of five intents: *Query*, *Refinement*, *Translation*, *Analysis*, and *Simulation*. Section 4.6 describes the Power Window Case Study (PWCS) which shows how MDE techniques in general, and model transformations in particular, can be used for the development of software for a power window. The case study contains a transformation chain of over 30 transformations. Section 4.7 applies our work to the case study. After a detailed description of two transformations extracted from the case study, their intents are identified and some of their intent properties are concretised into transformation properties for validation purposes (the validation itself is left for future work, though). Section 4.7 ends by identifying the intent, together with their properties, for all transformations in the case study. Section 4.8 and 4.9 discusses topics around the notion of intent, and the related work.

4.3 The Intents Catalog

Several classifications for model transformations exist in the literature. Such classifications are based on the transformation features and form (Czarnecki and Helsen 2006), or syntactic aspects (Mens and Van Gorp 2006). From a formal verification point of view, what really matters is the intent behind a transformation (cf. Section 3.2): the intent conveys the transformation's actual meaning, which influences the properties of interest that need to be verified. This section proposes an *Intents Catalogue*: a description of recurring model transformation intents and illustrative examples from the literature. With respect to the metamodel in Figure 4.2, this Catalogue informally indicates several instances of the ModelTransformationIntent class for which the following information is detailed: name, description and example.

Our Intents Catalogue is not an exhaustive list of all model transformation intents, but it encompasses existing lists (e.g. Czarnecki and Helsen 2006; Iacob, Steen, and Heerink 2008; Mens and Van Gorp 2006; Syriani 2011; Tisi et al. 2009, which are discussed in Section 4.9) and has already been presented in several workshops with various audiences to validate it.

Model Editing The simplest operations on a model are *adding* an element to the model, *removing* an element from the model, *updating* an element's properties, *navigating* through the elements, and *accessing* the properties of an element. These primitive operations are also known as the *CRUD* operations as first introduced by Kilov (1990). These simple operations are considered as a model transformation when the system is completely and explicitly modeled, such as in AToMPM (Syriani 2011).

Restrictive Query A query transformation requests some information about a model in the form of a proper sub-model or a *view*. This operation takes a model M as an input and outputs a **view** of M. For example, the query “retrieve all cycles in a Causal Block Diagram model” outputs a view of the causal block diagram model represented as a cyclic graph composed of strongly connected components. We consider any subsequent aggregation or restructuring of the resultant sub-model or view as an abstraction. EMF INC-Query (Bergmann et al. 2011) is a model transformation language that is used specifically for querying EMF models.

Refinement A refinement transformation produces a lower level specification, e.g. a platform-specific model, from a higher level specification, e.g. a platform-independent model (A. G. Kleppe, Warmer, and Bast 2003), i.e., refinement adds precision to models. As defined by Giese, Levendovszky, and Vangheluwe (2007), a model m_1 refines another model m_2 if m_1 can answer all questions that m_2 can answer. Typically, m_1 contains at least the same information as m_2. For example, a non-deterministic finite state automaton (NFA) can be refined into a deterministic finite state automaton (DFA). Denil et al. (2012) defined a set of refinement transformations that iteratively add platform knowledge to a deployment model.

Abstraction Abstraction is the inverse of refinement: if m_1 refines m_2 then m_2 is an abstraction of m_1. Typically, m_2 will hide some information while revealing other information. For example, an NFA is an abstraction of a DFA. Mannadiar and Vangheluwe (2010) used a transformation to extract user-interface behavior from a Statecharts model into a PhoneApps model.

Synthesis A synthesis transformation produces a well-defined language format from an input model, such as in *serialization*. Synthesis is also referred to as *Model-to-code generation* (Stahl, Voelter, and Czarnecki 2006) when the transformation produces source code in a target programming language. For example, Java code can be synthesized from a UML class diagram model.

Note that the synthesis intent can be considered as a special case of the refinement intent if the output of the transformation is an executable artifact. Further, the refinement intent can be viewed as a means to achieve the synthesis intent as demonstrated by Mannadiar and Vangheluwe (2010) and Tri and Tho (2012).

Reverse engineering Reverse engineering is the inverse of synthesis: it extracts higher level specifications from lower-level ones. For example, a UML class diagram model can be generated from Java code using Fujaba (T. Fischer et al. 2000). Note that reverse engineering can be considered as a subset of abstraction where the input model is code.

Approximation As defined by Giese, Levendovszky, and Vangheluwe (2007), an approximation transformation is a refinement transformation with respect to negated properties. That is, m_1 approximates m_2 if m_1 negates the answer to all questions that m_2 negates. For example, a Fast Fourier Transform is an approximation of a Fourier Transform which is computationally very expensive.

Translation A translation transformation translates the meaning of a model in a source language in terms of the concepts of another target language. The resulting model can then be used to achieve several tasks that are difficult, if not impossible, to perform on the originals. For example, Syriani and Ergin (2012) transformed a UML activity diagram into a Petri Net model in order to detect deadlocks and starvation, i.e., analysis is *delegated* to the Petri Net workspace. Translation is also common for capturing the semantics of new DSLs. In this case, the transformation specifies the *semantic mapping* (Harel and Rumpe 2000) into a semantic domain with well-known semantics. For example, Causal Block Diagram's semantics are expressed as Ordinary Differential Equations.

Analysis A model transformation can be used to implement analysis algorithms of varying complexities, starting from detecting dead code or unapplicable rules to model-checking temporal formulae over appropriate structures described by models. For example, Lúcio and Vangheluwe (2013a) implemented a symbolic model-checker for the DSLTrans transformation language using model transformations.

Simulation A simulation transformation defines the *operational semantics* of a modeling language by defining a model transformation that updates the modeled system's states. The output model of the transformation is then an "updated version" of the input model (i.e., the transformation is in-place). Simulation updates the abstract syntax of the model, which may trigger modifications in the concrete syntax. T. Kühne et al. (2010) provide one example where a model transformation was used to simulate a Petri Net model and produced a trace of the transitions firing.

Animation Animation is the visualization of a simulation. It projects the behavior of a model on a specific animation view. In contrast with a simulation transformation, an animation transformation operates on the concrete syntax (or the abstract syntax of the concrete syntax) of a model. For example, Ermel and H. Ehrig (2008) used a model transformation to define the mapping from simulation steps to animation steps of a radio clock.

Normalization A normalization transformation aims to decrease the syntactic complexity of models by translating complex language constructs of an input model into more primitive constructs, resulting in a canonical form of the input model. For example, Agrawal et al. (2006) normalized a Statechart model into its flattened form, replacing OR- and AND-states by the appropriate states and transitions. *Parsing* the concrete syntax of a modeling language back to its abstract syntax is also considered normalization that can be implemented by a model transformation involving the meta-model of the concrete syntax and the meta-meta-model of the language.

Rendering A rendering transformation assigns one (or more) concrete representation(s) to each abstract syntax element or group of elements in an input model, as long as the meta-model of the concrete syntax is defined explicitly. For example, Guerra and de Lara (2007) used event-driven grammars to relate the abstract and concrete syntaxes of visual languages.

Model Generation Model generation is a transformation that automatically produces possible (correct) instances of a metamodel, such as in (Winkelmann et al. 2008). The meta-model of a language can be defined using a grammar, e.g., the Extended Backus-Naur Form (EBNF), or a graph grammar (Viehstaedt and Minas 1995). Such model transformations are very useful for testing model transformations since it facilitates the automatic generation of input test models to verify the correctness of a transformation (Dalal et al. 1999).

Model Finding Adapted from Torlak and Jackson (2007), model finding is a transformation that searches for models that satisfy given constraints. In that case, several models are generated according to a set of rules and evaluated to check whether the generated models satisfy some constraints. If not, a backtracking mechanism reverses some of the applied rules to find another model. A typical use of this intent is in *design-space exploration* (e.g. Schatz, Holzl, and Lundkvist 2010) which supports decision-making when several solutions exist.

Migration A migration transformation transforms software models written in one language (or framework) into software models conforming to another language (or framework), while keeping the models at the same abstraction level Mc Brien and Poulovassi 1999. For example, transforming Enterprise Java Beans 2.0 (EJB2) class diagrams so that the resulting models conform to EJB3 can be achieved by a migration transformation as in Asztalos, Syriani, et al. 2011. The process of migrating each model individually so that they conform to the evolved metamodel can be automated through model transformations as presented in Cicchetti et al. 2008.

Optimization An optimization transformation aims at improving operational qualities of models, e.g., scalability and efficiency. For example, replacing n-ary associations with binary associations in a UML class diagram can optimize the code generated from the class diagram Gessenharter 2008.

Model Refactoring Model refactoring is a transformation that restructures a model to improve certain quality characteristics without changing the model's observable behavior (Fowler 1999; Griswold 1991). Zhang, Lin, and Gray (2005) proposed a generic model transformation engine that can be used to specify refactorings for domain-specific models.

Model Composition Model composition integrates models produced in isolation into a compound model, where each isolated model represents a concern that may overlap with any of the other isolated models. A particular instance of composition is *model merging*. In this case, the composition creates a new model such that every element from the union of both models is present exactly once in the merged model. Engel, R. Paige, and Kolovos (2006) proposed a transformation language that allows one to compute the merged model from two models conforming to the same meta-model.

Model Matching A model matching transformation creates correspondence links between corresponding entities. This is also known as *model weaving*. Fabro and Valduriez (2009) defined a generic meta-model to capture correspondences between models.

Model Synchronization Model synchronization integrates models that have evolved in isolation and that are subject to global consistency constraints by propagating changes to the integrated models. Such transformations are typically used when multiple views of a common repository model are accessed or modified as in (Guerra and de Lara 2006).

4.4 Characteristic Properties

This Section gathers formal definitions for the characteristics intent properties, built on top of basic definitions for the core components of model transformation, as depicted in Figure 2.5 (namely, transformation *specification* and *execution*), independent of the transformation's underlying paradigm. Directly inspired from the relevant literature in Computer Aided Verification, these definitions have been limited for covering the five intents of this Chapter. Nevertheless, they form an extensible basis as new intents will be described within our Description Framework.

Recall from Section 2.2 the basic definitions for (meta-models) and transformations. The sets of models and metamodels are denoted respectively by $\mathbb{M}$ and $\mathcal{M}$; given $\mathsf{M} \in \mathbb{M}$ and $\mathsf{MM} \in \mathcal{M}$, the fact that M conforms to MM is noted $\mathsf{M} \rhd \mathsf{MM}$, and the set of models conforming to MM is $\mathcal{L}(\mathsf{MM})$ (cf. Definition 2.1. A transformation specification is a triple $t = (\mathsf{MM_s}, \mathsf{MM_t}, \mathsf{spec})$, where $\mathsf{MM_s}, \mathsf{MM_t} \in \mathcal{M}$ are the source and target metamodel respectively, and $\mathsf{spec} \in \mathcal{L}$ the well-formed transformation specification written in a transformation language $\mathcal{L}$. The execution of t corresponds to a transition system $\mathsf{TS_t} = (\mathsf{S}, \mathsf{I}, \longrightarrow)$, where S is the set of execution states, and I the set of initial states (cf. Definitions 2.3 and 2.4). In this Section, we will abuse the notation and amalgamate S with $\mathbb{M}$, and note $\longrightarrow^*$ the transitive closure for $\longrightarrow$: this way, we say that M is *rewritten*, or *reduce*, to M' if $\mathsf{M}' \longrightarrow^* \mathsf{M}'$.

From the perspective of formal language theory, what a model designer defines with a metamodel is a language's *abstract syntax*, i.e. the designer captures the relevant concepts and their relationships in a way that enables their internal representation for further computations. To allow their manipulation by modelers, a metamodel must be accompanied with one or several *concrete syntaxes* that define their concrete representation, be it graphical or textual (or both). As a last ingredient, the *semantics* is necessary to perform manipulations of models according to the meaning attached to the modeled concepts.

Definition 4.1 (Model Semantics (borrowed from Harel and Rumpe 2000))**.** *Let* $\mathsf{MM} \in \mathcal{M}$ *be a metamodel. The semantics of* MM*, noted* $[\![\mathsf{MM}]\!]$*, is a pair* $[\![\mathsf{MM}]\!] = (\mathbb{D}_{\mathsf{MM}}, \mu_{\mathsf{MM}})$*, where* $\mathbb{D}_{\mathsf{MM}}$ *is called the* semantic domain *and* μ_{MM} *the* semantic mapping *defined as follows:*

$$\begin{array}{rcl} \mu_{\mathsf{MM}} : \mathcal{L}(\mathsf{MM}) & \longrightarrow & \mathbb{D}_{\mathsf{MM}} \\ \mathsf{M} & \mapsto & \mu_{\mathsf{MM}}(\mathsf{M}) \end{array}$$

The precise definition of $\mathbb{D}_{\mathsf{MM}}$ and μ_{MM} (noted without subscripts when clear from context) highly depends on the nature of the models in $\mathcal{M}(\mathsf{MM})$ and what the semantics will be used for. If MM does not have any associated behaviour, the semantic domain usually consists only of data structures. Otherwise, as it is generally the case for Domain-Specific Models, MM has a behaviour, and the semantic domain needs to appropriately capture it. Finally, if MM is a full transformation language (making the associated transformation a higher-order transformation), the semantic domain usually corresponds to a fully-fledged mathematical framework (such as category theory) whose precise definition is left implicit.

Note also that the semantics *style* depends on the machinery associated to $\mathbb{D}_{\mathsf{MM}}$: it can be denotational if $\mathbb{D}_{\mathsf{MM}}$ comes with a functional framework, operational if it is equipped with rewriting capabilities, axiomatic if it defines a Floyd-Hoare logic, or even translational if it represents a target computer language the semantics is translated into.

The aim of this section is to provide a minimal mathematical framework for the intent properties used in our description framework (Section 4.2). The next Sections detail the second important class of the metamodel of Figure 4.2, namely CharacteristicProperty. In our description framework, each transformation intent has corresponding mandatory (optional) properties that all transformations with this intent must (may) satisfy. Some intent properties can directly be instantiated for a given transformation, other properties are still quite abstract and will need to be concretised for the given transformation before they can be checked. Section 4.7 demonstrates for two example transformations how it is possible to find out the appropriateness of mandatory properties for a given transformation and also how to concretise abstract intent properties to concrete transformation properties that can be validated.

The description of characteristic intent properties assumes a transformation specification $t = (\mathsf{MM_s}, \mathsf{MM_t}, \mathsf{spec})$ with its associated execution $\mathsf{TS_t} = (\mathsf{S}, \mathsf{I}, \longrightarrow)$. Each property is given an abbreviation (in square brackets following the name) that is used to refer later to the property.

4.4.1 Fundamental Property

A *fundamental property* is one of the following singleton properties: *termination*, *determinism* and *type correctness*. They are called *singleton* because they directly apply to a transformation without needing further concretisation. Type Correctness is specific to model transformations whereas Termination and Determinism are common for any computational system. Further details about the verification of such properties have been detailed in Section 3.2.

Definition 4.2 (Termination [T]). $\mathsf{TS_t}$ *is* terminating *if there exists no infinite chain* $\mathsf{M_0} \longrightarrow \mathsf{M_1} \longrightarrow \ldots \longrightarrow \mathsf{M_n} \longrightarrow \ldots$ *starting from an input model* $\mathsf{M_0}$*. We say that* $\mathsf{M_n}$ *is an* output model *for* $\mathsf{M_0}$ *if there exists a finite chain* $\mathsf{M_0} \longrightarrow \mathsf{M_1} \longrightarrow \ldots \longrightarrow \mathsf{M_n}$ *such that no further transition from* $\mathsf{M_n}$ *exists.*

Consequently, a terminating transformation execution ensures the existence of a final output model for any legal input model. If from $\mathsf{M} \in \mathsf{I}$, we reach $\mathsf{M_n}$ without any further possible reduction, we say that $\mathsf{M_n}$ is *canonical* and note $\mathsf{M} \Rightarrow \mathsf{M_n}$.

Definition 4.3 (Determinism [D]). $\mathsf{TS_t}$ *is* deterministic *(or* confluent*) if for all model* M *that can be reduced to* $\mathsf{M_1}$ *and* $\mathsf{M_2}$ *(i.e.* $\mathsf{M_1} \overset{*}{\longleftarrow} \mathsf{M} \overset{*}{\longrightarrow} \mathsf{M_2}$*), there exists another model* $\mathsf{M'}$ *into which both* $\mathsf{M_1}$ *and* $\mathsf{M_2}$ *reduce, i.e.* $\mathsf{M_1} \overset{*}{\longrightarrow} \mathsf{M'} \overset{*}{\longleftarrow} \mathsf{M_2}$*.*

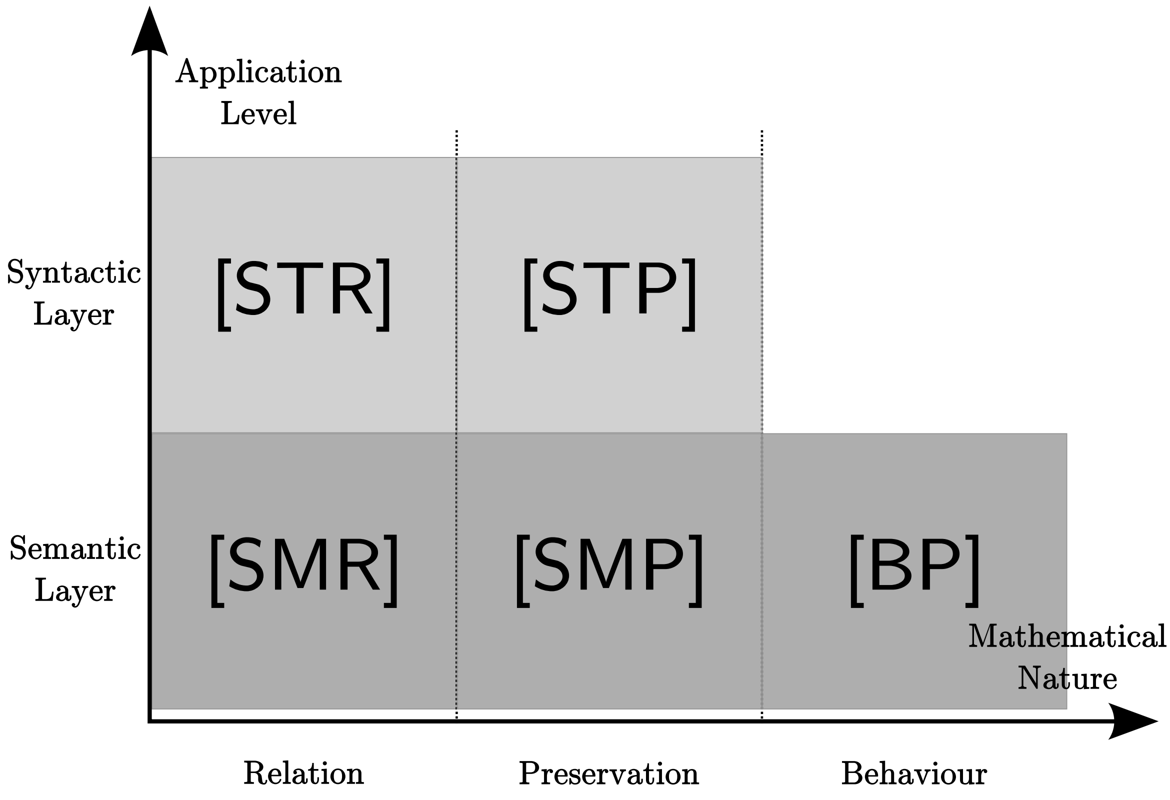


Figure 4.6 – Property Classes arranged according to their underlying mathematical *nature* and their application *level* (either syntactic or semantic)

Determinism is relevant only for declarative transformation languages (since other transformation languages are deterministic by nature). Executing a deterministic transformation means that the execution result does not depend on the rules' applications.

When TS_t is terminating and deterministic, it is said to be *convergent*, or *functional*, in which case we note the unique output model M' using a functional notation: $\mathsf{M}' = t(\mathsf{M})$. This notation is well-defined: M's image exists (by termination) and is unique (by determinism). For convergent transformation execution, we also denote indifferently by $t(\mathsf{M})$ the transformation specification *and* its execution starting from M.

Definition 4.4 (Type Correctness [TC])**.** *Let* t *be convergent. Then,* t *is* type correct *if it always outputs a conforming model.*

$$\forall \mathsf{M} \in \mathbb{L}(\mathsf{MM}), t(\mathsf{M}) \lhd \mathsf{MM}_\mathsf{t}$$

Proving this property for any input is hard: usually, type correctness is checked *a posteriori*, after the transformation completes, by running a pre-existing routine for checking type correctness.

4.4.2 Property Classes

A *property class* gathers properties that share the same form and that rely on the same artefacts. For a specific transformation obeying a given intent, a property class still needs to be concretised with respect to a transformation to afterwards effectively validate the transformation correctness. This will be illustrated later on *witness transformations* extracted from the Pwcs.

From our literature review, we collected several properties that we classify according to their *mathematical nature* and their *application level*, as illustrated in Figure 4.6:

Relation links artefacts from the transformation's left-hand side to corresponding artefacts from the right-hand side. The expressive power of such relations depends on both the axioms they satisfy, and the levels they operate at (either *syntactic* or *semantic*).

Preservation uses a property specification language $\mathcal{P}$ to express that whenever some properties on the left-hand side of the transformation hold, then it is "preserved" on the right-hand side. Preservation is also declined at both level, *syntactic* and *semantic*.

Behavioural Property differs from the previous property classes in the sense that it does not relate transformation artefacts, but rather characterises the transformation execution itself.

To distinguish between the different levels, we will use a generic notation $\mathcal{P}(\cdot)$ for specifying the nature of the languages involved, i.e. which artefacts, either syntactic or semantic, specification languages rely on. For example, $\mathcal{P}(\mathsf{MM})$ denotes a syntactic property specification language whose elements are MM's classes, attributes, references and so on. Notice however that theoretically comparing these property specification languages (i.e. their respective expressive power or their possible relationships) is beyond the scope of this thesis[2].

Unless explicitly stated, the following definitions assume a *convergent* and *type correct* transformation execution (otherwise, the definitions become meaningless).

4.4.2.1 Relations

A *Relation* class aims at establishing a mathematical relation between transformation's left-hand side (i.e. the source metamodel or the input model) and right-hand side (the target metamodel or the output model) artefacts.

Definition 4.5 (Syntactic [STR]/Semantic [SMR] Relations)**.** *A* syntactic relation property *is a relation* $\rho \subseteq \mathsf{M} \times \mathsf{M}'$ *over all legal model pairs* $(\mathsf{M}, \mathsf{M}')$ *such that* $\mathsf{M}' = t(\mathsf{M})$*. A* semantic relation property *is a relation* $\rho' \subseteq \mathbb{D}_{\mathsf{MM}_s} \times \mathbb{D}_{\mathsf{MM}_t}$ *over elements of the semantic domains attached to the source and target metamodels.*

Despite their resemblance, these relations fundamentally differs in nature: a *structural* relation links together *model elements*, like class instances, or association links; whereas a *semantic* relation links together *semantic domains' elements*.

How powerful and meaningful these relations are highly depends on the set of axioms these relations satisfy. For example, an interesting class of structural relations are *injective homomorphisms*, i.e. relations preserving models' structure with respect to external operations (this is for instance useful for the *Query* intent, among others).

A classical useful semantic relation is *simulation*, i.e. a relation ensuring that the execution of the input model cannot observationally be distinguished from the output execution. This means that the target can be transparently used *in lieu* of the source (which is particularly useful for the *Simulation* intent, among others).

An important syntactic relation is *traceability*, i.e. the ability for a transformation to create, either automatically or with the help of the designer, links from input artifacts to their corresponding output artifacts resulting from the transformation. We introduce a specific Definition because traceability characterises many intents.

Definition 4.6 (Traceability [TR])**.** *A syntactic relation property is a* traceability *property if a syntactic relation is created during the transformation execution for each model pair* $(\mathsf{M}, \mathsf{M}')$ *such that* $\mathsf{M}' = t(\mathsf{M})$*.*

[2]The paper submitted by Amrani et al. (2013) uses the same terminology, but proposes a *hierarchical* classification of fundamental properties with two main differences: no distinction appears between what we call *fundamental* properties and property CLASSES; instead, all properties are considered to be a specialisation of *Behavioural Property*, with two hierarchy paths for each of our *Application Layer*. This is not clear to us what this hierarchy relation means: when a property "inherits" from another, does it means that it is expressible in the same way, or that a verification technique used to discharge the verification of one property can similarly be used to discharge a sub-property? The first one is very unlikely, since for example, termination and determinism can not be expressed as a behavioural property. Similarly, following our study of Chapter 3, the techniques for our fundamental properties are radically different from the ones used for property classes.

4.4.2.2 Preservation

A *Preservation* property class stipulates that whenever some property, defined with respect to a *property language*, holds on the source metamodel and/or the input model, something equivalent, or similar in some sense, should hold on the target and/or output.

Definition 4.7 (Structural Preservation [STP]). *Let $\mathcal{P}(\mathsf{MM_s}, \mathbb{M})$ (resp. $\mathcal{P}(\mathsf{MM_t}, \mathbb{M})$) be a property language operating on the source (resp. the target) metamodel and the model set. A* structural preservation property *stipulates that whenever a formula $\pi \in \mathcal{P}(\mathsf{MM_s}, \mathbb{M})$ holds on the input model $\mathsf{M_i}$, then another pattern $\pi' \in \mathcal{P}(\mathsf{MM_t}, \mathbb{M})$ should hold on the corresponding output model $\mathsf{M_o}$.*

$$\mathsf{M_i} \vdash_{\mathsf{s}} \pi \in \mathcal{P}(\mathsf{MM_s}, \mathbb{M}) \implies \mathsf{M_o} \vdash_{\mathsf{t}} \pi' \in \mathcal{P}(\mathsf{MM_t}, \mathbb{M})$$

Property languages $\mathcal{P}(\mathsf{MM_s}, \mathbb{M})$ and $\mathcal{P}(\mathsf{MM_t}, \mathbb{M})$ obviously rely on the metamodels, but also on the models: it is often necessary to match elements values (such as an class attribute's value, or the object(s) pointed by a class reference, etc.). Typically, the OMG OCL (Object Management Group 2010) language can serve for expressing patterns. Notice however that the languages on both sides can differ: this is why we use two different satisfaction predicates $\vdash_{\mathsf{s}}$ and $\vdash_{\mathsf{s}}$.

Definition 4.8 (Semantic Preservation [SMP]). *Let $\mathcal{P}(\llbracket \mathsf{MM_s} \rrbracket)$ (resp. $\mathcal{P}(\llbracket \mathsf{MM_t} \rrbracket)$) be a property language on the source (resp. target) metamodel's semantics. A* semantic preservation *property stipulates that whenever the input model satisfies a semantic property ϕ, then the corresponding output model $\mathsf{M_o}$ satisfies a property ϕ' that is ensured to be equivalent in some sense to ϕ.*

$$\mathsf{M_i} \models_{\mathsf{s}} \phi \in \mathcal{P}(\llbracket \mathsf{MM_s} \rrbracket) \implies \mathsf{M_o} \models_{\mathsf{t}} \phi' \in \mathcal{P}(\llbracket \mathsf{MM_t} \rrbracket)$$

Here, languages $\mathcal{P}(\llbracket \mathsf{MM_s} \rrbracket)$ and $\mathcal{P}(\llbracket \mathsf{MM_t} \rrbracket)$ allow talking about elements of the semantic domains for each model. Similarly to the syntactic case, the satisfaction relation (noted here $\models_{\mathsf{s}}$ and $\models_{\mathsf{t}}$, to differentiate from the syntactic case) can differ in each side.

Ensuring that the properties expressed on the input and the output models are really equivalent is a challenge on itself. Sometimes, it is possible to automate this task by using so-called *property translators*, if the property languages on input and output are the same, or at least comparable. Generally however, when they differ too much, or the semantic gap between each metamodel is too deep, no general procedure exists for building such translators. This becomes the designer's job, with all the accompanying issues: aside from the properties' correctness, the translation can add another source of errors for the validation process (Varró and Pataricza 2003).

4.4.2.3 Behavioural Properties

The last property class qualifies transformations instead of (meta-)models of each side, and are particularly suitable for inplace, endogeneous transformations like *Simulation* or *Refinement*.

Definition 4.9 (Behavioural Property [BP]). *Suppose now that $\mathsf{TS_t}$ is not necessarily convergent. Let $\mathcal{P}(\mathsf{TS_t})$ be a property language over the transformation execution and $\mathsf{M_i} \in \mathbb{M}$ a legal input model for t. A* behavioural property *$\phi \in \mathcal{P}(\mathsf{TS_t})$ expresses the fact that starting from $\mathsf{M_i}$, the transformation execution satisfies ϕ.*

$$\mathsf{M_i}, \mathsf{TS_t} \models \phi \in \mathcal{P}(\mathsf{LTS_t})$$

This property class does not differ much from what is studied in the field of general-purpose programming language verification since here, the focus is on the correctness of the transformation's computation itself: thanks to the so-called decomposition theorem, any verification property of interest can be expressed as a conjunction of safety and liveness (see e.g. (Naumovich and L. Clarke 2000) for a formal definition of safety and liveness), making $\mathcal{P}(\mathsf{TS_t})$ representing temporal logics.

4.5 Five Examples

This Section presents the five intents we have chosen to illustrate further: *Query*, *Refinement*, *Translation*, *Analysis* and *Simulation*. Each intent is described systematically using the following approach:

1. We present informally the intent to convey the general idea behind it;
2. We review contributions from the literature to demonstrate different intent usages and help explain how the ModelTransformationIntent instance has been built;
3. We formalize the intent as an instance of the metamodel presented in figure 4.2. The goal of this formalization is to provide a *mapping* between the intent, informally presented in section 4.3, and its characteristic properties, defined in section 4.4.

4.5.1 Query

As with queries over databases, a query transformation applied to a model extracts a subset of information from that model. We refer to the extracted subset of information as a *view*. The Query/View/Transformations initial call for submissions (Gardner et al. 2003) defines a query as "an expression that is evaluated over a model" and a view as "a model which is completely derived from another model". This definition is very general since any automated transformation could be viewed as a way of completely deriving one model from another. In this paper, we define a query transformation as one that produces a *restrictive view* of the model by omitting a portion of the model - that is, it extracts a submodel. For example, the query "show only classes/associations of a class diagram" produces a restrictive view that extracts the submodel of a class diagram containing all and only the classes and associations.

4.5.1.1 Query in the Literature

Query transformations are often used as a preprocessing step to extract the portion of a model that is needed as input for another transformation. For example, in order to apply an analysis transformation to a state machine within a larger UML model, a query transformation will first be used to extract this state machine. Query transformations are also used to support the separation of concerns by extracting the submodels related to different concerns and then feeding these to their own transformations. For example, in the context of the Uwe web application, Koch et al. (2008) use query transformations to extract the subsets of the requirements model related to website function and website architecture, in order to feed these submodels into their own transformation chains that ultimately reintegrate these concerns downstream. We give additional examples of this technique below for the power window case study.

Model slicing represents a type of query transformation that has received recent attention by the modeling community. Model slicing, like program slicing, is intended to support human comprehension of a complex model by extracting submodels that are restricted to the behaviour and properties for a subset of model elements. Some of the slicing techniques produce *amorphous* (Harman, Binkley, and Danicic 1997) models, while other produce *structure-preserving* ones. The techniques that produce *structure-preserving* models can be considered as restrictive queries. For example, Lano and Rahimi (2011) describe slicing techniques for various UML diagrams with the goal of producing analysable models from those diagrams.

Similar approaches have been proposed for metamodel slicing. For example, Bae, Lee, and Chae (2008) use a model slicing technique to modularise and manage the complexity of the UML metamodel. The technique takes as input key elements of UML diagrams (e.g. Class Diagrams, Use Case Diagrams, etc) for which it produces a sub-metamodel that describe such diagrams, by navigating the associations emanating from the key elements. Following a similar line of thought, Sen et al. (2009) a more generic approach that makes use of a Kermeta model transformation to prune any given metamodel. The goal is to find a sub-metamodel for a

Attributes	
`name`	Query
`description`	Extract a submodel (the *view*) from a model that satisfies some criterion (the query).
`useContext`	1. Want to extract the relevant part (view) of a model for a task. 2. Want to decompose a model to manage complexity.
`example`	1. Extract the submodel that are immediate neighbours of a particular element. 2. Extract the submodel of structural elements from a UML model. 3. Model slicing. 4. Model decomposition.
`is_exogeneous`	True
`is_endogeneous`	True
`preconditions`	1. Must be able to characterize the submodel of interest using a condition expressible in terms of the metamodel of the base model.
Associations	
`mandatory`	1. **[T]** Terminating 2. **[TC]** Type Correct 3. **[STR]** The view must be a submodel of the base.
`optional`	1. **[D]** Deterministic 2. **[SMP]** Semantics preservation
`relatedIntent`	Abstraction

Table 4.1 – Query Intent Characterisation

given purpose, such as defining the allowed set of inputs for a model processing program or tool. The model transformation takes as inputs the large metamodel and a set of required classes and properties and returns a sub-metamodel including those classes and properties, and their mandatory dependencies. The authors also provide an additional algorithm to check that the pruned metamodel is a sub-type of the source metamodel. This ensures that instances of the pruned metamodel are also instances of the source metamodel.

Since the application of a query on a model produces a model fragment that is not necessarily well-formed, an important consideration for a query transformation is how to ensure type correctness (i.e. well-formed results). The work of Kelsen, Ma, and Glodt (2011), provides an efficient algorithm to address this problem by decomposing a fragment into its atomic constituents and then re-merging them while preserving well-formedness. The net effect is that the fragment is expanded to the nearest well-formed submodel that contains it.

4.5.1.2 Query Metamodel Instance

The attributes of the query transformation intent are shown in Table 4.1. If we consider the mandatory properties, *termination* **[T]** is a reasonable property to expect from a query – since it is of no use if it never produces a result. We also expect the resulting view to be well-formed with respect to the target metamodel and so it must be *type correct* **[TC]**. Most importantly, the resulting view must be a submodel of the input, or base, model. This is the key property that identifies a transformation as a (restrictive) query and can be formalised as a *structural relation* **[STR]** enforcing an *injective homomorphism* mapping from the view to the base.

A property that is optional is for the query to be *deterministic* **[D]** – i.e. that the query should always produce the same result on the same input model. Often this is expected, but there are cases where it is not needed. For example, consider a query transformation that extracts a submodel of a UML model showing an example of how a class is used. In this case, any sequence diagram that uses the class is sufficient and it is not necessary to always produce the same one. The optional property that the query be *semantics preserving* **[SMP]** means that the information in the view submodel should not change its meaning even though it is taken out of context of the whole model. This is often an important requirement when the view has a human consumer (e.g. model slicing) since otherwise the information in the view could be misleading.

The query transformation intent is related to the abstraction transformation intent as it can be seen as a form of information hiding.

4.5.2 Refinement

A transformation with the refinement intent is a transformation that produces a lower level specification, *e.g.,* a platform-specific model, from a higher level specification *e.g.,* a platform-independent model (A. G. Kleppe, Warmer, and Bast 2003).

4.5.2.1 Refinement in the Literature

From the literature review, a *Refinement* can either be an *interactive* or a *fully-automated* transformation. We describe the characteristics of each of them, based on more general studies for this particular intent.

Interactive Refinement Transformations The refinement approaches presented by Padberg (1999) and Scholz (1998) are rule-based. In the first approach, the rules need to adhere to specific properties in order to guarantee the preservation of safety properties and in the second approach specific refinement rules already exist. The user decides where to apply which rules.

Van der Straeten, Jonckers, and Mens (2007) present a formal approach to model refinement and its interplay with model refactoring. The user of the refinement needs to decide how to refine the models and afterwards behaviour preservation can be checked.

Fully-Automated Refinement Transformations Baresi et al. (2006) describe exogenous refinements of business-oriented architectures, abstracting from technology aspects, into service-oriented ones.

Mannadiar and Vangheluwe (2010) introduce two exogenous graph transformations, one of which is used to refine a domain-specific model (DSM) of the *PhoneApps* domain specific language (DSL) for a conference registration mobile application. The *PhoneApps* DSL captures both the behaviour and the visual structure of mobile device applications.

Tri and Tho (2012) discuss an approach for the automatic refinement of SEAM models. SEAM is a language and tool that supports visual modelling, that has the same modelling capability as UML with the additional advantage that SEAM can easily maintain consistency between design components since it can capture the entire system in a single view. Due to that single-view representation, the final SEAM model can become too complicated, which justify the introduction of a methodology to automatically refine abstract SEAM models into detailed SEAM models such that the final SEAM model can eventually be used to generate code.

4.5.2.2 Refinement Metamodel Instance

Table 4.2 instantiates the intent metamodel of Fig. 4.2 for the refinement intent, summarizing our findings in the literature. Since transformations with the refinement intent are required to add detail to existing models, it is intuitive that having a clear understanding of the information to be preserved and the information to be added are preconditions for such transformations. These preconditions were mentioned implicitly in all the previously described papers. For example, in the rule-based refinement approaches these preconditions are

Attributes	
`name`	Refinement
`description`	Add precision such that the output model contains at least the same amount of information as the input model. The information contained by a model is equivalent to the relevant questions that can be asked concerning the model (Giese, Levendovszky, and Vangheluwe 2007).
`useContext`	Add more detail to a model.
`example`	Going from a platform-independent model to a platform-specific model (A. G. Kleppe, Warmer, and Bast 2003).
`is_exogeneous`	True
`is_endogeneous`	True
`preconditions`	1. A clear understanding of the amount of information described by the input model, and how to preserve it. 2. A clear understanding of the information that needs to be added, and how to add it.
Associations	
`mandatory`	1. **[T]** Termination 2. **[TC]** Type Correctness 3. **[STR, SMR, STP, SMP]** Information Preservation 4. **[STR, SMR]** Information Creation
`optional`	
`relatedIntent`	Abstraction, Synthesis

Table 4.2 – Refinement Intent Characterisation

needed to be able to design the refinement rules as well as to apply them. In the studies discussed in subsection 4.5.2.1, it was usually mentioned that the mandatory *termination*, *type correctness*, *information preservation* and *information creation* properties stated in Table 4.2 need to be fulfilled. Whereas *termination*, *type correctness* have a one to one correspondence with properties **[T]** and **[TC]** in section 4.4, *information preservation* and *information creation* will generally need to be shown by a collection of several concrete properties, both at the structural and the semantic level. For example, the fact that there is a simulation between each input and output model's semantics might imply information preservation and can be expressed as *semantic relation* **[SMR]** property. Also, having a bijection between the syntactic elements of the input and the output models might imply information preservation and can be expressed as a *structural relation* **[STR]** property. It is also reasonable to think that *information preservation* might be expressed as a set of *structural preservation* **[STP]** properties where the information to be preserved is encoded in the syntactic property that is transported to the output model. The same reasoning holds at the semantic level for the usage of a set of *semantic preservation* **[SMP]** properties. Note that in Table 4.2 the notation **[STR, SMR, STP, SMP]** means that any non-empty combination of those four properties can be used to formally show *information preservation*.

Information creation implies the existence in the output model of syntactic and semantic elements that did not exist in the transformation's input model. It thus seems reasonable to think that **[STR, SMR]** can be helpful, if necessary, in showing *information creation*, depending on the notion *information creation* required by the considered transformation.

Some of the papers we surveyed have not explicitly verified all the properties in Table 4.2. Our work aims at identifying these gaps in order to allow for a more systematic engineering of model transformations with specific intents in the future. For example, Mannadiar and Vangheluwe (2010) and Tri and Tho (2012) do not verify the mandatory properties and but they use case studies to demonstrate that the refinement transformations fulfills their purpose. Both studies also informally discussed how a mapping is done between the input model and the refined model and thus, how information is preserved. Usually, the information creation property does not need to be checked explicitly, since it trivially follows from applying a refinement in the corresponding approaches. For endogenous approaches like, it is moreover usually trivial to check type correctness (cf. e.g. Padberg 1999).

Finally, the refinement intent is related to two other intents. It can be seen as the inverse of the abstraction intent. Moreover, a special case of refinement is synthesis where the target language is a platform-specific programming language. Further, refinement can also be seen as a means to generate models that can eventually be used for synthesis (Mannadiar and Vangheluwe 2010; Tri and Tho 2012).

4.5.3 Translation

A transformation with the *translation* intent is a transformation that translates the meaning of models conforming to a source metamodel into models conforming to a target metamodel. The resulting models can then be used to achieve tasks that are difficult, or impossible, to perform on the originals.

4.5.3.1 Translation in the Literature

From the review of the contributions present in the literature, it appears that a translation is performed for two main reasons: *bridging structures* to enable metamodel exchanges at a structural level (e.g., for importing models from another metamodeling framework); *delegating actions* to the target metamodel by *simulating*, or formally *analysing* input models using dedicated engines available for the output models. The delegation is valuable in the case where the cost of re-implementing a simulation/analysis engine for the source metamodel is too high.

Bridges The four-layered organisation depicted in Fig. 2.5 is shared by several technical spaces: modelware, grammarware, ontoware or dataware, to name just a few (Muller and Hassenforder 2005; Wimmer and Kramler 2005). Often, one has to bridge artifacts from one to another: for example, query languages and transaction operations in dataware are already available, taking advantage of SQL and its many capabilities and various implementations one can simply reuse instead of reimplementing things for a novel technical space. The goal of a bridge is to translate the meaning of the meta-metamodel itself, i.e. offering a way to automatically convert any metamodel of one technical space into another. This differs from the usual understanding of a transformation shown in Fig. 2.5, where the transformation is specified on a metamodel and executes on a model, not the level above. However, as previously noted, a meta-metamodel can usually be treated just as a metamodel and manipulated as such. Furthermore, bi-directional bridges are usually required for enabling round-trips between technical spaces.

Two papers (Muller and Hassenforder 2005; Wimmer and Kramler 2005) published in 2005 explicitly use the terms *grammarware* and *modelware* to refer to exchanges between textual and visual representations of models. Most probably, closing the gap between language theory (or compilation techniques, based on BNF grammars) and MDE (generally using MOF) are the most represented contributions (Deltombe, Le Goaer, and Barbier 2012; Izquierdo and García Molina 2012; Muller and Hassenforder 2005). Kern and his colleagues performed several bridges from various meta-metamodels into either MOF or its specific Eclipse implementation EMF: GOPRR, used in the commercial transformation engine MetaEdit+ (Kern 2009); Aris, the well-known enterprise modelling tool (Kern and S. Kühne 2007); Visio, the Microsoft general-purpose modelling tool (Kern and S. Kühne 2009). A comparative study (Kern, Hummel, and S. Kühne 2011) also describes the bFlow Toolbox, their integrated tool for performing these bridges seamlessly. Brunelière et al. (2010) and Bézivin et al. (2006) independently studied bi-directional bridges between Microsoft DSL Tools and Eclipse EMF, providing an efficient way to exchange models between one of the most popular DSL development tools.

Simulation Delegation A *Translation* is often specified for providing simulation (or execution) capabilities for models. This type of delegation is a popular approach for defining the semantics of DSLs (and, in this case, often more precisely called *Translational Semantics*). Since they capture domain expertise as concepts and rules with a precise meaning, the *Translation* just transposes these semantics in terms of a target metamodel that offers the necessary execution machinery. Another popular use for *Translation* consists of taking advantage of a richer framework to perform tasks specific to simulation, such as calibrating the parameter values of models to enhance their non-functional properties (typically, performance).

Rivera, Durán, and Vallecillo (2010) use Maude for specifying the behavioural semantics of domain specific modeling languages and for simulating the models by executing them using Maude rewriting rules. T. Kühne et al. (2009) define a transformation from Finite State Automata into Petri Nets, implementing the automata's semantics: by running the Petri Net translated model over an input sequence, it can check whether it belongs to the language accepted by the input automaton model.

MoTif is the result of combining the T-Core framework with a discrete event simulation language DEVS (Syriani and Vangheluwe 2010b). This allows model transformations to be expressed in a modular and compositional way together with the explicit introduction of a time dimension. Syriani and Vangheluwe (2008) demonstrated how adding the notion of time allows for the simulation-based design of reactive systems such as computer games. This allows the modelling of player behaviour and the incorporation of data about human players' behaviour and reaction times. The models of both player and game were used to evaluate, through simulation, the playability of a game design.

Troya, Rivera, and Vallecillo (2009) and Troya, Vallecillo, et al. (2013) employ simulations based on model transformations for reasoning about aspects of Quality of Service (QoS) such as performance and reliability. In their work, they add not only general runtime information to the models, as is for example, done by Engels et al. (2000) or in fUML, but they also add specific elements called observers to track information the designer is interested in. The authors used e-Motions (Rivera, Durán, and Vallecillo 2010) for implementing and executing the behaviour of the models to simulate. Internally, e-Motions is compiled to Maude.

Analysis Delegation A *Translation* can take advantage of the analysis capabilities of the target metamodel.

de Lara and Taentzer (2004) transform in models for process interaction expressed in a discrete event formalism tailored for the manufacturing domain into Timed Transition Petri Nets. This transformation is expected to terminate, to be deterministic, type correct and to preserve Process Interaction's behaviour. Termination, type correctness and behaviour preservation are proved informally, but determinism is proved using the classical critical pairs technique already implemented in AGG, the tool used to specify the transformation.

Varró, Varró-Gyapai, et al. (2006) prove the termination of graph transformations with negative application conditions by translating them into Petri Nets and showing that the resulting Petri Net is not partially repetitive, i.e. no (initial) marking has a firing sequence in which a transition occurs infinitely many times. Augur2, a graph transformation tool proposed by König and Kozioura (2008), approximates transformations with Petri Nets for analysing property preservation. Here, the property is specified as a graph pattern and then translated into an equivalent marking, which is checked for reachability. A counterexample is produced in case the marking is not reachable.

Narayanan and Karsai (2008a) proved reachability within StateCharts using a two-layered translation. First, a StateChart model is translated into an Extended Hybrid Automaton model, building traceability links between both instances. Then, the Extended Hybrid Automaton is translated in PROMELA, the entry language of the SPIN model-checker, where reachability can be checked. If a counterexample is produced, it can be traced back to the StateChart model following the traceability links previously established. Notice however that this technique is not general: it checks a particular property (reachability in the paper) on a particular StateChart model, and works only because the StateChart and the Hybrid Automaton models are proved to be bisimilar.

Cabot et al. (2010) automatically extract OCL invariants from bi-directional transformations expressed declaratively in QVT Object Management Group 2008, using a higher-order transformation. These in-

Attributes	
`name`	Translation
`description`	Translate the meaning of conforming input models into models conforming to a target metamodel to achieve a subsequent task.
`useContext`	Equip a DSL with an executable semantics, or perform a task difficult, or impossible to realise over the original models.
`example`	1. Provide a reference semantics for a DSL. (cf. Pwcs). 2. Exchange models between Microsoft Visio and Eclipse EMF (Kern and S. Kühne 2009). 3. Prove reachability in StateCharts using Promela (Narayanan and Karsai 2008a).
`is_exogeneous`	True
`is_endogeneous`	True
`preconditions`	
Associations	
`mandatory`	1. **[T]** Termination 2. **[D]** Determinism 3. **[TC]** Type Correctness 4. **[STP]** Semantic equivalence (*Bridge*) 5. **[SMR, SMP]** Observational equivalence / Similarity (*Simulation*, when source semantics available) 6. **[STR, STP]** Structural Preservation (*Simulation*, without source semantics available) 7. **[STP, SMP, SMR]** Soundness (*Analysis*)
`optional`	1. **[TR]** Backward Traceability to relate results back to the input. 2. Readability of the transformation's output for debugging purposes.
`relatedIntent`	Synthesis, Refinement, Analysis, Simulation

Table 4.3 – Translation Intent Characterisation

variants allow one to answer various questions about the transformation, such as whether a valid input or output model exists for the transformation, or whether an output model exists for any possible valid input. However, the actual invariant satisfaction problem is delegated to specialised tools able to work on UML models decorated with OCL invariant constraints.

4.5.3.2 Translation Metamodel Instance

Table 4.3 shows the ModelTransformationIntent's instance for the *Translation* intent. This intent is closely related to the following intents: *Synthesis*, *Refinement*, *Analysis* and *Simulation*. A *Synthesis* is a *Translation* where the output is expected to represent a programming language, generally outputted in textual form; the difference being that it generally induces a gap in the abstraction level. A *Synthesis* is therefore very close to a *Bridge* (i.e. the meaning of the source is integrally translated), or, often, a *Translation* with simulation delegation, when the resulting code is used for execution purposes. A *Refinement* is a (possibly endogeneous) *Translation*, with the specific task to add precision and information at each step. *Analysis* and *Simulation* are related because of the ability of a *Translation* to delegate such tasks to the target metamodel, where specialised engines already exist.

A *Translation* is by nature terminating **[T]** and deterministic **[D]**, otherwise the expected output models could never exist, or could be ambiguous regarding the original input. Because the output is expected to somehow "represent" the input, the transformation should be type correct **[TC]**.

The remaining mandatory properties depend on both the *Translation*'s nature and the existence of a precise semantics for the source metamodel.

Bridge If it is possible to attach a formal semantics to both meta-metamodels, then it becomes possible to formally compare conforming metamodels of both sides; otherwise, it should be possible to define structural preservation **[STP]** between both sides.

Simulation If the *Simulation* defines the source metamodel's semantics, then structural preservation **[STP]** is the only possible property. Otherwise, if the source metamodel has a predefined semantics, structural **[STP]**, but also semantics preservation **[SMP]**, are possible. It can also be interesting to prove a simulation relation **[SMR]** between the input and the output, thus ensuring for reactive systems that all actions of the input can actually be performed by the output (but obviously, also more actions, typically time-related).

Analysis Since an *Analysis* generally focuses on particular aspects of the inputs, the transformation should be "sound", i.e. it should verify some form of preservation of the property under analysis **[STP,SMP]**. Depending on the abstraction level difference of both sides, it is also possible to verify a simulation relation **[SMR]** between models in each side.

Some optional properties are also sometimes desirable. As already mentioned, a *Bridge* could sometimes be bidirectional. Traceability **[TR]** is also desirable for *Analysis* and *Simulation* to be able to relate results back to the input: for example, playing a counterexample obtained from an analysis in terms of the input to help identify errors.

4.5.4 Analysis

A transformation with the *Analysis* intent is a transformation that implements an analysis algorithm of any complexity. Examples include: the computation of a call graph for operations of a MOF model, detecting dead code or inapplicable rules, and the model-checking of a temporal formula over a given structure.

4.5.4.1 Analysis in the Literature

From the literature review, we noted two types of scenarios in which *Analysis* occurs. A transformation is a *Pure Analysis Transformation* if it expresses an analysis algorithm on its own, i.e. computes the necessary information for performing the analysis. Otherwise, it is a *Built-In Analysis Transformation* if it is executing with a transformation engine that is already equipped with analysis features.

Pure Analysis Transformations This *Analysis* scenario is, in fact, very rare. One reason is that specifying an analysis algorithm is typically complicated and so it is often easier instead to delegate the analysis to a dedicated tool after having translated the models. Furthermore, a key issue when analysing models is scalability, and this often requires the use of dedicated data structures to enable performance gains (for example, consider the use of binary decision diagrams for scalable model-checking).

Narayanan and Karsai (2008b) implemented a graph rewriting system in GREAT to transform UML activity diagrams to Communicating Sequential Process (CSP) models. The graph rewriting system was then checked for preserving structural correspondences between input and output models (property preservation). Unfortunately, no data related to the performance and scalability is given. Recently, Lúcio and Vangheluwe (2013a) explored the possibility of checking structural correspondence properties on DSLTrans transformations. The approach scales up to 21 rules for a transformation with acceptable computation times.

Built-In Analysis Transformations This *Analysis* scenario corresponds to the fact that a transformation is expressed in a transformation framework that is natively equipped with formal analysis capabilities. When possible, this represents a good choice, since the analyses are tailored for the transformation engine, ensuring good performance.

Attributes	
`name`	Analysis
`description`	Perform an analysis on the input models.
`useContext`	1. Develop an analysis algorithm using transformations. 2. Benefit from the built-in analysis capabilities of a transformation engine.
`example`	Reuse Maude's model-checking capabilities for model-checking graph transformations. (Rivera, Durán, and Vallecillo 2009)
`is_exogeneous`	True
`is_endogeneous`	False
`preconditions`	1. Access to analysis algorithms.
Associations	
`mandatory`	1. **[TC]** Type correctness
`optional`	
`relatedIntent`	Translation, Simulation

Table 4.4 – Analysis Intent Characterisation

Rivera, Durán, and Vallecillo (2009) encode graph transformations into Maude (Clavel et al. 2007). Graph transformations are specified visually by using AToM^3 (de Lara and Vangheluwe 2004) as a front-end, and encompass negative application conditions, well-formedness rules and both single and double pushout approaches. Since Maude provides reachability analysis, LTL model-checking, and theorem-proving capabilities, all these analysis become available for graph transformations, and the results are easily traced back due to their high-level encoding of (meta-)models. Gargantini, Riccobene, and Scandurra (2010) use Abstract State Machines (ASM) (Börger and Stärk 2003) to encode a DSLs' semantics. Metamodels and models are expressed with EMF whereas the transformation expressing their operational semantics is expressed with the ASM language. Using the built-in bidirectional translation into νSMV, LTL model-checking become possible in this framework.

Groove (Rensink 2003) allows the (bounded) model-checking of CTL formulæ over graph-based transformations with negative conditions (Kastenberg and Rensink 2006). The tool can also handle reachability analysis by expressing adequate invariants in CTL.

4.5.4.2 Analysis Metamodel Instance

Table 4.4 shows the `ModelTransformationIntent` instance for the *Analysis* intent. This intent is closely related to two other intents: *Translation* and *Simulation*: a *Translation* often delegates an analysis to the target metamodel; whereas a *Simulation* can directly benefit from the potentially available analysis capabilities of the simulation engine. When such capabilities exist, the task of the transformation designer consists of just specifying the transformation adequately (i.e. in the engine's own language), the analysis becoming the transformation's engine responsibility, not the designer's. For example, Rivera, Durán, and Vallecillo (2009) (cited as example in Tab. 4.4) use Maude as such a target, providing model-checking and theorem-proving analysis for all transformations specified within their framework.

Pure analysis transformations are obviously type correct when delivering a result **[TC]**. Beyond this, it is difficult to say more since it highly depends on the analysis being performed. They are not necessarily required to be terminating, or deterministic, since many types of static analysis are undecidable. For example, consider a model-checking procedure: it does not generally terminate for infinite systems, and if it does, the only requirement is to answer with one counterexample among all possible ones. In general, proving an analysis transformation's correctness is roughly equivalent to proving the correctness of an implementation with respect to an analysis algorithm. For example Lúcio and Vangheluwe (2013a) would be asked to prove that their transformations actually correctly realise model-checking.

The *Analysis* intent clearly needs further research. The fact that we cannot better characterise such an intent also comes from the fact that it is often, based on our observations, neither an atomic intent, nor has a single target (consider again model-checking: the analysis verdict is, if negative, accompanied with a counterexample).

4.5.5 Simulation

In the modeling community, simulation is a transformation that encodes some operational semantics of a language. Therefore it simply updates the state of a model in response to events (e.g., time, trigger, causal dependency). We can define a simulation such that its trace of execution is a label-transition system (LTS) where a node is a legal snapshot of the state of the model and a transition is the application of a rule.

Note that the term "model simulation" is understood differently in the modeling community and the simulation community. In the modeling community, model simulation normally refers to the development of an operational semantics for a modeling language, while in the simulation community, simulation (Shannon and Johannes 1976) refers to the process of designing a model of a real system and conducting experiments with this model for a certain purpose. Thus, the first interpretation can be seen as the enabler of the latter.

4.5.5.1 Simulation in the Literature

There is a large body of work discussing how to implement the operational semantics for modeling languages. Generally, there are two approaches for defining the behaviour of models: (*i*) by incorporating the runtime concepts into the metamodel and adding transformation rules for evolving the initial state of a model, and (*ii*) embedding the modeling language into some existing simulation formalism (as already discussed in Subsection 4.5.3.1). Thus, we refer the interesting reader for the second approach to Subsection 4.5.3.1 and deal in this subsection only with the first one.

Concerning the first approach, one way for defining an operational semantics is to introduce executability concerns by defining graph transformation rules operating on metamodel instances as proposed by Engels et al. (2000). Another possibility is to follow an object-oriented approach by specifying the behaviour of operations defined for the metaclasses of a modeling language (within the metamodels representing the abstract syntax of the languages) using a dedicated action language. Many action languages have been proposed, including the use of existing general purpose programming languages: Kermeta (Muller, Fleurey, and Jézéquel 2005), Smalltalk (Ducasse and Gîrba 2006), xCore (Clark, Evans, et al. 2004), EOL (Kolovos et al. 2012), the approach proposed by Scheidgen and J. Fischer (2007), and fUML (Mayerhofer, Langer, and Wimmer 2012). Prominent examples used in these papers are the definition of the operational semantics of Petri Nets or State Charts.

Most of this work only addresses the definition of the operational semantics of languages that model discrete systems without time, i.e., the time elapsed between two state changes is not considered. However, there are also some approaches dedicated to modeling specific real-time systems that require an explicit notion of time. For instance, de Lara, Guerra, et al. (2010) use so-called flow graph grammars for scheduling graph transformation rules and scheduling grammars for introducing an explicit notion of time for modeling a mail system.

An interesting problem and solution is presented in (Biermann, K. Ehrig, et al. 2009; Ermel and H. Ehrig 2008) where the goal is to have consistency between animation rules operating on the concrete syntax of a model and the simulation rules operating on the abstract syntax of the model. Although we consider this consistency property between animation and simulation as very important, in this paper we focus only on properties inherent to the simulation intent.

Attributes	
`name`	Simulation
`description`	To give an operational semantics to a modeling language by updating the state of the model.
`useContext`	Need to compute the trace of a model's execution, its final state or both.
`example`	Compute worst-case execution time, throughput, error rates of a production model.
`is_exogeneous`	False
`is_endogeneous`	True
`preconditions`	1. Access to intended semantics. 2. Metamodel contains runtime information as is currently provided by the dynamic metamodeling approach (Engels et al. 2000). As an example for runtime information consider the token concept in Petri Nets. 3. Modeling language has behaviour. 4. Real-time systems require a notion of time.
Associations	
`mandatory`	1. **[T]** Controlled Termination 2. **[TC]** Type correctness 3. **[BP]** Dynamic properties of the simulation
`optional`	1. Log of simulation is accessible. 2. Readability of the transformation's output. 3. If animation is provided, it has to correspond to the simulation
`relatedIntent`	Translation, Analysis, Synthesis, Animation

Table 4.5 – Simulation Intent

4.5.5.2 Simulation Metamodel Instance

Table 4.5 shows the `ModelTransformationIntent` instance for the *Simulation* intent. As already mentioned, the purpose of simulation in MDE is to give operational semantics to a modeling language by updating the state of a model. Of course, this applies only to behavioural models. The transformation is considered to be either exogenous if an already existing simulation formalism is selected for this purpose or endogenous if the behavioural semantics is directly attached to the language's metamodel.

In general, a simulation is a terminating transformation. When a terminating condition is met, the simulation must stop. This condition can be based on the state of a model, on the gained information, or on time. This latter point means that the transformations are expected to terminate at some point in time, although it may happen that the simulation has to be stopped even though there are still rules that can be applied. Concerning the second point, sometimes the successful execution of the simulation is meant to be non-terminating unless an information saturation point is met. This can be a failure or an exception case arises that may lead to rejecting the hypothesis to be tested, or the opposite, the information gained allows to accept the hypothesis. To sum up, *controlled termination* **[T]** has to be supported.

Whether a simulation transformation is deterministic **[D]** depends on the system being modelled and cannot be decided on a general basis. If the system is deterministic, the simulation should be deterministic, too; otherwise, one of the transformation rules is non-deterministically selected and applied.

Each simulation step should result in a valid model with respect to its metamodel: a stronger *type correctness* **[TC]** property than the one of Definition 4.4 needs to hold. However, ensuring this may require a sequence of several transformation rules corresponding to a single logical simulation step (making transformation rules act similarly to a *transaction*).

Proving the simulation to be correct usually required that a set of *behavioral properties* **[BP]** hold: among others, invariants, or reachability constraints over the set of reachable states of the simulated system.

Logging of transformation execution events is considered to be an useful but optional property. Especially, some transformation engines are able to produce complete logs, e.g., the order of the rules applied, the different execution states, the binding of the rules and timing information. Some approaches also provide the means to automatically produce views on the logging information to support better understandability of the simulation results, e.g., to show the number of events per event type. This is also connected to an optional property, the readability of the transformation's output. Here, not only the output model has to be in a human-readable form, but also the logging information since it may be considered to form a critical aspect of the simulation result.

Because an animation of a simulation is optional, we consequently consider the consistency property between animation and simulation as an optional property.

4.6 The Power Window Case Study

This section introduces the case study for this paper, developed in the context of an industrial project aimed at building control software for an automobile power window (Denil 2013; Mustafiz et al. 2012). A power window is basically an electrically powered window. Such devices exist in the majority of the automobiles produced today. Besides lifting and descending the window, a power window also includes an increasing set of additional functionalities, aimed at improving the comfort and security of the vehicle's passengers. To manage this complexity while reducing costs, automotive manufacturers use software to handle the control of such physical devices.

The Power Window Case Study (PWCS) consists of a chain of model transformations aiming at generating control software for a power window as C code, starting from high level requirements. The whole transformation chain contains 37 transformations and involves 28 different metamodels. The PWCSserves as an experimental platform for the research presented in this paper. We use it for two complementary purposes.

- since the PWCS was developed independently of our research, it presents an unbiased collection of transformations that we use for partially validating the Intents Catalogue of Section 4.3: we will identify occurences of intents in this real-world transformation chain.
- The PWCS can be used to illustrate the practical usefulness of our Intent/Property mapping in Section 4.5 and of our abstract framework for formalizing properties provided in Section 4.4: in particular, we extract from the transformation chain two witness transformations in Section 4.7 for two exemplary intents, namely, *Translation* and *Simulation*. By using our mapping we illustrate how to build concrete transformation properties that help in showing the correctness of the two witness transformations by instantiating the abstract framework.

Section 4.6.1 presents the FTG+PM formalism in which the PWCS transformation chain is expressed. Section 4.6.2 describes the transformation chain itself, with an emphasis on the steps the witness transformations are extracted from. Section 4.7.3 links the transformations appearing in the PWCS with their respective intents.

4.6.1 Formalism Transformation Graph and Process Model

Figure 4.7 depicts a condensed version of the FTG+PM for describing the Power Window software. It consists in two parts:

FTG (Formalism Transformation Graph) on the right side of the Figure, captures a set of Domain-Specific Formalisms, visually defined as labelled rectangles. Those formalisms are connected to small labelled circles that represent transformations from one formalism to the other.

PM (Process Model) on the left side of the Figure, represents a diagram describing a set of ordered tasks necessary to produce the Power Window code. This diagram is directly inspired from the UML Activity Diagram Language (Object Management Group 2011a).

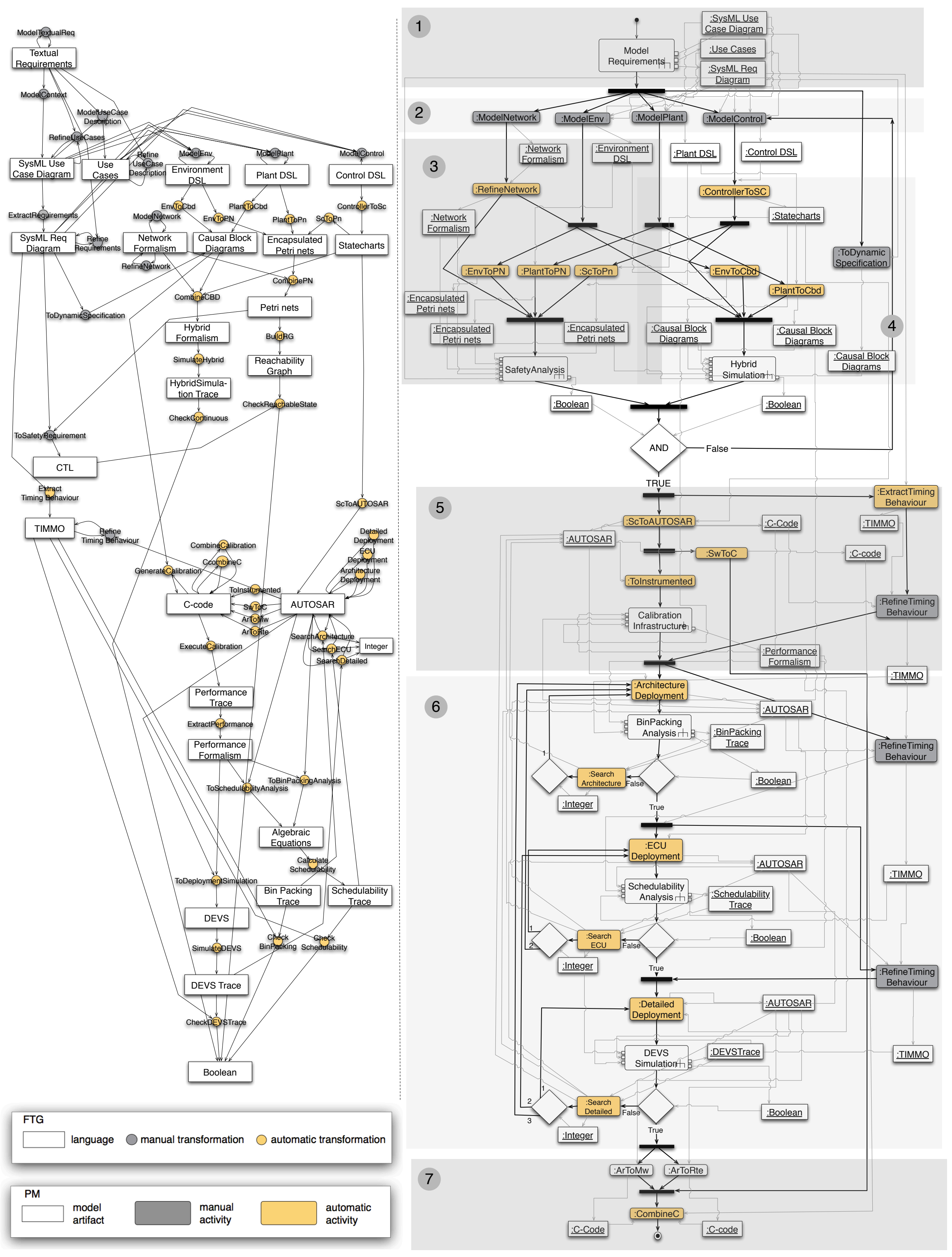


Figure 4.7 – FTG (on the left) and PM (on the right) for Power Window software development

Both sides are in a direct correspondence as follows: *actions*, represented by round-edged rectangles in the PM, are typed by executions of the transformations with the same label declared in the FTG; whereas *data objects*, represented by square-edged rectangles in the PM, represent instances of the models declared in the FTG that are produced and/or consumed by actions. A model is an instance of the formalism declared in the FTG with the same label. A *thin arrow* in the PM indicates a data flow; whereas a *think arrow* indicates control flow. An *horizontal bar* depicts a join/fork control flow construct; and a *decision node* has the form of a diamond. Some actions are greyed: this indicates that the corresponding transformation requires manual intervention (e.g., to provide a configuration parameter, or because it denotes a manual task), contrary to the others that are fully automated.

4.6.2 Description

The transformation chain of Figure 4.7 is constituted by several phases, identified by numbered layers: *Requirements Engineering*, *Design*, *Verification*, *Simulation*, *Calibration*, *Deployment* and finally *Code Generation*. Some of these phases phases can be executed in parallel (e.g., **??** and **??** that are later joinded before **??**). Others, such as **??** (*Deployment*), contains loops.

We focus on the four first phases that lead to a viable, trusted system that can then be calibrated and deployed: the construction of transformation properties used in Section **??** are extracted from these phases. A detailed description of the PWCS can be found in the corresponding literature (Denil 2013; Lúcio, Joachim, et al. 2012).

Note that the PM of Fig. 4.7 contains collapsed blocks (e.g., *Model Requirements*, *Safety Analysis* or *Hybrid Simulation*) that hide the details of the corresponding tasks. Whenever relevant for the explanation, we will explicitly detail the blocks content.

1. **Requirements Engineering.** Before design activities can start, engineers have to extract requirements from legal and technical documents in order to produce requirement and use case diagrams that document what is expected from the system. These transformations are usually done manually, although some parts could be automated (e.g., for populating those diagrams).

2. **Design.** Using these requirement artefacts, software engineers start the design activity following design practices inspired from control theory Dorf 2011: the *controller* is the piece of software controlling the window's functionalities; the *process* (also called *plant*) is the physical power glass window with all its mechanical and electrical components, i.e. the mechanical lift, the electrical engine and the sensors detecting the window's position or collision events; and the *environment* is constituted of the human actors and the other vehicle subsystems, e.g. the central locking system, the ignition system, etc. (the way the PWCS is built closely follows the work by Mostermann and Vangheluwe Mosterman and Vangheluwe 2004). Each aspect of the system is captured by a dedicated DSL (Domain-Specific Languages explicitly named *Environment*, *Plant* and *Control* in Fig. 4.7), later bound together using an extra *Network* DSL for expressing how they interact.

After this phase, the entire system is modelled and can be deployed. However, regulations in the automotive sector have strong security concerns that need to be addressed at early stages of the system design. Since the Power Window is a critical system, two validation tasks, namely Verification and Simulation, are conducted in parallel in the PWCS to ensure that the code generated from the models are trustable.

3. **Verification.** Formal Verification is applied by translating all domain-specific models from the previous stage into corresponding Petri Nets Peterson 1977. All the resulting Nets are then composed accordingly to the *Network* model, to obtain a fully functional Petri Net, on which reachability analysis of undesired states, specified according to the requirements, is then checked.

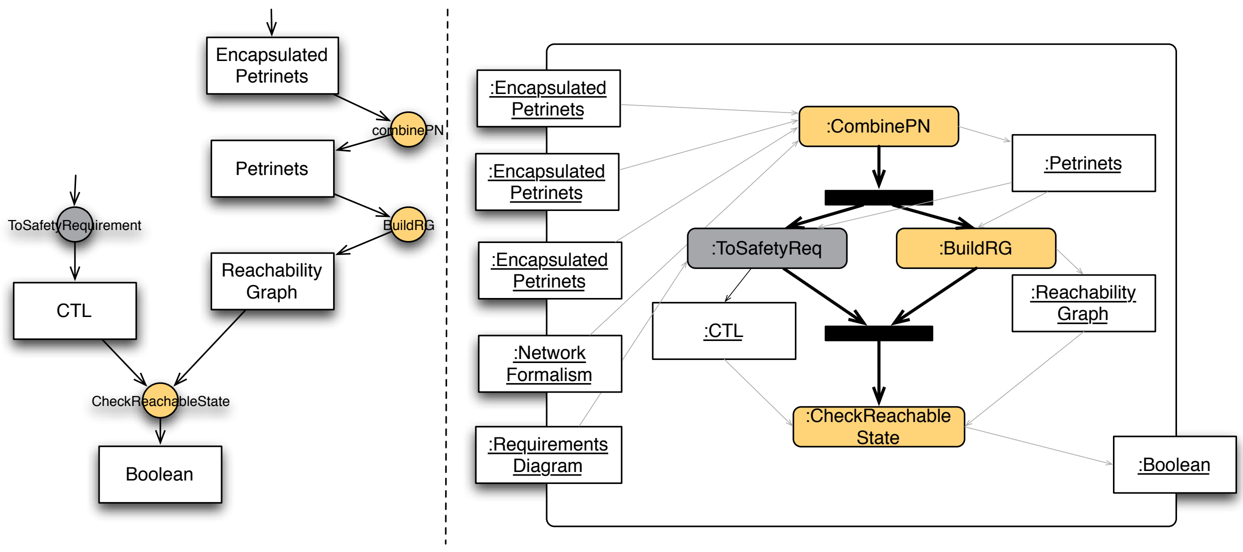


Figure 4.8 – Safety Analysis Ftg+Pm Slice, with Ftg on the left and Pm on the right

Figure 4.8 describes the details of the collapsed block corresponding for safety analysis. On the right side, the :CombinePN composes the five models resulting from the previous *Design* activity into a combined Petri Net that describes the behaviour of the whole system. This combined Petri Net is the source of two activities performed in parallel: the :toSafetyReq, which requires human intervention, produces a set of Ctl formulas encoding the requirements based on a safety requirement model; and the :BuildRG automatically builds the reachability graph corresponding to the combined Petri Net model. These activities are then joined together, since they are prerequisites before the ReachableState action is executed for model-checking the combined Petri Net behaviour against the safety requirements, and produces a verdict (given as a boolean value).

④ **Simulation.** On the other hand, a simulation of the whole system is conducted to evaluate the responsivity when interacting with the passengers. The continuous behaviour of the window is modeled using a hybrid formalism: the models for the environment and the plant resulting from the *Design* phase are translated in Causal Block Diagrams (Cbds)[3] whereas the controller model is transformed into a StateChart. The process of verifying the continuous behaviour is similar to the *Vertification* phase, although as a requirement language CBDs are also used.

When the *Verification* and *Simulation* tasks are both completed, engineers can think about how to efficiently deploy the system on the platforms they target (Phases ⑤ to ⑦). The *Calibration* phase aims at extracting a performance model that gives measurements about the execution times corresponding to the different use cases. This performance model is then used during the *Deployment* phase for selecting a deployment solution with real-time behaviour where spatial and temporal requirements are respected. Finally, when a feasible solution is found, the code specific to the target platforms can be synthesised: this includes the code of the application itself, but also the code corresponding to the middleware and to the runtime environment.

[3]Causal Block Diagrams are a general-purpose formalism used for modelling causal, continuous-time systems, mainly used in tools like Simulink.

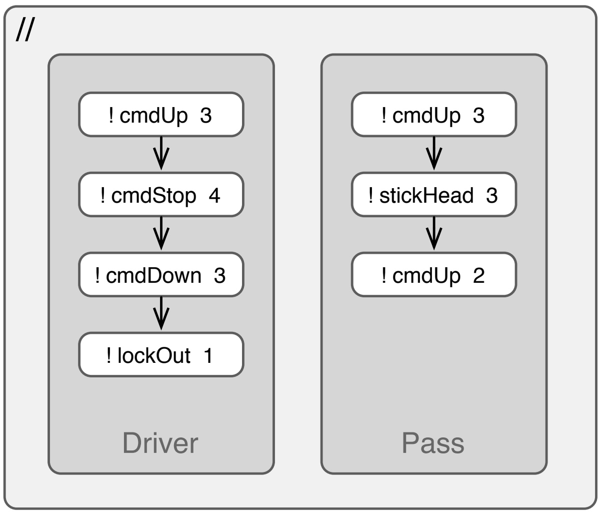


Figure 4.9 – Example model for the *Environment DSL* language

4.7 Identifying Transformation Intents within the Pwcs

This Section takes two transformations from the Pwcs and describes for each of them how we identified their intent by following the process described in Figure 4.4: the first one is a *Translation* whereas the second one is a *Simulation*. We also illustrate for each transformation a concretisation of one property among those ensuring the transformation correctness. Finally in Section 4.7.3, we provide an overview of the intents identified within the Pwcs.

4.7.1 Translation

As a first example transformation for which we want to identify the intent we chose the *EnvToPN* transformation.

Selecting Intent using Description Attribute As afore briefly mentioned, the *EnvToPN* transformation takes a model expressed in the *Environment DSL* language and produces as result a model in the *Encapsulated Petri Nets* language. The purpose of this translation is to profit from the fact that the *Encapsulated Petri Net* has a well known and studied semantics which can be used as a semantic domain for the *analysis* of models of the *Environment DSL* language. The *Environment DSL* language has no explicitly formalized semantics and the role of the translation is to provide an artifact that can explicitly produce such semantics in the form of a Petri Net like formalism. Consequently, the obvious intent of the transformation is to provide *Translational Semantics* to *Environment DSL* models in terms of the Petri Net formalism. This fits nicely to the description of the *Translation* intent in Table 4.3.

Checking Remaining Intent Attributes The *useContext* mentioned in Table 4.3 and the fact that the transformation needs to be exogeneous fit both as well. In what concerns the *example* attribute, example 3 is the same kind of translation having *analysis* as its purpose.

Checking Appropriateness of Mandatory Properties We switch to checking if the mandatory intent properties are appropriate for the *EnvToPN* transformation. As can be observed in Table 4.3, the *Translation* intent has as mandatory properties *termination*, *determinism*, *type correctness* and *soundness*. As for the first three properties, it is obvious that they are appropriate.

Because the semantics of models written in the *Environment DSL* language is not defined, it is not meaningful to discuss the preservation of semantic properties for the *EnvToPN* transformation. It is

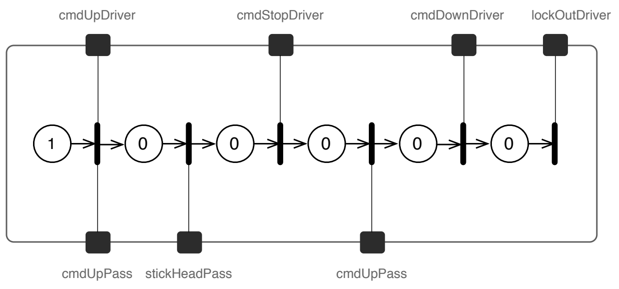


Figure 4.10 – Example model for the *Encapsulated Petri Nets* language

however meaningful to preserve syntactic properties of an *Environment DSL* model that reflect its correct translation into an *Encapsulated Petri Net* model. Consequently, we can conclude that the mandatory properties are appropriate.

Selecting Optional Properties The optional properties for the translation intent are *backward traceability* and *readability*. The implementation traceability was not required given the simple nature of the properties being verified in the PWCS. Special care was however devoted to readability of the transformation's output such that, given the very visual nature of Petri Nets, the models resulting from the *EnvToPN* transformations could be understood by humans. This proved useful both for debugging and especially for demoing purposes, as the PWCS has been presented at several venues as an example of transformation chaining for the contruction of complex systems using MDE principles.

Outlooking to Validating Properties After having identified the intent for the *EnvToPN* transformation, we want to validate as described in Fig. 4.5 if its mandatory/selected optional properties are indeed fulfilled. We give a brief idea of this process and first have a more detailed look into the *EnvToPN* transformation.

Fig. 4.10 depicts the result of executing the *EnvToPN* transformation on the model in Fig. 4.9, which represents the parallel issuing of two sequences of statements. The box annotated with *'Driver'* sends out four sequential commands to the set of buttons on the driver's door and the box annotated with *'Pass'* send three sequential commands to the set of buttons on the passenger's door. Note that each command box has a number in it, which represents the amount of time during which the command is in effect. The translational semantics of the model in Fig. 4.9 is produced by transformation as the *Encapsulated Petri Net* model in Fig. 4.10. The resulting model is a Petri Net where the commands issued by the driver are merged with the commands issued by the passenger along the same Petri Net transition timeline. Petri Net transitions pass messages to outside of the component via ports, represented as black squares on the border of the component. Due to timing constraints the driver and the passenger commands are sometimes issued simultaneously. In the model in Fig. 4.10 this translates into the fact that some of the transitions on the Petri Net in Fig. 4.10 are connected to more than one *port* in the component.

In Fig. 4.11, we express a structural preservation **[STP]** transformation property that we wish to hold for the *EnvToPN* transformation. Akehurst, Kent, and Patrascoiu (2003), Büttner, Egea, Cabot, and Gogolla (2012), Büttner, Egea, and Cabot (2012), Cariou et al. (2009), Gogolla and Vallecillo (2011), Guerra, De Lara, et al. (2013), Lúcio and Vangheluwe (2013b), Vallecillo and Gogolla (2012) have studied structural preservation properties. They allow expressing in a similar fashion how the structure of the transformation's input model influences the structure of the transformation's output model. In order to express such properties for all executions of a transformation those languages typically use a mix of the transformation's source and target metamodel elements, additional constraint languages (e.g. Büttner, Egea, and Cabot (2012) and Büttner, Egea, Cabot, and Gogolla (2012), Cariou et al. (2009), Gogolla and Vallecillo (2011) and Guerra, De Lara, et al. (2013) and Vallecillo and Gogolla (2012) all used OCL) and often metaclasses allowing describing traceability

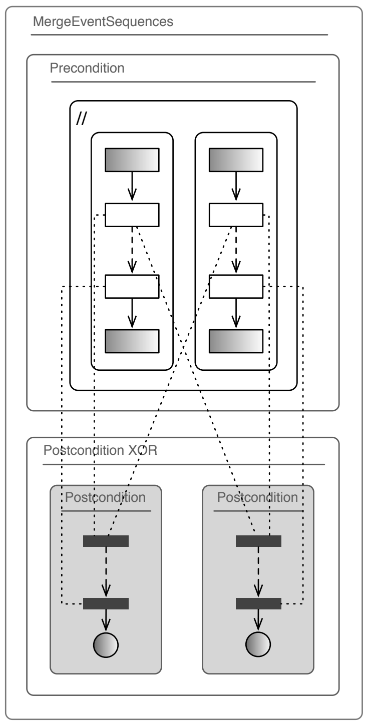


Figure 4.11 – Syntactic property preservation example for the EnvToPN Power Window transformation

connections between the source and target metamodel elements (Akehurst, Kent, and Patrascoiu 2003; Büttner, Egea, Cabot, and Gogolla 2012; Lúcio and Vangheluwe 2013b). The property language designed by Lúcio and Vangheluwe (2013b) served as a basis for expressing the transformation property in Fig. 4.11.

The **[STP]** transformation property in Fig. 4.11 states that whenever the input model includes two sequences of parallel output commands, each of those sequences containing a *first* and a *last* command, the resulting output model will merge the two *first* commands as a single transition and the final transition is coming from the *last* command of either the first or the second sequence (but not both, as denoted by the XOR operator). Note that in Fig. 4.11 the thick dashed arrows between *Precondition* or *Postcondition* elements state those elements are indirectly linked; thin dashed arrows between *Precondition* and *Postcondition* elements represent traceability links; and blend colored elements represent negative condition, i.e., elements that cannot not occur in input/output models. In Section 4.4 we have defined **[STP]** properties as follows:

$$\mathsf{M_i} \vdash_{\mathsf{s}} \pi \in \mathcal{L}(\mathsf{MM_s}) \implies \mathsf{M_o} \vdash_{\mathsf{t}} \pi' \in \mathcal{L}(\mathsf{MM_t})$$

For the example property in Fig. 4.11, π and π' correspond respectively to the *Precondition* and *Postcondition* part of the property. Also, $\mathsf{M_i}$ and $\mathsf{M_o}$ are instances of the *Environment DSL* and *Encapsulated Petri Net* languages respectively and $\mathsf{M_o}$ is the result of applying the *EnvToPN* transformation to $\mathsf{M_i}$. If $\mathsf{M_i}$ instantiates the property's *Precondition* pattern π, then $\mathsf{M_o}$ necessarily instantiates the property's *Postcondition* pattern π'. Note also that π and π' are related by the property's traceability links connecting the property's *Precondition* and *Postcondition* elements.

4.7.2 Simulation

As a second example transformation we have selected a Petri Net simulation called *BuildRG* and located in Area **??**. We describe the intent identification of this transformation with less detail. In particular, we select

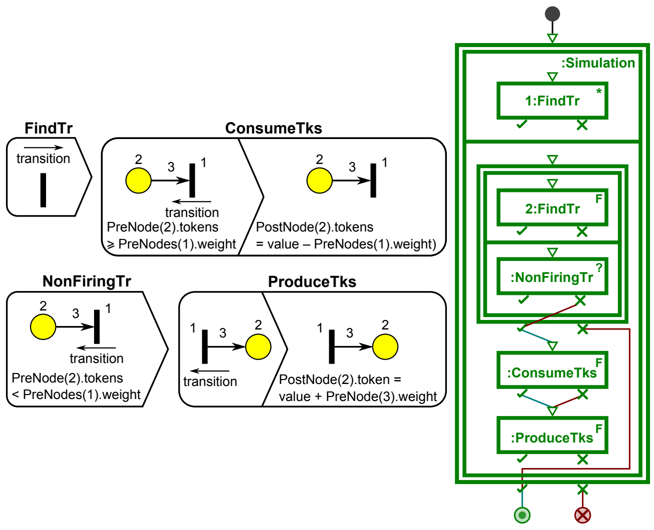


Figure 4.12 – Simulation of Petri Nets: transformation rules (left) and schedule (right).

the intent using the description attribute and then describe merely why one of the mandatory properties is appropriate.

Selecting Intent using Description Attribute The transformation *BuildRG* specifies the semantics of Place/Transition Petri Nets operationally, i.e. in an inplace fashion. This fits obviously to the description attribute of the *Simulation* intent in Table 4.5.

Checking Appropriateness of Mandatory Property [BP] Let us call $t = (\mathsf{MM_s}, \mathsf{MM_t}, \mathsf{spec})$ the corresponding transformation specification. Since a simulation is inplace, $\mathsf{MM_s}$ and $\mathsf{MM_t}$ both represent a metamodel for Place/Transition Petri Nets. The specification follows a graph-based approach, using MoTif (Syriani and Vangheluwe 2011) as a model transformation language $\mathcal{L}$. The attached transformation execution $\mathsf{TS_t}$ corresponds to the semantics on the MoTif execution engine.

As shown in Fig. 4.12, it is possible in MoTif to specify the transformation rules (on the left, adapted from T. Kühne et al. 2010 for the purpose of the Pwcs), but also their scheduling (on the right). Four rules compose the specification: `FindTr`, `ConsumeTks`, `NonFiringTr` and `ProduceTks`. The rules are organized in two nested loops; the outer-most, called Simulation, runs in an infinite loop. The first rule `FindTr` (which is a query consisting of solely a LHS) selects one transition. The transition found is assigned to a pivot variable *transition* to be referred by subsequent rules. Then, the transformation ensures that only firing transitions will be processed. To find enabled transitions, the transformation iterates through all transitions until one has been found that does *not* satisfy the pattern of a *non*-firing transition. This is done by iterating over every transition in the model and, if the `NonFiringTr` rule cannot succeed, it is assigned to the pivot in order to fire the transition. This interruption in the inner-loop is represented by connection from the fail port of the rule `NonFiringTr` to the success port of the enclosing rule block. When a firing transition is found, it is assigned the *transition* pivot, replacing the former transition. Then, tokens are transferred along this transition as depicted by rules `ConsumeTks` and `ProduceTks`. These two rules are applied for all adjacent arcs and places (denoted by an 'F'). After that, the first `FindTr` rule is applied again recursively, by re-matching the new model looking for a transition given the new marking. This control flow goes on until no more transitions are fireable. This transformation succeeds if the input model contains a transition and fails if not.

Outlook to Validating Mandatory Property [BP] After having identified the intent for the *BuildRG* transformation, we want to validate as described in Fig. 4.5 if its mandatory/selected optional prop-

Intent	Transformations
Restrictive Query	`CheckReachableState, CheckContinuous, ExtractPerformance, CheckBinPacking, SearchArchitecture, SearchECU, SearchDetailed, CheckSchedulability, CheckDEVSTrace`
Refinement	`ArchitectureDeployment, ECUDeployment, DetailedDeployment`
Abstraction	`ExtractTimingBehaviour`
Synthesis	`SCToAUTOSAR, SwToC, ToInstrumented, GenerateCalibration, ArToMw, ArToRte`
Translation	`EnvToPN, PlantToPN, ScToPN, ControllerToSc, EnvToCBD, PlantToCBD, ToBinPackingAnalysis, ToSchedulabilityAnalysis, ToDeploymentSimulation`
Simulation	`BuildRG, SimulateHybrid, ExecuteCalibration, SimulateDEVS, CalculateSchedulability`
Composition	`CombinePN, CombineCBD, CombineCalibration, CombineC`

Table 4.6 – Intents of transformations present in the Pwcs.

erties are indeed fulfilled. We give a brief idea of this process for the identified mandatory property **[BP]** as described above.

The **[BP]** mandatory intent property can be concretized in the following way. As defined in Definition 4.9, a **[BP]** depends on an input model. In our case, $\mathsf{M_i}$ is the Petri Net model illustrated in Fig. 9 of Lúcio, Mustafiz, et al. 2013 that models the behaviour of the power window control software. In particular, each place in that Petri Net must contain at most one token during its execution. Petri Nets of this kind are also called *1-safe*. Therefore, an indicator of the correctness of the *BuildRG* transformation could be that at each step of the simulation each place has at most one token, assuming of course the Petri Net model being simulated is indeed 1-safe. The following **[BP]** states that given that input Petri Net, the execution of the transformation from Fig. 4.12 will always satisfy that property ϕ expressed in Ltl[4]. Here, M denotes the marking of a place p in $\mathsf{M_i}$.

$$\forall p \in \mathsf{M_i}\,.\ \mathsf{M_i}, \mathsf{TS_t} \models \neg \,\square\, (|M(p)| > 1)$$

Simulation transformations often include a loop where the same steps are re-executed on the resulting model. Furthermore, some steps may require choices to be done. Thus a simulation execution consists of one branch in $\mathsf{TS_t}$. In our example, every loop of the simulation starts by looking for a non-firing transition. However, when found, only one such transition is taken into consideration. Therefore to verify that a transformation does not satisfy ϕ, it suffices to check whether ϕ is not satisfied at each step in one simulation execution. Some approaches allow one to specify such invariants on the model transformation steps directly (cf. e.g. Wimmer, Kappel, et al. 2009).

4.7.3 Overview

As a partial validation of our description framework, we applied the scenario described in Fig. 4.4: we identified the intents of transformations of the Pwcs. In Table 4.6 the Pwcs transformations are classified according to the intent they obey. As one can easily notice, some of our intents are not represented at all. This is not surprising however: the purpose of the Pwcs transformation chain is to generate trustable C code for hardware execution; consequently, intents related to transformation visualisation, synchronisation or syntactic manipulation, among others, have no corresponding transformation in the chain. On the contrary, some of the intents collect many transformations: most of the transformations belong to either *Query*, *Refinement*, *Synthesis*, *Translation* or *Simulation*.

[4]E. M. Clarke, Grumberg, and Peled (1999) provides more details on the syntax, semantics of Ltl, as well as its expressive power and a comparison with Ctl.

Transformation	Description	Precond.	Mandatory	Optional
`BuildRG`	The BuildRG transformation simulates the execution of a Place/Transition Petri Net in order to build that net's reachability graph. Safety requirements for the power window can then be checked on the produced reachability graph.	(1),(2),(3)	(1),(2),(3),(4)	(1),(2)
`SimulateHybrid`	This transformation simulates the interactions between the physical window and the designed window controller. While the physical window has continuous behaviour, i.e., the window is moving up/down in a continuous manner, the user can push buttons to control the window that correspond to discrete signals. Casual Block Diagrams (CBD) representing the window behaviour are co-simulated with Statecharts that represent the user events.	(1),(2),(3),(4)	(1),(2),(3),(4)	(1),(2),(3)
`CalculateBinPacking`	The bin packing transformation is a simple transformation that simulates and evaluates the usage of a hardware component by calculating the sum of each execution time of a function mapped to the hardware component divided by the period of the functions. The transformation is implemented as an equation and produces measurements.	(1),(2),(3),(4)	(1),(2),(3),(4)	(1),(2)
`ExecuteCalibration`	By running a simulation on a host computer, the input to execute an instrumented software application on the target platform for collecting measurements to obtain calibration parameters.	(1),(2),(3),(4),(5)	(1),(2),(3),(4)	(1),(2)
`SimulateDEVS`	AUTOSAR models are translated into DEVS for producing traces by simulating the DEVS representations. The output of the DEVS simulations are traces that are further analyzed by a boolean formula.	(1),(2),(3),(4),(5)	(1),(2),(3),(4)	(1),(2)

Table 4.7 – Model transformation examples from the PWCS falling under the *Simulation* intent

In addition to the coarse-grained alignment, Tables 4.7 and 4.8 show the detailed results for the *Simulation* and *Translation* transformations identified in the PWCS. For each transformation, we describe what the transformation does, and report on the satisfaction of the preconditions, and mandatory/optional properties. This gives an interesting snapshot on the applicability of our method in real-world transformation chains. Although both tables collect transformations with the same intent, the preconditions and even the mandatory properties are not the same for all transformations: In Table 4.7, these differences are due to the fact that pre-conditions (4) and (5) in the Simulation intent (Table 4.5) are considered optional; in Table 4.8 the differences in the mandatory properties come from the fact that some properties in the Translation intent (Table 4.3) are dependent on the translation type (bridge, simulation, or analysis).

4.8 Discussion

The notion of *intent* is central to our Description Framework: it captures the adequate abstraction level for expressing property classes. We discuss here this notion from a broader viewpoint and sketch possible future work around this notion.

4.8.1 Towards an Intent Taxonomy

At a first glance, our research problem required to identify the various model transformation intents reported in the literature: Section 4.3 describes an Intent Catalog that can be viewed as a description of possible instances of the ModelTransformationIntent class of the metamodel appearing in Figure 4.2.

This description was "flat" in the sense that it only consists of a comprehensive list where instances are not really linked together, except for the intents illustrated in this thesis: by expliciting the possible values of the relatedIntent reference, we showed for example that *Simulation*, *Translation* and *Analysis* share some similarities.

Transformation	Description	Precond.	Mandatory	Optional
`EnvToPN`	Build a Petri Net representation of a specialised model of the passenger's interactions with the powerwindow.	None	(1),(2),(3),(7)	(2)
`SCToPN`	Build a Petri Net representation of a statechart model representing the powerwindow control software to allow checking power window security requirements.	None	(1),(2),(3),(4)	(2)
`PlantToPN`	Build a Petri Net representation of a specialised model of the powerwindow physical configuration to allow checking power window security requirements.	None	(1),(2),(3),(7)	(2)
`ControllerToSC`	Produce a statechart for providing semantics to a specialised model of the power window control flow.	None	(1),(2),(3),(7)	(2)
`PlantToCBD`	Generate a causal block diagram (as python code) that can be used both for simulation of the combined system and for calibration of the combined system	None	(1),(2),(3),(6)	(2)
`EnvToCBD`	Generate a causal block diagram (as python code) that can be used both for simulation of the combined system and for calibration of the combined system	None	(1),(2),(3),(6)	(2)
`ToBinPackingAnalysis`	Build an equational algebraic representation of the dynamic behavior of the involved hardware components from an AUTOSAR (Website n.d.) specification to allow checking processor load distribution.	None	(1),(2),(3),(6)	(2)
`ToSchedulabilityAnalysis`	Build an equational algebraic representation of the dynamic behavior of the involved hardware and software components from an AUTOSAR specification to allow checking software response times.	None	(1),(2),(3),(6)	(2)
`ToDeploymentSimulation`	Build a DEVS representation of the deployment solution to allow checking latency times, deadlocks and lost messages.	None	(1),(2),(3),(6)	(2)

Table 4.8 – Model transformation examples from the Pwcs falling under the *Translation* intent

Nevertheless, our Catalog should be subject to a better classification based on characteristic properties, leading to an *Intent Taxonomy*, i.e. an organisation that emphasises shared similar qualities of intents.

To further illustrate this point, let us consider the best suitable intent of our thesis, namely *Translation*. It is related to four other intents: *Synthesis*, *Refinement*, *Analysis* and *Simulation* (cf. Tab. 4.3). All share the idea of *translating* input models' meaning into the target domain, but they do so in different ways. *Simulation* and *Analysis* can both be performed translationally, thus delegating their core task to a target dedicated engine; their translation requires using approximations, or abstractions, for the notions that do not natively exist in the target, with the idea to preserve the original semantics. On the contrary, *Synthesis* and *Refinement* follows a different translation scheme: here, preserving the semantics is also important, but it must be carried out with an extra constraint of putting more details, lowering the abstraction level during the process. Not surprisingly, all these intents have close characteristic property classes for ensuring their correctness, as demonstrated by comparing the contents of Tables 4.3 and 4.5.

We foresee the existence of intents *clusters* that regroup intents sharing close characteristic property classes for ensuring their correctness. Not surprisingly, *Synthesis*, *Refinement*, *Analysis* and *Simulation* form such a cluster, as witnessed by the careful examination of Tables 4.3 and 4.5, which can be extended with *Translational Semantics* and *Migration*. Identifying precisely such clusters is left for future work, but several obvious clusters can already be pointed from our Intent Catalog. For example, Model Visualisation could contain *Animation* and *Rendering*, regrouping intents with a target dedicated to visual elements; or Syntactic Manipulation, containing *Normalisation* and *Refactoring*, which regroups intents working at a syntactic level.

4.8.2 Composing Intents

Beyond its exhaustivity, the Intent Catalog raises the fundamental question of its *atomicity*: *can each intent be further decomposed into more fundamental intents?* The same question can be envisaged from the dual perspective of *combination closure*: *is it possible to create new intents by combining, in any possible way (sequence, composition, etc.), the current intents together?* Despite their theoretical nature, these questions have a direct impact on our work: by considering the best granularity level, it seems possible to have a property characterisation of intents that can scale up, and ultimately clarify properties for particular combinations of intents. As a simple example, consider two translations in sequence. It seems reasonable to consider the sequence as a translation itself, but depending on the nature of the end-target metamodel, the sequence can actually be better characterised as code generation, like modern compilers proceed when they need an intermediate representation (e.g., the Java source code is first compiled into bytecode, then compiled into platform-specific code).

This conceptual question has direct practical consequences from the certification viewpoint. The combination closure principle could help designers elaborate compositional verification techniques, just in the spirit of the corresponding research line in Computer-Aided Verification (cf. Cheung and Kramer 1999; de Roever et al. 2001; Graf, Steffen, and Lüttgen 1996 for an overview and a survey of the existing techniques for various applications): instead of certifying intents at a whole, the idea is to decompose the certification burden according to atomic sub-transformations and to elaborate results to exploit them for combined transformations.

Beyond the theoretical conceptualisation, we observed an engineering anchoring of this (de-)composition principle: in the PWCS, the PM offers a language support to *compose* transformations in order to consider them as a black box, just like procedures for imperative programming languages. For example, Figure 4.8 describes how the originally designed DSL models are first translated into Petri Nets, then combined to perform safety analysis. Although the entire box is qualified as *Analysis*, it is unsurprisingly organised into a sequence of transformations: the first is an *Abstraction* aiming at building what corresponds to Petri Net's state space; the second is the exhaustive state space exploration to check the satisfiability of safety formulæ. This (de-)composition principle also induces that a transformation intent is *context-sensitive*. For example, considered on its own, all transformations aiming at translating these DSL models into Petri Nets are simply *Translational Semantics*, whereas if considered in the context of their final goal with the subsequent box, this is rather a *Translation* with Analysis Delegation, since the purpose is to combine all DSL models to analyse them.

4.9 Related Work

Since we investigated model transformations intents and their relevant properties, we discuss three lines of contributions related to our work: intents in software engineering; classifications of model transformations and classifications of model transformation verification approaches.

4.9.1 Intents in Software Engineering

The notion of *intents* in software engineering is not new. In 1994, Yu and Mylopoulos (1994) realized that research in this area was, at the time, more focused on design and implementation of software—the *what* and the *how*—rather than on the requirements necessary to understand the software to improve the underlying development processes—the *why*. MDE is following a similar path: research has been more devoted to the different modelling and transformation activities rather than exploring the intents behind such activities.

Two studies (T. Kühne 2006; Muller, Fondement, et al. 2010) investigated the rationale (i.e., purpose or *intent*) behind modelling artifacts. T. Kühne (2006) identified two modelling intents based on the relationship between the modeled artifacts and their representative models: *token* models "project and translate" artifacts from the reality, and *type* models that additionally perform an "abstraction" step from the artifacts to represent universal aspects. Recently, Muller, Fondement, et al. (2010) explored the relationship between artifacts and their symbolic representations, using intention as a core constituent to the modelling activity. The intents discussed in the two former studies are amongst the intents presented in this paper, besides other additional intents that we investigate using our Intent/Property mapping (Section 4.5).

In the field of requirements models, requirements patterns have been proposed to facilitate requirements analysis (Withall 2007). Similar to transformation intents, requirements patterns are high-level descriptions of the properties that the implementation should possess. A key difference is that our notion of intents focuses on model transformations used in MDE, whereas requirements patterns have a much wider scope and are not tailored to the intricacies of a specific domain.

This Chapter extends a preliminary version of the Description Framework, proposed by Amrani et al. (2012a) with three major additions:

1. The Intents Catalogue summarises many of the intents discussed in the literature, and extends the one given by Amrani et al. (2012a), making it more precise (in particular for the *Translation*);
2. The relations between *Analysis*, *Translation* and *Simulation* are carefully handled in our Chapter, especially with the fact that a *Translation* can often be considered as an *enabler* to the other ones (what we called *delegation* in Section 4.5.3);
3. We refined our treatment of those intents, and explored two new ones: *Query* and *Refinement*.

4.9.2 Classifications of Model Transformations

Several studies (Czarnecki and Helsen 2006; Iacob, Steen, and Heerink 2008; Mens and Van Gorp 2006; Tisi et al. 2009; Visser 2005) proposed different classifications of model transformations based on different transformation aspects. Mens and Van Gorp (2006) provided a multidimensional taxonomy of transformations based on aspects related to the manipulated models (e.g., the abstraction level of the transformation's input and output models) and the used transformation execution strategies (e.g., in-place and out-place transformations). The classification dimensions are illustrated on transformations that can be grouped according to our intents. This Chapter investigated well-known *uses* of transformations, proposed fourteen additional intents to seven intents outlined by Mens and Van Gorp (2006), and discussed several intent properties. Iacob, Steen, and Heerink (2008) presented design patterns for model transformations expressed in QVT Relations, but the intents behind the transformations are not discussed.

Tisi et al. (2009) examined higher-order transformations, i.e., transformations manipulating transformations. They classified them based on whether their input and/or output models are transformations or not, resulting in four combinations: *synthesis* produces a transformation from a non-transformation; *analysis* takes an input transformation and produces a non-transformation output; *(de-)composition* uses multiple transformations both in input and output; and *modification* takes an input transformation and then produces a modified version of the input as an output. Our intents are more general in the sense that we do not distinguish between transformation and non-transformation models allowing for a wider applicability of the intent catalogue.

Czarnecki and Helsen (2006) classified the features of transformation languages by establishing a feature model. To do so, they introduced five intended applications of transformations which are also covered in our transformation intent catalogue.

A taxonomy of program transformations is presented by Visser (2005). Instead of proposing a taxonomy of multiple dimensions, like the one of Mens and Van Gorp (2006), Visser (2005) employed one discriminator for the taxonomy: out-place vs. in-place transformations (named as *translations* and *rephrasing*). Some of the leaf nodes in the taxonomy are program-specific, *e.g.,* (*de-*)*compilation*, *inlining*, and *desugaring*. Other nodes in the taxonomy are covered in our intent catalogue. Moreover, we present several intents that are specific to model transformations.

To sum up, our transformation intent catalogue is more comprehensive than previous attempts. Besides providing a name and an example of each intent, comprehensive meta-information (e.g., the use context, preconditions) and properties of interest for the given intent are proposed. To the best of our knowledge, the latter aspect has not been previously investigated.

4.9.3 Classifications of Model Transformation Verification Approaches

Several studies (Amrani et al. 2012b; Calegari and Szasz 2013; Gabmeyer, Brosch, and Seidl 2013; Rahim and Whittle 2013) proposed classifications of formal verification approaches of model transformations. Amrani et al. (2012b) (as well as Chapter 3) presented a tridimensional space for classifying transformation verification approaches, that have been also reused by Calegari and Szasz (2013) to derive a state-of-the-art of model transformation verification. Rahim and Whittle (2013) classified model transformation verification approaches with respect to the general approach used (e.g., testing, theorem proving, and model checking) and investigated the approaches with respect to the three dimensions of Chapter 3 (i.e., transformation language, verification property, and verification technique).

Gabmeyer, Brosch, and Seidl (2013) presented a feature model for the classification of verification approaches of software models that can be leveraged for the classifications of model transformation verification approaches by considering transformations as models. Our tridimensional approach corresponds on a general level to the main features of the feature model presented by Gabmeyer, Brosch, and Seidl (2013).

4.10 Summary

In this Chapter, we proposed a methodology aimed at facilitating the use of model transformations in industry in general, but also at paving the way to efficient development of verified transformations for safety-critical applications. This methodology adequately capture the *goal* and *requirements* of model transformation, and simplify the process of transformation specification, development, reuse, maintenance, validation and verification.

Our methodology contains at its core the *Description Framework*: it consists of a mapping between two new concepts, namely *transformation intents* and their *characteristic properties*. We proposed a preliminary Intents Catalogue of 21 intents, and presented in more details five of them: for each intent, we give at least one sample transformation from the literature possessing this intent. Although we do not claim for exhaustivity, this Catalogue encompasses the most frequently occuring intents that we found after a thorough literature review. We also provided a high level formalisation, at different abstraction levels, of the characteristic properties that were necessary for the illustrative five intents. The Description Framework is evaluated and extensively illustrated by applying these five intents to an industrial-scale Case Study describing a software for an automotive Power Window: this Case Study consists of more than 30 transformations whose intent has been identified using our Framework. We also extracted from this Case Study two witness transformations, on which we demonstrated concrete examples of properties.

Part II

Formal Specification of Kermeta

This Part defines a reference semantics for the Kermeta language, the core metamodelling and transformation language for the Kermeta platform.

The core Kermeta Language is decomposed into two sublanguages: the *Structural Language* (SL) allows modelers to specify metamodels and models in a way that respects the OMG standard MOF; whereas the *Action Language* (AL) allows transformation designers to define model transformations with object-oriented statements used to fill MOF's operation bodies.

It is not realistic to address the whole Kermeta language in such a work, since it covers many orthogonal features that do not directly impact the formal analysis of Kermeta transformations, but are rather user-oriented features aimed at simplifying and facilitating Kermeta specifications. As a consequence, the subset we address includes all model- and object-oriented features at the core of the AL.

Three Chapters naturally constitute this Part: Chapter 5 is an introduction to the Kermeta Platform; Chapter 6 captures the semantics of the Structural Language whereas Chapter 7 addresses the semantics of the Action Language.

The content of this Part was first published as two Technical Reports: one using a Z formalisation of the SL (Amrani and Amálio 2011), which helped deriving the full set-based mathematical formalisation of Chapter 6; then as a preliminary Report covering both languages (Amrani 2011). It was published as a double-round, double blind peer-reviewed Chapter of the Book *Formal And Practical Aspects of Domain-Specific Languages: Recent Developments* (Amrani 2013).

5 Kermeta in a Nutshell

For the purpose of the Formal Specification, this preliminary Chapter presents the Kermeta Language: it first describes how the Language was historically designed, and illustrates the Language's constructions on a small, yet representative running example, which also serves in the forthcoming chapters to illustrate the formal semantics. The last Section explains the mathematical notations at the basis of our formal definitions.

5.1 History

Soon after the emergence of MDE, the OMG started to propose standardised languages for the description of metamodels (called *metadata*at this time). The goal was to normalise metamodel representations and to facilitate data exchanges between applications and stakeholders: originally, MOF was perceived as a "*metadata management framework, and a set of metadata services to enable the development and interoperability of model and metadata-driven systems*" (Object Management Group 2006). However, the question of model transformation was still an open issue: as already seen in the previous Chapters, a plethora of model transformation languages are available, all with different characteristics and purposes.

Kermeta (originally typed *KerMeta* for KER*nel for* META*modelling*) emerged in 2006 in an effort to propose a complete metamodelling framework that complies with the following requirements:

- Providing an integrated framework where all metamodelling activities are centralised in a common infrastructure;
- Respecting the OMG standards, particularly in the domain of metamodelling specification and static semantics definition;
- Using an object-oriented action language for expressing model transformations

Using an object-oriented action language naturally fits EMOF's behavioural nature that nevertheless requires a careful choice of constructions for the action language to properly fit with EMOF's semantics. Figure 5.1 summarises the vision behind Kermeta, seen as the central, common kernel for metamodelling activities.

In order to respect the previous requirements, Kermeta is built on top of EMOF following a particular process using *weaving/promotion*, that aims at bootstrapping Kermeta in itself (Muller, Fleurey, and Jézéquel 2005). Figure 5.2 illustrates the process. Starting from EMOF, a metamodel for the Structural Language (noted SL) (i.e. EMOF itself, with special features specific to Kermeta) and the Action Language (noted AL) are defined, conforming to the standard EMOF (on the left of the Figure). Then, these languages are *weaved*, i.e. structurally merged, to create a new metamodel, denoted xEMOF in the Figure, that fully captures executable metamodels in Kermeta. However, this weaving only results in structural definitions: a *promotion*, aiming at both defining the semantics of both languages, and enabling further metamodels' instantiation, is required to create Kermeta's framework. The promotion, depicted by the grey arrow, makes xEMOF a meta-metamodel from which it is then possible to normally define, as represented on the right of the Figure, other metamodels like UML, or Kermeta's

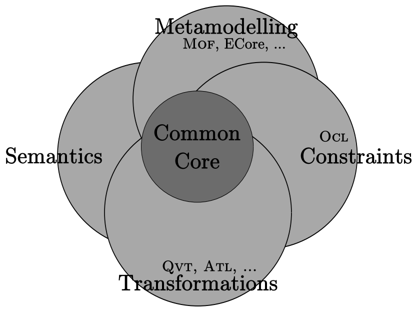


Figure 5.1 – Kermeta as a Kernel for different standardised MDE languages.

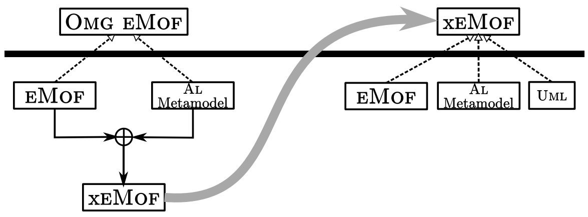


Figure 5.2 – Construction of Kermeta's Languages using weaving/promotion.

languages themselves. This *promotion* operation basically consists of defining the execution infrastructure for xEMOF, based on the weaving of both the structural and the action languages, in order to make things work properly in the next step.

The rest of this Chapter is organised accordingly to Kermeta's language organisation: Section 5.2 describes Kermeta's Structural Language (SL) whereas Section 5.3 describes Kermeta's Action Language (AL). Both are illustrated with a small DSL very familiar to computer scientists: the Finite State Machine (FSM). By choosing such a popular DSL, we assume the reader sufficiently familiar so we do not have to explain the DSL itself, but rather focus on how it is built.

5.2 Metamodelling: the Structural Language (SL)

Kermeta's SL is fully compatible with the OMG standardised structural language MOF, as described in Figure 5.2. This Section describes Kermeta's SL in detail, highlighting the key differences with MOF, and the restrictions we operated for the purpose of the formal specification, and illustrates its use with the FSM example, both in diagrammatic and textual representations.

5.2.1 SL Meta-metamodel

Figure 5.3 depicts the EMOF meta-metamodel as used in Kermeta: it describes how Kermeta metamodels are syntactically formed. A metamodel is a set of *Packages* that can contains nested *subpackages*. Each *Package* owns a set of *Types* that are either *DataTypes* or *Classes*. *DataTypes* naturally include usual *PrimitiveTypes* (booleans, integers, reals and strings) as well as *Enumerations*, constituted by *EnumerationLiterals* simply represented by their *name* (inherited from *NamedElement*). A *Class* also has a *name*, can be *abstract* and can have *super*classes, and contains *Features*. Since *Feature* inherits from *CollectionType*, it has a multiplicity (i.e. a *lower*- and *upper*-bound), and a type that can be arranged in collections (characterised by the uniqueness and ordering of their elements). A *Feature* has a *name*, and is either a *Property* or an *Operation*. A *Property* is either an *Attribute* (whose *type* is always a *DataType*), or a *Reference* (whose type is always a *Class*), that can be a *containment* and can possess an *opposite* reference. Just like *Classes*, an *Operation* can be *abstract*, has a (possibly empty) *Parameter* list and a *Body* (for non abstract *Operations*). As usual, a *Parameter* has a *name* and a *type*, except that this type can be combined with collections (since *Parameter* also inherits from *CollectionType*).

Since MOF is a refoundation of the core concepts present in UML, and UML was directly inspired from Object-Oriented Programming Languages, with a strong emphasis on Java, MOF unsurprisingly shares with Java several concepts: *Package*, *Enumeration*, *Class*, and *Operation* cover the same ideas than their Java equivalents. However, there exist some key differences:

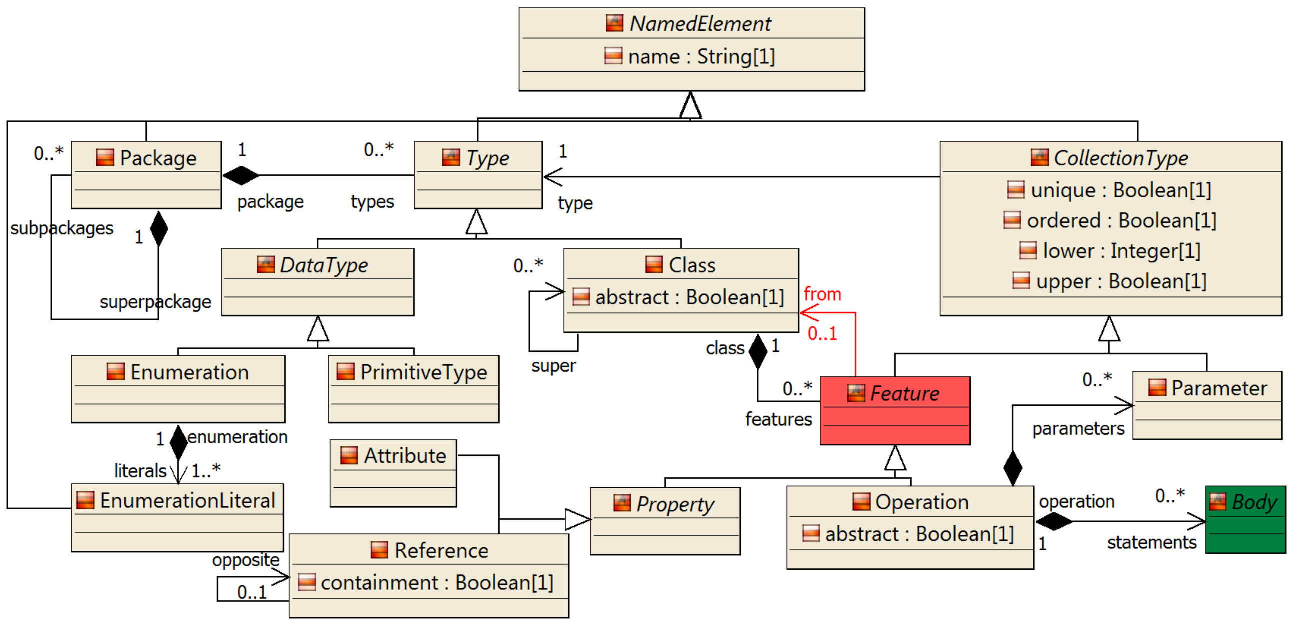


Figure 5.3 – Simplified Kermeta's meta-metamodel (Drey et al. 2009), compliant with the OMG EMOF standard (Object Management Group 2006). In green, the class *Body* used in Kermeta for enriching *Operation*s with a behaviour. In red, the class *Feature* and the reference *from* are added to Kermeta's original meta-metamodel to enable full type checking in our semantics specification of Chapter 6.

- MOF allows multiple inheritance, a feature that is present for Java only at the level of interfaces, but not for classes.

- The concept of *Property* replaces what is known in Java as *fields*, with two differences: no modifier (i.e. public, protected, and private Java keywords) can alter a property visibility; and the Java distinction between static and instance fields is irrelevant for MOF, since in models, all declared property in a class is automatically visible for each of its possible instance.

- In Java, the concept of collection is handled through the Collection API: it is ultimately defined as regular Java code instead of being a first-class concept as in MOF; and multiplicities have no equivalent in Java.

- Operation overloading, i.e. defining operations with the same name but different parameters, is fordidden in Kermeta: in the presence of multiple inheritance, it seriously complicates dynamic operation retrieving. Property redefinition does not makes sense for models, since any property is automatically inherited. However, operation overriding (or redefinition), i.e. redefining an operation's behaviour in subclasses, is possible like in Java, and is desirable for handling polymorphism.

Furthermore, if we notice that MOF concrete syntax, i.e. the way metamodels are physically represented, is largely inspired from UML, it becomes very easy to understand diagrammatic representations of metamodels in MOF. In fact, Figure 5.3 takes advantage of MOF's meta-circularity property to represent MOF's metamodel using its own diagramatic syntax. Kermeta can of course work with this representation (more precisely, the Eclipse Ecore implementation of EMOF, and UML class diagrams), but it also provides a purely textual representation: very close to Java's syntax, it is naturally understood by anyone with some experience in Java. This textual representation is aimed at simplifying model exchanges and management (since textual representations are easier to handle).

Figure 5.3 also shows the key differences between the standard EMOF and the meta-metamodel used in Kermeta. In green, an additional class *Body* is added to enable the weaving of the Action Language. In red, an

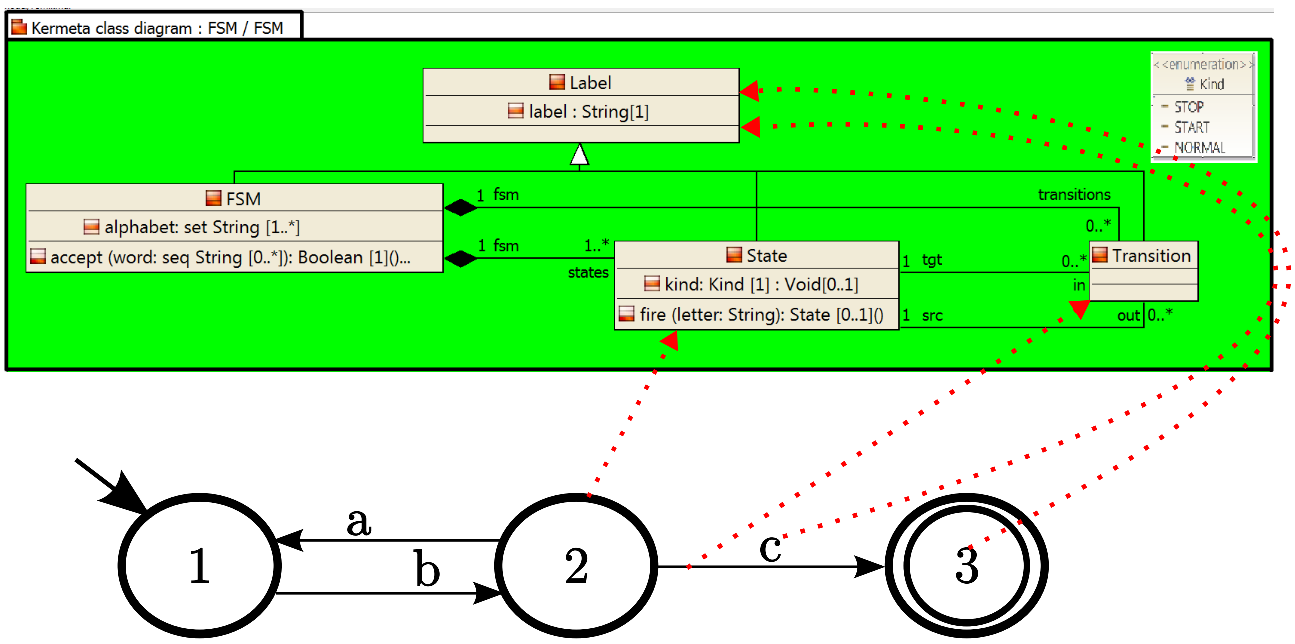

Kermeta class diagram : FSM / FSM
Label
label : String[1]
<<enumeration>>
Kind
STOP
START
NORMAL
FSM
alphabet: set String [1..*]
accept (word: seq String [0..*]): Boolean [1]()...
1 fsm
1 fsm
1..*
states
transitions
0..*
State
kind: Kind [1] : Void[0..1]
fire (letter: String): State [0..1]()
1 tgt
1 src
0..*
in
out
0..*
Transition
1
2
3
a
b
c

Figure 5.4 – The Finite State Machine (FSM) DSL in Kermeta

extra class *Feature* and its reference *from* are added for the purpose of our formal specification, for enabling full static type checking. Notice finally that we simplified the meta-metamodel by not including Kermeta's features that we do not formalise:

Genericity (also known as parametric classes), a notion not yet available in MOF, is a powerful paradigm for achieving concise structural description and reuse.

Model Type is a powerful extension that enriches MOF notion of type with the capacity of manipulating entire models as a type. When combined with operation parameters, this feature achieves a high-level of abstraction in the specification of behaviour.

Aspects are a convenient way of expressing cross-cutting concerns in a simple and uniform fashion, and for combining them properly. This feature introduces modularity in the metamodelisation process.

These features are already known from Object-Oriented Languages (with the notable exception of *model types*, very specific to models); however, their formal grounds are still open research questions (see for example Castagna and Xu 2011). Besides, sticking to the standard features present in MOF gives more perspective to our formalisation: it becomes more standard in the sense that it covers any MOF-like structural language (even UML with the same features).

5.2.2 The FSM Metamodel

Figure 5.4 depicts a possible metamodel and a simple model for FSMs. On top of the Figure, a package named FSM (in green) contains four classes named Label, FSM, State and Transition, and one enumeration named Kind. Each class inherits from Label (except Label itself): the inherited label attribute represents the FSM name, the state's names and the transition's label respectively. The FSM alphabet attribute, typed as a set of Strings, represents the possible actions for transitions (here, a, b and c). An FSM contains a non-empty set of States and of Transitions (accessible through the references states and transitions). A State has a kind that determines its nature: either a START or a STOP (i.e. final) state, or a NORMAL state; whereas a Transition is attached to a source and a target State (corresponding respectively to the src and tgt references). On the bottom of the Figure is represented a model with three states and three transitions, using the traditional visual notation for Fsms. Several red dashed arrows relate model elements to their metamodel class they are instance of (e.g., states, transitions and labels).

Below is the textual representation for the same metamodel. Since Kermeta's textual syntax has a lot in common with Java, we only comment on specific Kermeta constructions. Two constructions usually start any metamodel declaration: **require**, in Line 2, is used to import other declarations in a modular fashion; and **using** in Line 3 simplifies further type declarations by setting a default namespace to avoid typing fully qualified types (here, it is used for primitive types like integers and booleans). Kermeta distinguishes between an **attribute** and a **reference** by using two dedicated keywords. A reference with an opposite, like transition in Line 13, is denoted using a dash: here # fsm, declared in class Transition, is the opposite reference (which symmetrically has to declare the other as in Line 20). Type declaration is simplified in Kermeta: multiplicities are simply declared in the form [low..min]; and Kermeta introduces some keywords for collection types (for example in Line 13 and 14, one finds **set** and **seq** for respectively declaring sets and sequences). Creating the simple model of Figure 5.4 requires the use of the Action Language, explained in the next Section.

```
 1  package FSM;
      require kermeta
 3    using kermeta::standard

 5    enumeration Kind {NORMAL;START;STOP;}

 7    class Label{
        attribute label: String
 9    }

11    class FSM inherits Label{
        // FSM assumes there is only one START and one FINAL State
13      attribute alphabet: set String [1..*]
        reference states: seq State [1..*] # fsm
15      reference transitions: seq Transition [0..*]# fsm
      }
17
      class State inherits Label{
19      attribute kind: Kind
        reference fsm: FSM # states
21      reference in:  Transition [0..*] # tgt
        reference out: Transition [0..*] # src
23    }

25    class Transition inherits Label{
        reference fsm: FSM # transitions
27      reference tgt: State [1..1] # in
        reference src: State [1..1] # out
29    }
```

5.3 Transformations: the Action Language (Al)

As described in Figure 5.3, the Sl and the Al are connected at the level of the class *Operation*. However, weaving new behavioural elements within a meta-metamodel should be carefully done to ensure full static typing. We describe the Kermeta Al, comparing it with corresponding Java constructions. We then list our restrictions, before illustrating how to use the Al on our Fsm example.

5.3.1 Al Meta-metamodel

Figure 5.5 describes the Al, as described in Kermeta's Manual (Drey et al. 2009). Several classes from this metamodel play the exact same role as their Java counterparts. *Literal* describes literal values (e.g. for property or variable values). *Block*, *VariableDecl*, *Conditional* and *Loop* cover the usual imperative constructions for blocks, variable declarations, conditional and iterative statements, as in Java; *Self* is Kermeta equivalent for **this** in Java. *Raise* and *Rescue* handle exceptions (corresponding to throw and **catch** in Java). The class

Assignment covers two Java constructions, namely assignment and casting. *CallExpr* is an umbrella class for all calls, i.e. an element whose value is needed: a *CallVar* stands for example for a variable on the right-hand side of an assignment; *CallResult* is equivalent to **return** in Java (Kermeta uses a special variable **result** for storing operation values); *CallFeature* enables navigation through class properties and operation calls; and *CallSuperOp* allows to call an operation defined in superclasses. Three classes denote constructions specific to Kermeta with no equivalent in Java for now: *LambdaParameter* and *LambdaExpression* enable the use of functionals[1], just like in OCL (e.g., a -> select(...), and *TypeReference* allows to use model types.

We already restricted Kermeta SL by not considering genericity, aspects and model types. We additionally do not address exception handling, since it is somehow orthogonal to DSL behavioural specification. Summarising, the following constructions will be ignored: *LambdaParameter* and *LambdaExpression*; *TypeReference*; and finally *Raise* and *Rescue*.

5.3.2 The FSM Model & Behaviour

Creating a model is easy, since Kermeta uses an instance creation statement very close to Java. The only difficult part here is to set model properties properly to represent the desired model instance.

```
// Declare and initialise (with ‘‘empty’’) the FSM Model
var abc: FSM init FSM.new
// Declare and initialise State s1 as START
var s1: State init State.new
s1.kind := Kind.START
// Initialise references states and transitions
abc.states.add(s1)
// Other states here
// Declare and initialise Transition ta as START
var ta: Transition init Transition.new
abc.transitions.add(t1)
// Other transitions tb, tc
var outS1 : Transition [0..*] init OrderedSet<Transition>.new
outS1.add(ta)
// same for tb, tc
s1.out.addAll(outS1)
```

The behavioural semantics of an FSM closely follows the classical semantics. An *FSM* is said to *accept* a *word* if, by successively firing transitions from a current state, it ends in a final state when the word is entirely consumed. A *State fires* when there exists an outgoing *Transition* with the adequate *label* as the current *letter* of the *word*. We reproduce here the corresponding operation bodies.

```
package FSM;
   // Previous code here...

   class FSM inherits Label{

      // ...

      operation getStart(): State is do
         var i : Integer init 0
         from i := 0
            until i == states.size() or states.at(i).kind == Kind.START
         loop
            i := i+1
         end
         if i == states.size() then
            result := void
         else
            result := self.states.at(i)
         end
```

[1]At the time we are writing (September 2013), so-called *lambda expressions*, or *closures*, have been integrated in Java 8, whose public release in scheduled for March 2014. Scala, another object-oriented programming language, already offers such features in an integrated fashion; this is one of the reasons Kermeta's interpreter engine has been reimplemented in Scala.

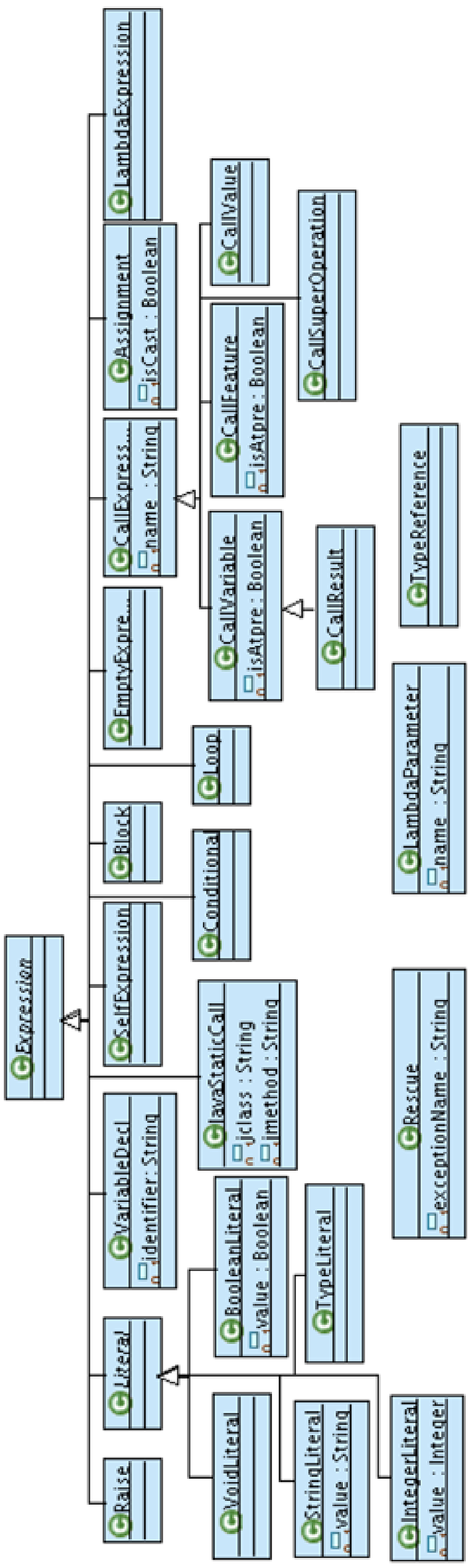


Figure 5.5 – Kermeta's Action Language (from (Drey et al. 2009, §3.3))

```
   end

   operation getFinal(): State is do
      var i : Integer init 0
      from i := 0
         until i == states.size() or states.at(i).kind == Kind.STOP
      loop
         i := i+1
      end
      if i == states.size() then
         result := void
      else
         result := self.states.at(i)
      end
   end

   operation accept(word: seq String [0..*]) : Boolean is do
      var current: State init self.getStart()
      var final  : State init self.getFinal()
      var toEval : seq String[0..*] init word
      var isNull : Boolean init false

      from var i : Integer init 0
         until i == toEval.size() or isNull //((not toEval.isEmpty()) and (not isNull))
      loop
         current := current.fire(toEval.at(i))
         if(current.isVoid) then
            isNull := true
         end
         i := i+1
      end
      result := (current == final)
   end
}

class State inherits Label{

   // ...

   operation fire(letter: String): State [0..1] is do
      var trans: seq Transition [0..*] init self.out.asSequence()

      if(trans.isVoid) then
         // no output transitions: return what is suitable for no state (card = 0)
         result := void
      else
         var current: Transition init trans.at(0) // head only reads the head, do not modify the sequence

         from var i : Integer init 0
            until i == trans.size() or trans.at(i).label == letter//(current.isVoid) or (current.label ==
                 letter)
         loop
            i := i+1
         end
         if(current.isVoid) then
            result:= void
         else
            result:= current.tgt
         end
      end
   end
}
```

5.4 Mathematical Background

The sign $\triangleq$ defines a set either by extension or by intension. The set of booleans is noted $\mathbb{B} \triangleq \{\top, \bot\}$ for truth values true and false, respectively. The set of naturals and integers are noted $\mathbb{N}$ and $\mathbb{Z}$ respectively; and we define $\mathbb{N}^\star \triangleq \mathbb{N}\backslash\{0\}$; we note $\mathbb{R}$ and $\mathbb{S}$ the sets of real numbers and strings, respectively. Let S, S', S'', S''' be sets. The notation $S_\perp$ stands for $S \cup \{\perp\}$, with $S \cap \perp = \varnothing$. The size of S is noted $|S|$.

5.4.1 Functions

Since all the mathematical framework is based on set theory and makes extensive use of functions, we explicit here the notations used.

Total, resp. partial, functions are noted $f\colon S \to S'$ and $f'\colon S \nrightarrow S'$; their domain is noted $\mathsf{Dom}\,(\cdot)$ and their range $\mathsf{Ran}\,(\cdot)$. Partial functions are right-associative: $g\colon S \nrightarrow S' \nrightarrow S'' \nrightarrow S'''$ means $g\colon S \nrightarrow (S' \nrightarrow (S'' \nrightarrow S'''))$ and we abbreviate $((g(s))(s'))(s'')$ into $g(s)(s')(s'')$, or sometimes write $g_s^{s'}(s'')$ for short. Substituting f by $(s_1, s_2) \in S \times S'$, noted $f[s_1 \mapsto s_2]$, results in a function f' where forall $s \neq s_1$, $f'(s) = f(s)$ and $f'(s_1) = (s_2)$. If $h\colon S \longrightarrow S' \times S''$, we sometimes write $h(s) = (\_, s'')$ or $h(s) = (s', \_)$ when the first or the second element is not relevant in context, and use the same notation for substitution.

Example 5.1 (A Simple Function)**.** Suppose a function f defined as follows (where $|\cdot|$ denotes the length of a string):

$$\begin{array}{rcl} f\colon \mathbb{S} & \nrightarrow & (\mathbb{R} \nrightarrow \mathbb{R}) \\ \mathsf{s} & \mapsto & g_{\mathsf{s}} \end{array} \qquad\qquad \begin{array}{rcl} g_{\mathsf{s}}\colon \mathbb{R} & \nrightarrow & \mathbb{R} \\ x & \mapsto & \frac{|\mathsf{s}|}{x} \end{array}$$

Suppose that for some reason, we would like to restrict f to the strings representing week days (encoded on three characters): we then have $\mathsf{Dom}\,(f) = \{\mathsf{mon}, \cdots, \mathsf{sun}\} \cup \{\epsilon\}$. Similarly, we also have for all $\mathsf{s} \in \mathsf{Dom}\,(f)$ that $\mathsf{Dom}\,(f(\mathsf{s})) = \mathsf{Dom}\,(g_{\mathsf{s}}) = \mathbb{R}^\star$, since a fraction is not defined for zero as a denominator. In fact, the image of a string by f is actually a (partial) function itself, as suggests the definition of f on the left. Since $\mathsf{aaa} \notin \mathsf{Dom}\,(f)$, aaa has no image, but mon does: for example, $(f(\mathsf{mon}))(3)$, preferably noted $f(\mathsf{mon})(3)$, is equal to 1. We could also substitute the effect of one value with a special function. For example, $f[\mathsf{mon} \mapsto \mathbf{0}_\mathbb{R}]$ changed the image for mon to the $\mathbf{0}_\mathbb{R}$ function on real (i.e. the constant function that associates 0 to any real), while keeping the old definition of other week days. □

5.4.2 Abstract Datatypes Specifications

This Section introduces notations and formal counterparts (H. Ehrig and Mahr 1985; Spivey 1992) for the classical abstract datatypes useful for representing several collections of values, which is a central part of our semantic domains. Classical collection kinds common in MDE are *bags*, *sets*, *sequences* and *ordered sets*[2] and are considered well-formed, i.e. collections containing values of the same type.

Let C be a collection of values of type T, and $t, t' \in \mathsf{T}$ some values. We suppose predefined functions over collections: $\sharp(\mathsf{C})$ returns the number of elements in C and $\mathrm{type}(\mathsf{C})$ returns the type of C, i.e. T. Adding and removing an element t from C is uniformly noted $\mathsf{t} \oplus \mathsf{C}$ and $\mathsf{t} \ominus \mathsf{C}$ respectively, whose effect is properly defined below for each collection kind.

Definition 5.1 (Bag Collection)**.** *A* bag *(or* multiset*) of values in* T*, represents a collection where an element can be repeated many times (in contrast to sets) and is noted* $[\mathsf{T}]$*.*

$$[\mathsf{T}] \triangleq \{\mathsf{T} \longrightarrow \mathbb{N}^\star\}$$

[2]The term usually used in MDE is *ordered set*, which is quite confusing: in set theory, an ordered set is a set equipped with an order *on* the element (i.e. a binary relation); whereas in the MDE vocabulary, an ordered set is a set where the elements are stored, or are accessible, *in* a certain order. In order words, the first is naturally not concerned with the computational considerations of the second. In this Background Section, we will prefer a more precise denomination: "*sequence with unique representative*".

A bag $\{\mathsf{t}_1 \mapsto n_1, \ldots, \mathsf{t}_\mathsf{n} \mapsto n_n\}$ is preferably written $[\mathsf{t}_1, \ldots, \mathsf{t}_\mathsf{n}]$, where each element t_i appears n_i times in the list $\mathsf{t}_1, \ldots, \mathsf{t}_\mathsf{n}$. The empty bag $[\ ]$ is a shortcut for the empty function from T to $\mathbb{N}^\star$. Let $\mathsf{B}, \mathsf{B}' \in [\mathsf{T}]$. We note $\mathsf{t}\sharp\mathsf{B}$ the number of times t appears in B; and use the usual set notations for bag membership and subbaging: $\mathsf{t} \in \mathsf{B}$ holds if t appears in B at least one time; and $\mathsf{B} \subseteq \mathsf{B}'$ holds if all elements appearing in B appear in B' at least the same number of times. Adding an element in B, noted $\mathsf{t} \oplus \mathsf{B}$ (or removing it, noted $\mathsf{t} \ominus \mathsf{B}$) changes the number t is repeated, or adds it if it was not there (resp. removes if it appeared only once).

Definition 5.2 (Set Collection)**.** *A set collection is a collection of elements that are not repeated, noted* $\wp(\mathsf{T})$[3].

Since this notion coincide with the *subset* notion in set theory, it does not need more explanation. The uniform notations for adding $\oplus$ and removing $\ominus$ an element from a set are synonyms of the usual set operators union $\cup$ and difference $\setminus$.

Definition 5.3 (Sequence collections)**.** *A* sequence *(also called* list*) is a collection where the order of the elements matters. We distinguish between sequences allowing multiply repeated values, noted* $\langle\mathsf{T}\rangle$ *and sequences forbidding repetitions, i.e. with a unique representative of each element (also called* ordered sets*), noted* $\langle\!\langle\mathsf{T}\rangle\!\rangle$.

$$\begin{aligned}\langle\mathsf{T}\rangle &\triangleq \{f\colon \mathbb{N} \nrightarrow \mathsf{T} \mid \mathsf{Dom}\,(f) = \{1..\sharp f\}\}\\ \langle\!\langle\mathsf{T}\rangle\!\rangle &\triangleq \{f\colon \mathbb{N} \nrightarrow \mathsf{T} \mid \mathsf{Dom}\,(f) = \{1..\sharp f\} \wedge f \text{ injective}\}\end{aligned}$$

A sequence $\{1 \mapsto \mathsf{t}_1, \ldots, n \mapsto \mathsf{t}_\mathsf{n}\}$ is preferably noted $\langle\mathsf{t}_1, \ldots, \mathsf{t}_\mathsf{n}\rangle$ or $\langle\!\langle\mathsf{t}_1, \ldots, \mathsf{t}_\mathsf{n}\rangle\!\rangle$. The empty sequence $\langle\ \rangle$ and $\langle\!\langle\ \rangle\!\rangle$ are shortcuts for the empty function from $\mathbb{N}$ to T. Let $\mathsf{S}, \mathsf{S}' \in \langle\mathsf{T}\rangle$ be two sequences. Adding t on top of S is noted $\mathsf{t} \oplus \mathsf{S}$; removing one occurrence of t in S is noted $\mathsf{t} \ominus \mathsf{S}$. Concatenating two lists, noted $\mathsf{S} \circ \mathsf{S}'$, results in a sequence containing the elements of S followed by those of S'. If S is non-empty, it can be decomposed in $\mathsf{S} = \mathsf{t}_1 :: \mathsf{S}'$, where t is the element at the *head* and S' is the rest, or the *tail*, of the sequence.

Collections possess two orthogonal dimensions: *uniqueness*, i.e. whether a collection admits an element multiply or not; and *ordering*, i.e. whether the order of these elements matter or not. If operations to remove repetition and choose an arbitrary order between unordered elements are available, it is possible to convert any collection to any other one. The following definition introduces two operators that convert collections.

Definition 5.4 (Downward/Upward Conversions)**.** *Let* $\mathsf{Collection} \triangleq \{\mathsf{Bag}, \mathsf{Set}, \mathsf{List}, \mathsf{OSet}\}$*. The family of* downward conversion *operators* ($\bigtriangledown_\mathsf{c}$) *and* upward conversion *operators* ($\bigtriangleup_\mathsf{c}$)*, indexed by* $\mathsf{c} \in \mathsf{Collection}$*, convert collections adequately by imposing an arbitrary order or removing repetitions of elements.*

Example 5.2 (A Simple Function)**.** Let $\mathsf{B} = [1, 2, 5, 1, 3, 2, 9] \in [\mathsf{Integer}]$ be a bag of integers. Then $\bigtriangledown_\mathsf{List}(\mathsf{B})$ results in a list $\mathsf{L} \in \langle\mathsf{T}\rangle$ with all elements of B in an arbitrary order: for example, in the order of the presentation, $\mathsf{L} = \langle 1, 2, 5, 1, 3, 2, 9\rangle$. Similarly, $\bigtriangledown_\mathsf{Set}(\mathsf{B})$ results in the following set collection $\mathsf{S} = \{1, 2, 5, 3, 9\} \in \wp(\mathsf{T})$, whereas $\bigtriangledown_\mathsf{OSet}(\mathsf{B})$, with the natural order over natural, results in the ordered set $\mathsf{O} = \{1, 2, 3, 5, 9\} \in \langle\!\langle\mathsf{T}\rangle\!\rangle$. Creating a bag from O is done by applying the adequate upward convertor $\bigtriangleup_\mathsf{Bag}(\mathsf{O})$, which results in the bag $\mathsf{B}' = [1, 2, 3, 5, 9]$. This proves that conversion operators are neither commutative, nor associative. □

[3] We use the so-called "Weierstraß p" $\wp$ symbol to denote powerset instead of the $\mathbb{P}$ symbol as found in Z, because it is already used as a set symbol later.

6 Structural Language

This Chapter formalises the Structural Language (SL) of Kermeta considered in this work: in a nutshell, the SL is a conservative extension of MOF (in its 2.0 version, i.e. without any consideration about genericity) allowing full static typing. The considered extension enables the Action Language to take full advantage of these structures. The formalisation proposed in this Chapter applies directly to any MOF-like language, therefore constituting a canonical semantics for MOF-like structural languages that use the same concepts: packages, enumerations, classes with attributes, references with multiplicities and uniqueness/orderness, and operations.

6.1 Structural Semantics Overview

The Kermeta SL is an conservative extension of the OMG standard MOF. We choose to simplify this language by not considering some of the already existing Kermeta constructions: genericity, model typing and aspects. By isolating this subset of Kermeta's AL, we lose powerful expressive constructions, but gain the fact that the formalisation goes beyond Kermeta: in fact, it works well for any MOF-like SL, even for UML Class Diagrams with the same features.

The MOF Specification (Object Management Group 2006) comprises two parts: EMOF (or Essential MOF) and CMOF (or Complete MOF). EMOF is in fact a meta-metamodel: it is a model that describes a language for defining models. CMOF extends EMOF by explicitly defining what the OMG calls the "CMOF *abstract semantics*" (Object Management Group 2006, §15), i.e. a language (always as a model) that describes what is represented by models.

In Kermeta, things are slightly different. Creating metamodels can be achieved in two ways: either *visually*, by importing a diagrammatic metamodel from another formalism (e.g. UML or ECORE) or by using the dedicated editor; or *textually*, by using the classical Eclipse editor.

For the sake of generality, we explain our formal specification of Kermeta's SL based on the visual representation of EMOF metamodel. The Figure 6.1 describes our approach. After having described the core artifacts necessary to mathematically specify metamodels and models, we define two sets:

- $\mathcal{M}$ is the set of all Kermeta metamodels that can therefore be seen as a mathematical representation of the EMOF meta-metamodel.
- $\mathbb{M}$ is the set of all Kermeta models by only capturing the syntax of models.

Of course, when a metamodel designer chooses a particular metamodel $\mathsf{MM} \in \mathcal{M}$, it induces a set of models $\mathsf{M}_1, \ldots, \mathsf{M}_n, \ldots \in \mathbb{M}$ that are *valid* regarding MM (notice that the set of induced models is generally infinite). This notion of validity is traditionally called *conformance* in the MDE community: by defining a model $\mathsf{M} \in \mathbb{M}$, one has to ensure that M is actually one of the models induced by MM, i.e. $\exists i \cdot \mathsf{M} = \mathsf{M}_i$. This relation is noted $\mathsf{M} \blacktriangleright \mathsf{MM}$.

The rest of this Chapter is organised as follows: the next Section defines the core artifacts on which rely the whole formalisation: *names*, *types* and *values*. Then, we proceed as explained: Section 6.3 formalises MOF

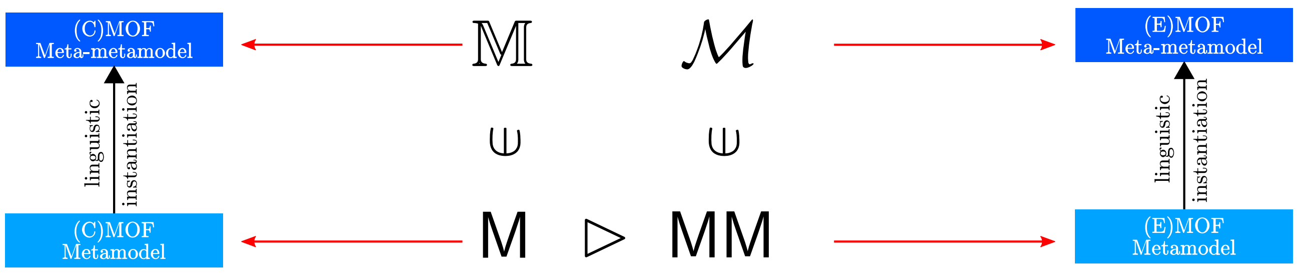


Figure 6.1 – Overview for the construction of the SL semantics (from Section 2.1).

metamodels; Section 6.4 formalises models; and finally Section 6.5 relates models to metamodels by expressing their conformance. All definitions are illustrated using the FSM example presented in Chapter 5; the full formalisation, with all components, is available in Appendix A. The Chapter ends by proposing a discussion and reviewing some related work in Section 6.6.2.

6.2 Names, Types and Values

This Section introduces basic mathematical constructions used all over the formalisation, namely *names*, *(syntactic) types* and *(semantic) values*. These constructions are extracted either from the metamodel itself, providing a basic interpretation of core artifacts in the MOF metamodel, or from the instance model described as a semantic domain for MOF in the MOF Specification Document (Object Management Group 2006).

6.2.1 Names

In MOF, every concrete class inheriting from the *NamedElement* class should have a name[1]. The following Definition formally defines these concrete elements and associated names.

Definition 6.1 (Elements — Names)**.** *The set* Element *is the set of concrete* MOF *artifacts. The set* Name *represents the names for any* MOF *artifact.*

$$\begin{aligned} \mathsf{Element} &\triangleq \{\mathsf{Pkg}, \mathsf{Class}, \mathsf{Enum}, \mathsf{Attr}, \mathsf{Ref}, \mathsf{Op}, \mathsf{Param}\} \\ \mathsf{Name} &\triangleq (\mathsf{Name}_e)_{e \in \mathsf{Element}} \end{aligned}$$

To avoid the subscripted notation, we introduce a more compact notation for referring to names: for example, the set of class names $\mathsf{Name}_{\mathsf{Class}}$ will be noted ClassN.

Notice that because Name is sorted, it can represent adequately ontologically different artifacts with the same name. In the FSM example, the package named FSM contains a class also named FSM, which is valid in MOF. These is represented as follows: $\mathsf{FSM} \in \mathsf{Name}_{\mathsf{Package}}$ and $\mathsf{FSM} \in \mathsf{Name}_{\mathsf{Class}}$.

Furthermore, we require that package names in a metamodel are unique, since they are the entry point for accessing other metamodel components. To achieve this requirement, we flatten package names as follows: suppose a package P contains another package PP, then these packages will be represented with the following names: $\mathsf{P}, \mathsf{P} :: \mathsf{PP} \in \mathsf{PkgN}$, respectively (or by using any other separator distinct from valid name characters). Consequently, the extraction of names from a metamodel represented as a diagram is straightforward. From now on, unless clearly specified, we will not distinguish between an element and its name in a metamodel.

[1]For a discussion on how MOF actually deals with names, element identifiers and constraints over names in a metamodel, the reader can refer to Sections §10, §12.4 and §12.5 of the Specification Document (Object Management Group 2006). The least we can say is that names and identifiers are intricately defined and should receive proper attention. We choose to simplify the notion of identifier, i.e. how to uniquely refer to a metamodel element, by only considering names. Furthemore, this approach fits well with the way Kermeta's Action Language is defined.

Finally, the notions of *property* (i.e. either attribute or reference) and *feature* (i.e. either property or operation) names are introduced by following MOF definitions.

$$\begin{array}{rcl} \mathsf{PropN} & \triangleq & \mathsf{AttrN} \cup \mathsf{RefN} \\ \mathsf{FeatN} & \triangleq & \mathsf{PropN} \cup \mathsf{OpN} \end{array}$$

Example 6.1 (FSM names)**.** The main package name would be represented as $\mathsf{FSM} \in \mathsf{PkgN}$. Class names correspond to the following: $\mathsf{Label}, \mathsf{FSM}, \mathsf{State}, \mathsf{Transition} \in \mathsf{ClassN}$. Names for other artifacts are easily inferred. Notice here how sorts help distinguishing between the package and the class with the same name. □

6.2.2 Syntactic Types

MOF defines syntactic types that can be used at the metamodeling level: they correspond to the abstract class *Type* in MOF and Kermeta (cf. Fig. 5.3).

A type (sometimes called *basic* type) is either a *primitive type*, a *class name* or an *enumeration*, both declared in the scope of a given package. Each type is combined with two other pieces of information: a *collection kind* and a multiplicity, which constitutes our notion of *syntactic type* MType.

Definition 6.2 (Syntactic Types)**.** *The set of* primitive types *PrimType is one of the classical built-in type names. A* (basic) type *(name) Type is either a primitive type, or a class name or an enumeration name declared in the scope of a package. A* collection kind *is one of the usual collection encountered in metamodeling, namely* bag, list, set *or* ordered set[2]*, or no collection, which is denoted by* $\perp$*. A* collection type *is a pair of a collection and a type. A* multiplicity type *is a pair of a multiplicity, given through its lower and upper bounds, and a collection type. The sets PrimType and Collection are equipped with the usual order (denoted respectively* $\preccurlyeq_{\mathsf{Prim}}$ *and* $\preccurlyeq_{\mathsf{Coll}}$ *represented by the Hasse diagram in the right.*

$$\begin{array}{rcl} \mathsf{MType} & \triangleq & (\mathbb{N} \times \mathbb{N}_\star) \times \mathsf{CType} \\ \mathsf{CType} & \triangleq & \mathsf{Collection} \times \mathsf{Type} \\ \mathsf{Collection} & \triangleq & \{\mathsf{Bag}, \mathsf{List}, \mathsf{Set}, \mathsf{OSet}, \perp\} \\ \mathsf{Type} & \triangleq & \mathsf{PClassN} \cup \mathsf{DataType} \\ \mathsf{PrimType} & \triangleq & \{\mathsf{Boolean}, \mathsf{Integer}, \mathsf{Real}, \mathsf{String}\} \end{array}$$

$$\begin{tikzcd} & & & & \text{Bag} & \\ & \text{Real} & & \text{Set} \arrow[ur] & & \text{List} \arrow[ul] \\ \text{Boolean} & \text{Integer} \arrow[u] & \text{String} & & \text{OSet} \arrow[ul] \arrow[ur] & \end{tikzcd}$$

In the previous definition, we introduced a shortcut for class or enumeration declared in the scope of a package as following:

$$\begin{array}{rcl} \mathsf{DataType} & \triangleq & \mathsf{PEnumN} \cup \mathsf{PrimType} \\ \mathsf{PClassN} & \triangleq & \mathsf{PkgN} \times \mathsf{ClassN} \\ \mathsf{PEnumN} & \triangleq & \mathsf{PkgN} \times \mathsf{EnumN} \end{array}$$

We define $\mathbb{N}_\star \triangleq \mathbb{N} \cup \{\star\}$ to reflect the usual notation for MOF upperbounds, together with the associated order relation $\leqslant_\star$ defined by $n \leqslant_\star n' \overset{\triangle}{\iff} (n' = \star) \vee (n \leqslant n')$.

Let $\mathsf{mt} = ((\mathsf{low}, \mathsf{up}), (\mathsf{C}, \mathsf{t})) \in \mathsf{MType}$ be a multiplicity type. To lighten the notation, we will note $\mathsf{mt} = (\mathsf{low}, \mathsf{up}, \mathsf{C}, \mathsf{t})$. Furthemore, mt is *valid* iff

$$(\mathsf{low} \leqslant_\star \mathsf{up}) \wedge (\mathsf{C} \neq \perp \implies 1 \leqslant_\star \mathsf{up})$$

[2]Note here that the traditionally used denomination "ordered set" is confusing: in the MDE area, it denotes a set where the *order of elements* matters, whereas in general it denotes a set equipped with an *order over the elements.*

Example 6.2 (Fsm Types)**.** There is only one enumeration named $\mathsf{Kind} \in \mathsf{EnumN}$, and therefore $(\mathsf{FSM}, \mathsf{Kind}) \in \mathsf{PEnumN}$. The set $\mathsf{C}_{\mathsf{FSM}}$ of all class names is $\mathsf{C}_{\mathsf{FSM}} = \{(\mathsf{FSM}, \mathsf{Label}), (\mathsf{FSM}, \mathsf{FSM}), (\mathsf{FSM}, \mathsf{State}), (\mathsf{FSM}, \mathsf{Transition})\} \subseteq \mathsf{PClassN}$.

Focusing on the class FSM, the attribute alphabet has a collection type $(\mathsf{Set}, \mathsf{String}) \in \mathsf{CType}$ and a multiple type $((1, \star), (\mathsf{Set}, \mathsf{String})) \in \mathsf{MType}$. Similarly, the reference transitions as a multiple type $((0, \star), (\mathsf{Set}, \mathsf{Transition})) \in \mathsf{MType}$. Note that we will always flatten multiple types, e.g. for transitions, we note $(0, \star, \mathsf{Set}, \mathsf{Transition}) \in \mathsf{MType}$. □

6.2.3 Semantic Values

The set of (semantic) values $\mathbb{V}$ constitutes the core semantic domain for models. It roughly follows the under-specified Omg Mof definitions (cf. (Object Management Group 2006, §15.2), which is part of cMof). The definition follows the structural construction of syntactic types, to which they are formally related later in Def. 6.6.

Definition 6.3 ((Semantic) values)**.** *The set* $\mathbb{V}$ *of (semantic) values is the (disjoint) union of* basic sets of values, *namely the sets of* booleans $\mathbb{B}$, integers $\mathbb{N}$, reals $\mathbb{R}$, strings $\mathbb{S}$, enumeration literals $\mathbb{E}$ *and* objects *(also called instances)* $\mathbb{O}$, *with the* bags, (sub)sets, sequences *and* ordered sets *of these basic sets.*

$$\begin{array}{lccccccccccc}
\mathbb{V} \triangleq & \mathbb{B} & \uplus & \mathbb{Z} & \uplus & \mathbb{R} & \uplus & \mathbb{S} & \uplus & \mathbb{E} & \uplus & \mathbb{O} \\
& [\mathbb{B}] & \uplus & [\mathbb{Z}] & \uplus & [\mathbb{R}] & \uplus & [\mathbb{S}] & \uplus & [\mathbb{E}] & \uplus & [\mathbb{O}] \\
& \wp(\mathbb{B}) & \uplus & \wp(\mathbb{Z}) & \uplus & \wp(\mathbb{R}) & \uplus & \wp(\mathbb{S}) & \uplus & \wp(\mathbb{E}) & \uplus & \wp(\mathbb{O}) \\
& \langle\mathbb{B}\rangle & \uplus & \langle\mathbb{Z}\rangle & \uplus & \langle\mathbb{R}\rangle & \uplus & \langle\mathbb{S}\rangle & \uplus & \langle\mathbb{E}\rangle & \uplus & \langle\mathbb{O}\rangle \\
& \langle\!\langle\mathbb{B}\rangle\!\rangle & \uplus & \langle\!\langle\mathbb{Z}\rangle\!\rangle & \uplus & \langle\!\langle\mathbb{R}\rangle\!\rangle & \uplus & \langle\!\langle\mathbb{S}\rangle\!\rangle & \uplus & \langle\!\langle\mathbb{E}\rangle\!\rangle & \uplus & \langle\!\langle\mathbb{O}\rangle\!\rangle
\end{array}$$

This construction, although complicated, does not admit any ill-formed collection of values, i.e. collections with values of different sets (for example, a list with one integer and one boolean). The *size* of a value $|\bullet| : \mathbb{V} \to \mathbb{N}$ is defined by extension from Sec. 5.4.2. The type of a value will be defined later, when formal definitions of metamodels and models will be available.

Example 6.3 (Fsm Model Values)**.** Due to the chosen concrete syntax, some values in the model are explicit, whereas others are not. For example, the labels corresponding to states are “1”, “2” and “3”, which are strings (denoted here between “ ·”); but the input arrow of state “1”, denoting that it is start state, says that attribute kind in class $(\mathsf{FSM}, \mathsf{State})$ is set to the value $\mathsf{START} \in \mathbb{E}$, which corresponds to its declaration. □

6.3 Metamodels

A metamodel basically consists of declarations: Mof elements (through their name) are bound to (syntactic) types and other information, w.r.t. a particular topology defined by Mof. This Section defines several sets of functions capturing the structure of these declarations. Of course, all these declarations make sense when put together to define the metamodel itself.

This Section proceeds as follows: for each Mof artifact, an informal description based on the metamodel representation in Fig. 5.3 is provided, from which a formal definition is derived by providing adequate mathematical structures in order to create a set of functions formalising the idea of this artifact. As we said before, the formalisation only addresses the *concrete* classes of the meta-metamodel (namely, the classes corresponding to *packages*, *enumerations*, *classes*, *properties* and *operations*), which are actually instanciated in a particular metamodel specification.

6.3.1 Package

From the metamodel in Fig. 5.3, the class *Package* has two references: *subpackages* (with its opposite *superpackage*) allows a package to eventually contain subpackages; and *types* (with its opposite *package*) allows a package to eventually contains types, i.e. either enumerations or classes.

The set $\mathcal{P}$ of *package functions* represents the *packages declarations*: such a function $p \in \mathcal{P}$ maps packages that are part of a metamodel to their subpackages, and nested classes and enumerations; and p is *correct* if no package is contained in itself[3].

$$\mathcal{P} \triangleq \{p \;:\; \mathsf{PkgN} \nrightarrow \wp(\mathsf{PkgN}) \times \wp(\mathsf{ClassN}) \times \wp(\mathsf{EnumN}) \mid \forall \mathsf{pkg}, p(\mathsf{pkg}) = (P, C, E) \Rightarrow \mathsf{pkg} \notin P\}$$

Notice here why we required that package names need to be unique throughout a metamodel: $\mathcal{P}$, as well as the following definitions, is a function from package names, which implies to be able to uniquely access any metamodel package through its name. This is generally not true in Mof: it is valid in Mof to have a package named P containing a subpackage of the same name P. The package name uniqueness requirement does not change the structural consistency of metamodels, and the convention we proposed is not ambiguous, allowing one to retrieve the original metamodel names.

Example 6.4 (Fsm Package Function)**.** Let $p_{\mathsf{FSM}} \in \mathcal{P}$ be the package function for the Fsm metamodel. As noticed in Example 6.1, $\mathsf{Dom}\,(p_{\mathsf{FSM}}) = \{\mathsf{FSM}\}$, and its image is $(\varnothing, \mathsf{C}_{\mathsf{FSM}}, \{\mathsf{Kind}\})$ with $\mathsf{C}_{\mathsf{FSM}}$ defined in Example 6.1. □

6.3.2 Enumeration

From Fig. 5.3, the class *Enum* has one reference *literals* (with its opposite reference *enum*) to the class *EnumLit*, constraining an enumeration to possess at least one literal. The *EnumLit* class also inherits from *NamedElement*, making any literal possessing a name, and the set of literals for an enumeration is ordered.

The set $\mathcal{E}$ of *enumeration functions* represents *enumeration declarations*: such a function $e \in \mathcal{E}$ maps enumerations declared inside a package to an ordered set of enumeration literals.

$$\mathcal{E} \triangleq \{e \;:\; \mathsf{PkgN} \nrightarrow \mathsf{EnumN} \nrightarrow \langle\!\langle \mathbb{E} \rangle\!\rangle\}$$

Again, we impose that enumeration literals are unique throughout a metamodel. This is also generally not true, neither in Mof or in Kermeta. Nevertheless, this constraint seems reasonable: in Kermeta, enumeration literals are always qualified by their enumeration name (cf. Appendix A.2 in *getStart*'s body). Like for package names, it is always possible to ensure this requirement by considering qualified enumeration literal names.

Under this condition, it is possible to define a reverse function *enum* that retrieves the enumeration which a literal is defined in.

$$\begin{array}{rcl} \mathit{enum} : \mathsf{EnumLit} & \rightarrow & \mathsf{PkgN} \times \mathsf{Enum} \\ \mathsf{lit} & \mapsto & (\mathsf{pkg}, \mathsf{enum}) \text{ if } \mathsf{lit} \in e(\mathsf{pkg})(\mathsf{enum}) \end{array}$$

Example 6.5 (Fsm Enumeration Function)**.** Let $e_{\mathsf{FSM}} \in \mathcal{E}$ be the enumeration function for the Fsm metamodel. Recall that it is right associative. We define the domains of all partial functions for this first time, we will then omit this trivial details unless it contains something special. The Fsm metamodel contains only one package, which contains only one enumeration: $\mathsf{Dom}\,(e_{\mathsf{FSM}}) = \{\mathsf{FSM}\}$, and $\mathsf{Dom}\,(e_{\mathsf{FSM}}(\mathsf{FSM})) = \{\mathsf{Kind}\}$. Furthermore, this enumeration contains

[3]Remember that package names are flattened, making each package name unique within a metamodel.

three values: $e_{\mathsf{FSM}}(\mathsf{FSM})(\mathsf{Kind}) = \{\mathsf{START}, \mathsf{STOP}, \mathsf{NORMAL}\}$. This induces for example that $enum(\mathsf{START}) = (\mathsf{FSM}, \mathsf{Kind})$. □

6.3.3 Class

From Fig. 5.3, the class *Class* has one attribute and two references. The attribute *abstract* indicates that a class is abstract; it implies that the class is concretely represented with its name in italic, and make this class not directly instantiable. The *super* reference indicates that a class can refer to superclasses from which it inherits; and the *features* reference (with its opposite *class*) allows a class to declare features, i.e. properties and/or operations.

The set $\mathcal{C}$ of *class functions* represents *classes declarations*: such a function $c \in \mathcal{C}$ maps classes declared inside a package to a boolean indicating if they are abstract, and the set of their superclasses; it is *correct* if no class inherits from itself, and if the inheritance hierarchy is acyclic.

$$\mathcal{C} \triangleq \{c : \mathsf{PkgN} \nrightarrow \mathsf{ClassN} \nrightarrow \mathbb{B} \times \wp(\mathsf{ClassN}) \mid (\mathsf{pkg}, \mathsf{c}) \not\prec^{+}_{\mathsf{Class}} (\mathsf{pkg}, \mathsf{c})\}$$

The previous definition uses a (partial) order relation $\prec_{\mathsf{Class}} \subseteq \mathsf{PClassN} \times \mathsf{PClassN}$, induced by the inheritance hierarchy inside the same package (i.e. $(\mathsf{pkg}, \mathsf{c}) \prec_{\mathsf{Class}} (\mathsf{pkg}, \mathsf{c}')$ if c defines c' as one of its superclass). The relation $\prec^{+}_{\mathsf{Class}}$ used in the previous definition is the transitive closure of this order.

$$\forall \mathsf{pkg} \in \mathsf{Dom}\,(c), (\mathsf{pkg}, \mathsf{c}) \prec_{\mathsf{Class}} (\mathsf{pkg}, \mathsf{c}') \overset{\triangle}{\Longleftrightarrow} \mathsf{c}, \mathsf{c}' \in \mathsf{Dom}\,(c(\mathsf{pkg})) \;\wedge\; c(\mathsf{pkg})(\mathsf{c}) = (\_, C) \wedge \mathsf{c}' \in C\}$$

We define an order $\preccurlyeq_{\mathsf{Type}}$ on Type by union of the order $\preccurlyeq_{\mathsf{Prim}}$ on primitive types (from Definition 6.2) and the order $\preccurlyeq_{\mathsf{Class}}$ on classes (enumerations are not ordered). Then, $\preccurlyeq_{\mathsf{Type}}$ is extended into $\preccurlyeq \subseteq \mathsf{CType} \times \mathsf{CType}$ to cover collection type:

$$\begin{array}{rcl} \preccurlyeq_{\mathsf{Type}} & = & \preccurlyeq_{\mathsf{Class}} \cup \preccurlyeq_{\mathsf{Prim}} \\ (\mathsf{C}, \mathsf{t}) \preccurlyeq (\mathsf{C}', \mathsf{t}') & \overset{\triangle}{\Longleftrightarrow} & \mathsf{C} \preccurlyeq_{\mathsf{Coll}} \mathsf{C}' \wedge \mathsf{t} \preccurlyeq_{\mathsf{Type}} \mathsf{t}' \end{array}$$

Example 6.6 (FSM Class Function)**.** Let $c_{\mathsf{FSM}} \in \mathcal{C}$ be the class function for the FSM metamodel. The domains are $\mathsf{Dom}\,(c_{\mathsf{FSM}}) = \{\mathsf{FSM}\}$, and $\mathsf{Dom}\,(c_{\mathsf{FSM}})(\mathsf{FSM}) = \mathsf{C}_{\mathsf{FSM}}$ (as defined in Example 6.2). All classes except Label are similarly defined: all inherit from Label and are concrete, whereas Label is abstract. We therefore have the following definitions (and similarly for other classes):

$$\begin{array}{c} c_{\mathsf{FSM}}(\mathsf{FSM})(\mathsf{Label}) = (\top, \varnothing) \\ c_{\mathsf{FSM}}(\mathsf{FSM})(\mathsf{FSM}) = (\bot, \{\mathsf{Label}\}) \end{array}$$

This induces the class order $\prec_{\mathsf{Class}}$ to be defined for all class c other than Label by $(\mathsf{FSM}, \mathsf{c}) \prec_{\mathsf{Class}} (\mathsf{FSM}, \mathsf{Label})$. □

6.3.4 Property

In Fig. 5.3, the class *Property* inherits from *Feature*, which in turn inherits from *CollectionType*. The class *Feature* has one reference *from* that indicates optionally indicates which class should be considered if the feature (name) is multiply inherited, and a *class* reference indicating that a feature is always always declared, or contained, in one class. The class *CollectionType* simply represents an MType. Since *Property* is abstract, we should look at the concrete classes that inherit from it: the *Attribute* class has no properties, but is required to possess a *DataType* as a *type*; whereas the *Reference* class is required to possess a *Class* as a *type* (cf. (Steinberg et al. 2009)). Furthermore, the *Reference* class has one attribute *containment*, indicating if the reference is a containment (which is notationally represented by a black diamond in the concrete notation); and an *opposite* reference indicating that a reference admits another reference from its type as an opposite (which is notationally represented by removing the arrow head).

The set $\mathcal{P}rop$ of *property functions* represents *property declarations*: such a function $\text{prop} \in \mathcal{P}rop$ maps each class property to a boolean indicating if it is a containment reference, an optional class used for the from clause, the multiplicity type of the property and its optional opposite reference.

$$\mathcal{P}rop \triangleq \{\text{prop} \;:\; \mathsf{PkgN} \nrightarrow \mathsf{ClassN} \nrightarrow \mathsf{PropN} \nrightarrow \mathbb{B} \times \mathsf{ClassN}_{\perp} \times \mathsf{MType} \times \mathsf{RefN}_{\perp}\}$$

Suppose that $\text{prop}(\mathsf{pkg})(\mathsf{class})(\mathsf{p}) = (cnt, \mathsf{from}, \mathsf{mt}, \mathsf{opp})$ with $\mathsf{mt} = (low, up, \mathsf{coll}, \mathsf{t})$ is a property function. Then prop is *correct* if it respects the previously stated constraints:

- if $\mathsf{p} \in \mathsf{AttN}$ is an attribute, it is assumed to be always contained[4], does not possess an opposite, and its type is $\mathsf{DataType}$.

$$cnt = \top \wedge \mathsf{t} \in \mathsf{DataType} \wedge \mathsf{opp} = \perp$$

- if $\mathsf{p} \in \mathsf{RefN}$ is a reference, then its type is a class and its opposite reference opp is conversly mapped to the enclosing class of p

$$\mathsf{t} = (\mathsf{pkg}', \mathsf{c}') \in \mathsf{PClassN} \wedge \mathsf{opp} \neq \perp \implies \begin{cases} \text{prop}(\mathsf{pkg}')(\mathsf{c}')(\mathsf{opp}) & = & (cnt', \mathsf{from}', \mathsf{mt}', \mathsf{p}) \\ \text{with } \mathsf{mt}' & = & (low', up', \mathsf{coll}', \mathsf{pkg}, \mathsf{c}) \\ \text{and } cnt = \top & \implies & cnt' = \perp \wedge (low', up') = (0, 1) \end{cases}$$

Example 6.7 (FSM Property Function)**.** Let $prop_{\mathsf{FSM}} \in \mathcal{P}rop$ be the property function for the FSM metamodel. We illustrate the definition of properties contained in classes Label and FSM: they cover the main variations encountered within the FSM metamodel.

$$\begin{aligned} prop_{\mathsf{FSM}}(\mathsf{FSM})(\mathsf{Label})(\mathsf{label}) &= (\top, \perp, (1, 1, \perp, \mathsf{String}), \perp) \\ prop_{\mathsf{FSM}}(\mathsf{FSM})(\mathsf{FSM})(\mathsf{alphabet}) &= (\top, \perp, (1, \star, \mathsf{Set}, \mathsf{String}), \perp) \\ prop_{\mathsf{FSM}}(\mathsf{FSM})(\mathsf{FSM})(\mathsf{states}) &= (\top, \perp, (1, \star, \mathsf{OSet}, \mathsf{State}), \mathsf{fsm}) \\ prop_{\mathsf{FSM}}(\mathsf{FSM})(\mathsf{FSM})(\mathsf{transitions}) &= (\top, \perp, (0, \star, \mathsf{OSet}, \mathsf{Transition}), \mathsf{fsm}) \end{aligned}$$

For example, reference $\mathsf{transitions}$ in class FSM is a containement reference (the first $\top$), and is not ambiguously defined (second argument $\perp$). Its multiple type is defined according to Example 6.2, and it has an opposite reference fsm from class $\mathsf{Transition}$. Attribute $\mathsf{alphabet}$ is similarly defined, except that it cannot have an opposite, thus the $\perp$ as last element. As an example, and to be able to check that *prop* is correct, here is the definition for reference fsm in class $\mathsf{Transition}$:

$$\text{prop}(\mathsf{FSM})(\mathsf{Transition})(\mathsf{fsm}) = (\perp, \perp, (1, 1, \perp, \mathsf{FSM}), \mathsf{transitions})$$

□

6.3.5 Operation

In Fig. 5.3, the class *Operation* inherits from *Feature*, which in turn inherits from *CollectionType*. Therefore, it shares with the *Property* class the same capabilities. Furthemore, the *Operation* class has one attribute and two references. The *abstract* attribute indicates that an operation is abstract, which means that it has no body and

[4]With respect to containment, Kermeta's concrete syntax is interesting: the keyword `attribute` is used not only for MOF's attributes, but also MOF's references that are containments. Interestingly, UML diagrammatic notation represents attributes "contained" in the box representing a class.

is also contained in an abstract class. The *parameters* reference points to the *Parameter* class, which represents the (ordered) list of parameters, which are typed because the *Parameter* class inherits from *CollectionType*. The *statements* reference points to the *Body* class and corresponds to the body of the operation, whose precise definition is the purpose of Chapter 7.

The set $\mathcal{O}$ of *operation functions* represents *operations declarations*: such a function $o \in \mathcal{O}$ maps each class operation to a boolean indicating if it is abstract, an optional class used for the from clause, a multiplicity type representing the return type (where $\perp$ corresponds to `void`), a (ordered) sequence of parameters together with their types, and the definition of the body; and o is correct if every operation defined as abstract does not have a body.

$$\begin{aligned}
\mathcal{O} &\triangleq \{o\colon \mathsf{PkgN} \nrightarrow \mathsf{ClassN} \nrightarrow \mathsf{OpN} \nrightarrow \mathbb{B} \times \mathsf{ClassN}_\perp \times \langle\!\langle \mathcal{P}\mathsf{arams} \rangle\!\rangle \times \mathsf{MType}_\perp \times \mathsf{Body}_\perp \mid \\
&\qquad \forall \mathsf{pkg} \in \mathsf{PkgN}, \mathsf{c} \in \mathsf{ClassN}, \mathsf{op} \in \mathsf{OpN}, o(\mathsf{pkg})(\mathsf{c})(\mathsf{op}) = (abs, \_, \_, \_, b), abs \implies b = \perp\} \\
\mathcal{P}\mathsf{arams} &\triangleq \mathsf{ParamN} \times \mathsf{MType}
\end{aligned}$$

Here, $\mathcal{P}\mathsf{arams}$ is defined as a pair containing the parameter name and its multiplicity type.

Example 6.8 (FSM Operation Function)**.** Let $o_{\mathsf{FSM}} \in \mathcal{O}$ be the operation function for the FSM metamodel. We illustrate the definition of operations contained in classes FSM: this class contains operations with and without parameters.

$$\begin{aligned}
o(\mathsf{FSM})(\mathsf{FSM})(\mathsf{accept}) &= (\perp, \perp, \langle\!\langle(\mathsf{word}, (0, \star, \mathsf{List}, \mathsf{String}))\rangle\!\rangle, (1, 1, \perp, \mathsf{Boolean}), b_{\mathsf{accept}}) \\
o(\mathsf{FSM})(\mathsf{FSM})(\mathsf{getStart}) &= (\perp, \perp, \langle\!\langle\rangle\!\rangle, (1, 1, \perp, \mathsf{State}), b_{\mathsf{getStart}}) \\
o(\mathsf{FSM})(\mathsf{FSM})(\mathsf{getFinal}) &= (\perp, \perp, \langle\!\langle\rangle\!\rangle, (1, 1, \perp, \mathsf{State}), b_{\mathsf{getFinal}})
\end{aligned}$$

Take operation accept: it is not abstract and is not redefined to need a from clause, which justifies the two first $\perp$; it has one parameter word, given with its multiple type, and has as a result a Boolean, encoded by the multiple type $(1, 1, \perp, \mathsf{Boolean})$ (with lower and upper bounds both equal to 1, we impose this return value); and b_{accept} just captures a list of statements constituting accept's body. The two getters are similarly defined, except that they return a State and that they have no parameters (denoted by $\langle\!\langle\rangle\!\rangle$). □

6.3.6 Metamodel

The set of metamodel $\mathcal{M}$ can now be defined by putting the previous definitions together. Note that the use of (partial) function naturally ensures Kermeta's constraint on unicity of names: it is not possible to represent, in our framework, e.g. a metamodel that would contain two packages (or two properties in the same class) with the same name.

Definition 6.4 (Metamodel $\mathcal{M}$)**.** *A metamodel* $\mathsf{MM} \in \mathcal{M}$ *is a tuple* $\mathsf{MM} = (p, c, e, \mathrm{prop}, o) \in \mathcal{P} \times \mathcal{C} \times \mathcal{E} \times \mathcal{P}\mathsf{rop} \times \mathcal{O}$.

Let $\mathsf{MM} = (p, c, e, \mathrm{prop}, o) \in \mathcal{M}$ be a metamodel of interest. The set of *qualified property names* QPropN (respectively, *qualified operation names*, and *qualified feature names*) is the set of property (resp. operation, feature) names correctly defined in the scope of MM:

$$\begin{aligned}
\mathsf{QPropN}_{\mathsf{MM}} &\triangleq \{(\mathsf{pkg}, \mathsf{c}, \mathsf{p}) \in \mathsf{PkgN} \times \mathsf{ClassN} \times \mathsf{PropN} \mid \mathsf{pkg} \in \mathsf{Dom}\,(\mathrm{prop}) \wedge \mathsf{c} \in \mathsf{Dom}\,(\mathrm{prop}(\mathsf{pkg})) \wedge \mathsf{p} \in \mathsf{Dom}\,(\mathrm{prop}(\mathsf{pkg})(\mathsf{c}))\} \\
\mathsf{QOpN}_{\mathsf{MM}} &\triangleq \{(\mathsf{pkg}, \mathsf{c}, \mathsf{op}) \in \mathsf{PkgN} \times \mathsf{ClassN} \times \mathsf{OpN} \mid \mathsf{pkg} \in \mathsf{Dom}\,(o) \wedge \mathsf{c} \in \mathsf{Dom}\,(o(\mathsf{pkg})) \wedge \mathsf{p} \in \mathsf{Dom}\,(o(\mathsf{pkg})(\mathsf{c}))\} \\
\mathsf{QFeatureN}_{\mathsf{MM}} &\triangleq \{(\mathsf{pkg}, \mathsf{c}, \mathsf{f}) \in \mathsf{PkgN} \times \mathsf{ClassN} \times \mathsf{FeatureN} \mid (\mathsf{pkg}, \mathsf{c}, \mathsf{f}) \in \mathsf{QPropN} \vee (\mathsf{pkg}, \mathsf{c}, \mathsf{f}) \in \mathsf{QOpN}\}
\end{aligned}$$

Example 6.9 (FSM Qualified Names)**.** We can now mathematically build the metamodel $\mathsf{MM}_{\mathsf{FSM}} \in \mathcal{M}$ corresponding to the FSM metamodel from all the previous examples:

$$\mathsf{MM}_{\mathsf{FSM}} = (p_{\mathsf{FSM}}, c_{\mathsf{FSM}}, e_{\mathsf{FSM}}, prop_{\mathsf{FSM}}, o_{\mathsf{FSM}})$$

The qualified name sets corresponding to $\mathsf{MM}_{\mathsf{FSM}}$ then correspond to the following:

$$\begin{aligned}
\mathsf{QPropN} &= \{(\mathsf{FSM}, \mathsf{Label}, \mathsf{label}), (\mathsf{FSM}, \mathsf{FSM}, \mathsf{alphabet}), (\mathsf{FSM}, \mathsf{FSM}, \mathsf{states}), (\mathsf{FSM}, \mathsf{FSM}, \mathsf{transitions}),\\
&\quad (\mathsf{FSM}, \mathsf{State}, \mathsf{kind}), (\mathsf{FSM}, \mathsf{State}, \mathsf{fsm}), (\mathsf{FSM}, \mathsf{State}, \mathsf{in}), (\mathsf{FSM}, \mathsf{State}, \mathsf{out}),\\
&\quad (\mathsf{FSM}, \mathsf{Transition}, \mathsf{fsm}), (\mathsf{FSM}, \mathsf{Transition}, \mathsf{src}), (\mathsf{FSM}, \mathsf{Transition}, \mathsf{tgt})\}\\
\mathsf{QOpN} &= \{(\mathsf{FSM}, \mathsf{FSM}, \mathsf{accept}), (\mathsf{FSM}, \mathsf{FSM}, \mathsf{getStart}), (\mathsf{FSM}, \mathsf{FSM}, \mathsf{getFinal}), (\mathsf{FSM}, \mathsf{State}, \mathsf{fire})\}
\end{aligned}$$

□

A number of functions are also defined to ease the manipulation of a particular metamodel's information. These functions act like projector functions, i.e. they select a particular element in the image of the functions constituting the metamodel.

$$\begin{aligned}
super_{\mathsf{MM}} &: \mathsf{PClassN} \longrightarrow \wp(\mathsf{Class}) & \mathrm{from}_{\mathsf{MM}} &: \mathsf{QFeatN} \longrightarrow \mathsf{Class}_{\perp}\\
\mathrm{abs}_{\mathsf{MM}} &: \mathsf{QOpN} \cup \mathsf{PClassN} \longrightarrow \mathbb{B} & \mathrm{type}_{\mathsf{MM}} &: \mathsf{QFeatN} \longrightarrow \mathsf{MType}_{\perp}\\
\mathrm{partypes}_{\mathsf{MM}} &: \mathsf{QOpN} \longrightarrow \langle\!\langle \mathsf{CType} \rangle\!\rangle & \mathrm{parnames}_{\mathsf{MM}} &: \mathsf{QOpN} \longrightarrow \langle\!\langle \mathsf{ParamN} \rangle\!\rangle
\end{aligned}$$

We only give the formal definition for super, the other functions are built the same way from MM's functions. Let $(\mathsf{pkg}, \mathsf{c}) \in \mathsf{PClassN}$ such that $\mathsf{c} \in \mathrm{Dom}\,(c(\mathsf{pkg}))$, $\mathrm{super}_{\mathsf{MM}}(\mathsf{pkg}, \mathsf{c}) = C \overset{\triangle}{\Longleftrightarrow} c_{\mathsf{MM}}(\mathsf{pkg})(c) = (\_, C)$: it represents the set of $(\mathsf{pkg}, \mathsf{c})$ superclasses. Similarly, abs indicates if a class $(\mathsf{pkg}, \mathsf{c})$ or an operation $(\mathsf{pkg}, \mathsf{c}, \mathsf{op})$ is abstract; the fonction $\mathrm{from}_{\mathsf{MM}}$ retrieves the disambiguation class (if any) of a feature; and type the multiplicity type of a feature (which can be $\perp$, i.e. *void*, in the case of an operation); and partypes and parnames retrieve the ordered list of respectively the collection types and the names of an operation's parameters.

Example 6.10 (FSM Auxilliary Functions)**.** We illustrate each function on $\mathsf{MM}_{\mathsf{FSM}}$ using one example, whose result can be retrieved from the previous examples:

$$\begin{aligned}
super_{\mathsf{MM}_{\mathsf{FSM}}}(\mathsf{FSM}, \mathsf{FSM}) &= \{\mathsf{Label}\}\\
abs_{\mathsf{MM}_{\mathsf{FSM}}}(\mathsf{FSM}, \mathsf{Label}) &= \top\\
abs_{\mathsf{MM}_{\mathsf{FSM}}}(\mathsf{FSM}, \mathsf{FSM}) &= \perp\\
abs_{\mathsf{MM}_{\mathsf{FSM}}}(\mathsf{FSM}, \mathsf{FSM}, \mathsf{accept}) &= \perp\\
partypes_{\mathsf{MM}_{\mathsf{FSM}}}(\mathsf{FSM}, \mathsf{FSM}, \mathsf{accept}) &= \langle\!\langle (0, \star, \mathsf{List}, \mathsf{String}) \rangle\!\rangle\\
parnames_{\mathsf{MM}_{\mathsf{FSM}}}(\mathsf{FSM}, \mathsf{FSM}, \mathsf{accept}) &= \langle\!\langle \mathsf{word} \rangle\!\rangle\\
type_{\mathsf{MM}_{\mathsf{FSM}}}(\mathsf{FSM}, \mathsf{FSM}, \mathsf{accept}) &= (1, 1, \perp, \mathsf{Boolean})\\
type_{\mathsf{MM}_{\mathsf{FSM}}}(\mathsf{FSM}, \mathsf{FSM}, \mathsf{getStart}) &= \perp\\
type_{\mathsf{MM}_{\mathsf{FSM}}}(\mathsf{FSM}, \mathsf{FSM}, \mathsf{alphabet}) &= (1, \star, \mathsf{Set}, \mathsf{String})\\
type_{\mathsf{MM}_{\mathsf{FSM}}}(\mathsf{FSM}, \mathsf{FSM}, \mathsf{states}) &= (0, \star, \mathsf{OSet}, \mathsf{State})
\end{aligned}$$

□

A metamodel $\mathsf{MM} = (p, e, c, \mathrm{prop}, o)$ is *valid* if all classes declared inside a package also appear in the domains of c, prop and o under the same package, and every class containing an abstract operation is also declared abstract[5].

[5] Strictly speaking, this constraint comes from Kermeta: "*Kermeta requires that every class that contains an abstract operation must be declared as an abstract class.*" (Drey et al. 2009, §2.8.2)

$$\begin{array}{rcl} p(\mathsf{pkg}) = (P, C, E) & \implies & \begin{cases} E & = & \mathsf{Dom}\,(e(\mathsf{pkg})) \\ C & = & \mathsf{Dom}\,(c(\mathsf{pkg})) \\ C & = & \mathsf{Dom}\,(\mathrm{prop}(\mathsf{pkg})) \\ C & = & \mathsf{Dom}\,(o(\mathsf{pkg})) \end{cases} \\ abs(\mathsf{pkg}, \mathsf{c}, \mathsf{o}) & \implies & abs(\mathsf{pkg}, \mathsf{c}) \end{array}$$

Example 6.11 (FSM Metamodel Validity)**.** $\mathsf{MM}_{\mathsf{FSM}}$ is obviously valid: domains are correctly defined; and the condition on abstract components obviously holds since the metamodel contains no abstract operation. □

6.4 Models

A metamodel defines a set of models. A model basically consists of a collection of *objects* (or equivalently called *instances*). Each object has a *type*, i.e. a class of a metamodel, and maintains a *state*. An object's state is intuitively a mapping between property names and values, in such a way that the involved names correspond to the declared names for the object's type.

6.4.1 Accessible Features

In the presence of inheritance, and especially multiple inheritance, referring to features by their names might be ambiguous because the names can be repeated throughout the inheritance hierarchy. In fact, we will show that for a given object, each property name is unique with respect to the disambiguation clause.

Suppose now a model $\mathsf{M} \in \mathbb{M}$ of a metamodel $\mathsf{MM} \in \mathcal{M}$, and an object o of type $(\mathsf{pkg}, \mathsf{c}) \in \mathsf{PClassN}$. To be correctly defined, o's state should associate a value to all *accessible* properties declared in MM, i.e. those properties defined in the *inheritance scope* of c: either directly declared in c, or inherited from c's superclasses. Similarly, o's accessible operations are those operations that are either defined directly in c or inherited from superclasses.

We define two functions to capture *accessible features*: the function π_{MM} over a metamodel MM (omitted when clear from context) recursively computes the information of all properties accessible from classes of MM; and the function ω_{MM} recursively computes the information of all accessible operations.

$$\begin{array}{rcl} \pi_{\mathsf{MM}} & : & \mathsf{PkgN} \nrightarrow \mathsf{ClassN} \nrightarrow \mathsf{PropN} \nrightarrow \mathbb{B} \times \mathsf{MType} \times \mathsf{RefN}_{\perp} \\ \omega_{\mathsf{MM}} & : & \mathsf{PkgN} \nrightarrow \mathsf{ClassN} \nrightarrow \mathsf{OpN} \nrightarrow \mathbb{B} \times \langle\!\langle \mathcal{P}\mathsf{arams} \rangle\!\rangle \times \mathsf{MType}_{\perp} \times \mathsf{Body} \end{array}$$

Notice that the signature of these functions is slightly different: since they compute the accessible features, they are not ambiguous any more: no component records the disambiguation clause.

Functions π_{MM} and ω_{MM} are well-founded because they follow the well-founded order $\prec_{\mathsf{Class}}$ over classes. Nevertheless, we must ensure that they are well-defined, i.e. *each accessible property for o has an unique name* and *each call resolves into a unique operation.*These properties hold for different reasons; we only sketch the proof for each of them.

Properties. First, as we mentioned before, direct property names are unique. Second, Kermeta, following MOF, disallows property overriding because "*it simply does not make sense from a structural point of view*" (Drey et al. 2009, §2.9.5): this ensures that in a single inheritance path, property names are unique. Finally, any class with multiple superclasses must ensure uniqueness of property names by using the *from* disambiguation clause.

Operations. First, as we mentioned before, overloading is forbidden in Kermeta: inside a given class, operation names are unique. Second, Kermeta allows invariant overriding, meaning that in a single inheritance path, an operation call resolves to the closest operation up to the class hierarchy. Finally, in case of multiply inherited operations, each class must ensure the uniqueness of the inherited operations (Drey et al. 2009, §2.9.5.4) by using the *from* disambiguation clause.

Consequently, each accessible property has a unique name and we can define a function decl that associates to a property (name) *the* (unique) class where it is defined.

$$\begin{array}{rcl} \text{decl}\colon \mathsf{PClassN} \times \mathsf{PropN} & \nrightarrow & \mathsf{PClassN} \\ ((\mathsf{pkg},\mathsf{c}),\mathsf{propN}) & \mapsto & \begin{cases} (\mathsf{pkg},\mathsf{c}) & \text{if } \ \text{from}(\mathsf{pkg},\mathsf{c},\mathsf{propN}) = \bot \\ & \text{and } \mathsf{propN} \in \mathsf{Dom}\,(\pi(\mathsf{pkg})(\mathsf{c})) \\ (\mathsf{pkg},\mathsf{c}') & \text{if } \ \text{from}(\mathsf{pkg},\mathsf{c},\mathsf{propN}) = \mathsf{c}' \\ & \text{and } \mathsf{propN} \in \mathsf{Dom}\,(\pi(\mathsf{pkg})(\mathsf{c}')) \end{cases} \end{array}$$

As operations rely on dynamic lookup, i.e. the chosen operation's body for a call depends on the dynamic type of the object, it is not possible to define a similar function for operations. The decision will be made dynamically (cf. 7.4.3.4).

Example 6.12 (FSM Accessible Features)**.** Recall, from the previous examples, the metamodel corresponding to the FSM $\mathsf{MM}_{\mathsf{FSM}} = (p_{\mathsf{FSM}}, c_{\mathsf{FSM}}, e_{\mathsf{FSM}}, prop_{\mathsf{FSM}}, o_{\mathsf{FSM}}) \in \mathcal{M}$. We had for example for class State that $\mathsf{Dom}\,(prop_{\mathsf{FSM}}(\mathsf{FSM})(\mathsf{State})) = \{\mathsf{kind}, \mathsf{fsm}, \mathsf{in}, \mathsf{out}\}$, meaning that these properties were actually declared within class State. However for our model, we need to set the State's label, inherited from class Label. Hence, from the inheritance hierarchy induced by c_{FSM} (cf. Example 6.6), we obtain that

$$\mathsf{Dom}\,(\pi_{\mathsf{FSM}}(\mathsf{FSM})(\mathsf{State})) = \{\mathsf{kind}, \mathsf{fsm}, \mathsf{in}, \mathsf{out}, \mathit{label}\}$$

The same mechanism holds for ω_{FSM}, but it is not possible to illustrate it on the FSM since no operation is inherited or redefined. □

6.4.2 Model

The following definition only translates in a mathematical structure the considerations made at the beginning of this Section, taking advantage of the previous remarks on uniqueness of properties.

Definition 6.5 (Model $\mathbb{M}$)**.** *The set of models $\mathbb{M}$ is a set of functions that associate model objects to their type and state.*

$$\begin{array}{rcl} \mathbb{S}\mathsf{tate} & \triangleq & \{\sigma \ : \ \mathsf{PropN} \nrightarrow \mathbb{V}\} \\ \mathbb{M} & \triangleq & \{\mathsf{M} \ : \ \mathbb{O} \nrightarrow \mathsf{PClassN} \times \mathbb{S}\mathsf{tate}\} \end{array}$$

Given a model $\mathsf{M} \in \mathbb{M}$, we note σ^o_{M} the state of the object $o \in \mathbb{O}$, if $o \in \mathsf{Dom}\,(\mathsf{M})$ and $type_{\mathsf{M}}(o)$ its type (subscripts are omitted if clear from context).

Example 6.13 (FSM Model)**.** Let us now call $\mathsf{M}_{\mathsf{abc}} \in \mathbb{M}$ the representation of the model depicted in Fig. 5.4. Using the names as object identifiers, we have $\mathsf{Dom}\,(\mathsf{M}_{\mathsf{abc}}) = \{abc, 1, 2, 3, a, b, c\}$. We obviously have $type(abc) = (\mathsf{FSM}, \mathsf{FSM})$, $type(1) = type(2) = type(3) = (\mathsf{FSM}, \mathsf{State})$ and $type(a) = type(b) = type(c) = (\mathsf{FSM}, \mathsf{Transition})$. We only describe the state of the necessary instances for the conformance proof.

$$\begin{aligned}
\sigma^{abc}(\mathsf{label}) &= \text{"(ab) + c"} \\
\sigma^{abc}(\mathsf{alphabet}) &= \{\text{"a"}, \text{"b"}, \text{"c"}\} \\
\sigma^{abc}(\mathsf{states}) &= \langle\!\langle 1, 2, 3 \rangle\!\rangle \\
\sigma^{abc}(\mathsf{transitions}) &= \langle\!\langle a, b, c \rangle\!\rangle
\end{aligned}
\qquad
\begin{aligned}
\sigma^{1}(\mathsf{label}) &= \text{"1"} \\
\sigma^{1}(\mathsf{kind}) &= \mathsf{START} \\
\sigma^{1}(\mathsf{in}) &= \langle\!\langle b \rangle\!\rangle \\
\sigma^{1}(\mathsf{out}) &= \langle\!\langle a \rangle\!\rangle \\
\sigma^{1}(\mathsf{fsm}) &= abc
\end{aligned}$$

$$\begin{aligned}
\sigma^{a}(\mathsf{label}) &= \text{"a"} \\
\sigma^{a}(\mathsf{src}) &= 1 \\
\sigma^{a}(\mathsf{tgt}) &= 2 \\
\sigma^{a}(\mathsf{fsm}) &= abc
\end{aligned}
\qquad
\begin{aligned}
\sigma^{b}(\mathsf{label}) &= \text{"b"} \\
\sigma^{b}(\mathsf{src}) &= 2 \\
\sigma^{b}(\mathsf{tgt}) &= 1 \\
\sigma^{b}(\mathsf{fsm}) &= abc
\end{aligned}$$

□

The following definition relates values to (collection) types in the context of a model: types of scalar values (booleans, integers, reals and strings) do not depend on the model; the type of an enumeration literal is the enumeration where this literal is defined; and the type of an object is the object's type in the model.

Definition 6.6 (Type of a value)**.** *The function τ trivially associates a (syntactic) type to a (semantic) value.*

$$\begin{aligned}
\tau_{\mathsf{M,MM}} : \mathbb{V} &\longrightarrow \mathsf{CType} \\
v &\mapsto \begin{cases}
(\bot, \mathsf{Boolean}) & \text{if } v \in \mathbb{B} \\
(\bot, \mathsf{Integer}) & \text{if } v \in \mathbb{N} \\
(\bot, \mathsf{Real}) & \text{if } v \in \mathbb{R} \\
(\bot, \mathsf{String}) & \text{if } v \in \mathbb{S} \\
(\bot, \mathrm{enum}_{\mathsf{MM}}(v)) & \text{if } v \in \mathbb{E} \\
(\bot, \mathrm{type}_{M}(v)) & \text{if } v \in \mathbb{O} \\
\dots & \\
(\mathsf{Bag}, \mathsf{Boolean}) & \text{if } v \in [\mathbb{B}] \\
(\mathsf{Set}, \mathsf{Boolean}) & \text{if } v \in \wp(\mathbb{B}) \\
\dots &
\end{cases}
\end{aligned}$$

6.5 Conformance

We say that M *conforms to* MM, and note $\mathsf{M} \blacktriangleright \mathsf{MM}$, if M actually belongs to the set induced by MM. This predicate is defined recursively: $\mathsf{M} \blacktriangleright \mathsf{MM}$ holds if all objects of the metamodel respect these conditions: $\forall o \in \mathsf{Dom}(\mathsf{M})$,

- o's type is declared in MM and each accessible property from this type has a value;

$$\mathrm{type}(o) = (\mathsf{pkg}, \mathsf{c}) \Longrightarrow \begin{cases}
\mathsf{p} \in \mathsf{Dom}(p) \\
p(\mathsf{pkg}) = (P, C, E) \Rightarrow \mathsf{c} \in C \\
(\mathsf{Dom}(\sigma^o) = \mathsf{Dom}(\pi(\mathsf{pkg})(\mathsf{c})) \\
\forall \mathsf{p} \in \mathsf{Dom}(\sigma^o), \sigma^o(\mathsf{p}) \in \mathbb{V}
\end{cases}$$

- For each property $\mathsf{p} \in \mathsf{PropN}$ that is accessible from o (i.e. $\mathsf{p} \in \mathsf{Dom}(\sigma^o)$ and that has value $v \in \mathbb{V}$ and type $\mathsf{mt} = (low, up, \mathsf{C}, \mathsf{t}) \in \mathsf{MType}$ (i.e. $\sigma^o(\mathsf{p}) = v$ and $\pi(\mathsf{pkg})(\mathsf{c})(\mathsf{p}) = \mathsf{mt}$), the following holds (i) v's type specialises (C, t); and (ii) v's size respects p's declared bounds.

$$\begin{array}{lrcl}
\text{(i)} & \mathrm{type}(v) & \preccurlyeq & (\mathsf{C}, \mathsf{t}) \\
\text{(ii)} & low \leqslant & |\, v \,| & \leqslant up
\end{array}$$

- Furthermore, if v is a collection of (internal) values $v_1, \dots, v_n$ (i.e. $\mathsf{C} \neq \bot$) then each of these v_i has a type that specialises t (remember that collection values are always well-formed (cf. Def. 6.3);

$$\forall i, type(v_i) \preccurlyeq \mathsf{t}$$

- Finally, if the property p is a reference with an opposite property opp (i.e. $\mathsf{p} \in \mathsf{RefN}$ such that $\mathsf{opp}(\mathsf{pkg}, \mathsf{c}, \mathsf{p}) = \mathsf{opp} \neq \bot$), then opp's value is either an object v itself, or a collection of (internal) objects $v'_1 \dots v'_n$; in this case, the original object o must be included between opp's (internal) value(s).

$$\mathsf{opp} \neq \bot \Longrightarrow \forall i, o \subseteq \sigma^{v_i}(\mathsf{opp})$$

Example 6.14 (Conformance of the FSM model to its metamodel)**.** We now have the full definition of the metamodel $\mathsf{MM}_{\mathsf{FSM}} = (p_{\mathsf{FSM}}, c_{\mathsf{FSM}}, e_{\mathsf{FSM}}, prop_{\mathsf{FSM}}, o_{\mathsf{FSM}}) \in \mathcal{M}$ and the model $\mathsf{M}_{\mathsf{abc}} \in \mathbb{M}$. The conformance basically states that all values and instances conform to their declarations. This proof is fully presented in Appendix A, since it is lengthy to present it fully here. □

6.6 Discussions

This Section first discusses some of the semantics' design choices and Kermeta's specific constructions, and then compares with related works in the domain of the Structural Language.

6.6.1 Design Choices

Names. The proposes formalisation heavily relies on names: the sets $\mathcal{M}$ of metamodels and $\mathbb{M}$ of models bind names with respectively types and values. A first point of discussion is the way names are extracted from metamodels. We briefly discussed in Sec. 6.2.1 and illustrated the extraction process from the graphical representation on our running example in Sec. 6.2.1. The extraction process from the textual representation is further simplified by the classical use of namespaces into BNF grammars. Moreover, we introduced two simplifications on package and enumeration literal names: each of these names is always prefixed by another name that is guaranteed to be unique (the "path" of package names, and the enumeration name, respectively). These simplifications seem reasonable since they are used in Kermeta itself (cf. Appendix A.2). A last point is the use of property namespaces in Kermeta: the keyword **attribute** introduces a property (i.e. either a MOF attribute or a reference) that is contained. We reflected this point in our constraint on the definition of $\mathcal{P}\mathsf{rop}$.

The choice of using names to build the definitions of $\mathcal{M}$ and $\mathbb{M}$ can easily be criticised in the sole context of a structural part's formalisation: it is easier to consider feature *identifiers*, reflecting what is behind MOF. Identifiers also avoid using qualified names, building the functions π_{MM} and ω_{MM}, and using the disambiguation clause **from**. As a counterpart, they introduce an unnecessary burden when dealing with the Action Language, whose statement construction relies explicitly on feature names. To deal with identifiers, one has either to completely change the Action Language itself to allow direct use of identifiers (which is, at the moment, unrealistic given the cascaded changed required in the whole Kermeta tool infrastructure), or to require an explicit mapping from identifiers to names, which is exactly the reason why identifiers are usually used. This latter solution assumes that the way identifiers are built is known from the user to define such a mapping properly: this contradicts the intuition of using models independently of their implementation.

Types. The types formalised in the set MType do not trustfully represent Kermeta's types. First, primitive types are not "*primitive*" but are rather meta-represented as classes in the package `kermeta::standard`: they all subclass `ValueType`, with eventually umbrella classes to wrap common features (as `Numeric` for numerical types Integer and Real). As a concrete consequence, they do not respect the expected Hasse Diagram of Def. 6.2 and require a proper API to implement that. Furthermore, some of these types do not possess a "surface" syntax and should be obtained by using API features (e.g., var x: Real init "1.0".toReal to obtain the real value 1.0).

Similarly, collection representation in Kermeta does not follow the natural Hasse Diagram of Def. 6.2 but rather adopts a hierarchy that favours implementation and code generation purposes (cf. (Drey et al. 2009, §2.14)): for example, the abstract class `OrderedCollection`, from which `OrderedSet` and `Sequence` inherit, factorises common features like iterators. These choices reflect more implementation efficency choices than theoretical considerations.

6.6.2 Related Works

We discuss in this Section the related works that focus on structural parts of a DSML; the discussion of those that directly relate to DSML transformations and DSML behavior are postponed to Chapter 7.

Historically, most formalisations focused on structural aspects, which is an understandable and natural first step. Several formal frameworks were used, but accordingly to our initial motivation, they contrast with our work by the fact they are expressed in the syntax of a particular tool. Moreover, some of them are incomplete regarding MOF.

As already mentioned, Algebraic Specifications (AS) were used to formalise MOF in (Boronat and Meseguer 2008) together with its reflection capabilities, using MAUDE. Abstract State Machines (ASM) serve both as formal foundations for MOF (Gargantini, Riccobene, and Scandurra 2009) and as a metamodeling framework for ASM (Gargantini, Riccobene, and Scandurra 2008), thus providing bridges between MDE and ASM tools. The Z language was used in (Song et al. 2005) but does not cover packages, and stays very general for attributes/references and does not take into consideration collection/multiplicity types, despite the fact that Z offers these concepts natively.

Category Theory is another important framework: Constructive Type Theory was used in (Poernomo 2006) to formalise MOF in the context of the "proofs-as-programs" concept, but without an explicit notion of containment/opposite references, and does not provide a clear mechanism to transform MOF specification into Constructive Type Theory; Graph-Based (GB) formalisations are used to achieve both formalisation of MOF and transformations (Biermann, Ermel, and Taentzer 2008; Rozenberg 1997), but usual MOF constructions like containment and inheritance were not addressed until recently (Jurack and Taentzer 2010).

Numerous contributions try to formalise UML diagrams (and in particular, as expected, UML Class Diagrams) by translating them into languages with well-defined semantics. As a general remark, these contributions always address specific parts or small subsets of UML Diagrams. Among many other formalisms, we already mentioned Z (Song et al. 2005), but also Object-Z in (Soon-Kyeong and Carrington 1999), the algebraic language CASL (Reggio, Cerioli, and Astesiano 2001) where the behavior is expressed with transition systems that naturally flow from algebraic specifications, and the theorem-prover PVS by abstracting Diagrams into state predicates in (Krishnan 2000).

7 Action Language

Kermeta's AL is structurally weaved into the SL by extending the class *Operation* with a *Body* (cf. Fig. 5.3), which consists of a sequence of statements (cf. (Drey et al. 2009; Muller, Fleurey, and Jézéquel 2005)). This Chapter starts by reminding the restrictions we consider on the original Kermeta AL. Following these restrictions, the Chapter proceeds by defining a *core* AL for Kermeta, sufficient to represent sufficiently rich Kermeta actions to handle a large variety of transformations: this core AL can be seen as an intermediate representation for Kermeta operations' bodies if the actions used comply with our subset AL. We will refer to our subset AL as *core* AL whereas Kermeta's AL will be refered to as *original*.

The Chapter then proceeds by formally specifying the core AL's semantics using the classical Structural Operational Semantics (SOS) framework (Winskel 1993): from the definition of the AL abstract syntax, we present a type-checking system in Sec. 7.3 that ensures the correct use of the statements; and finally provide the rewriting rules in Sec. 7.4.

7.1 Restrictions

Figure 7.1 depicts (an excerpt of) Kermeta's original AL, as presented in the Manual (Drey et al. 2009, §3.3) (where the AL is called "*Behavior Package*" and has one superclass named *Expression* from which all other constructions inherit).

We do not address *genericity* and *model type* because it heavily complicates the concise definition of the semantic domain: the classes *TypeLiteral* (for literal representing types), *LambdaParameter* (representing parameters that uses a generic type), *TypeReference* (for referencing a type) and *LambdaExpression* (for defining expressions that uses generics) are not part of our AL.

Since exception handling mechanisms concerns abnormal behaviours of transformations, and because we target formal verification, we do not consider exception constructions: the classes *Raise* (for raising an exception inside an operation's body) and *Rescue* (for declaring an exceptions attached to an operation) do not appear in our AL.

Finally, we do not consider *native calls* to Java (cf. *JavaStaticCall*) as it concerns more interaction with the platform.

Two other classes of this metamodel do not appear formally in our metamodel, namely *Block* and *VariableDecl*, because they are represented otherwise (see details below). All other classes from the *Expression* behavioral metamodel of Kermeta are handled in our core AL.

7.2 Definition

In Kermeta, the operation's body consists in an *ordered* sequence of *unique statements* (in the sense that, despite the possibility to have structurally identical statements, the statements themselves are different entities).

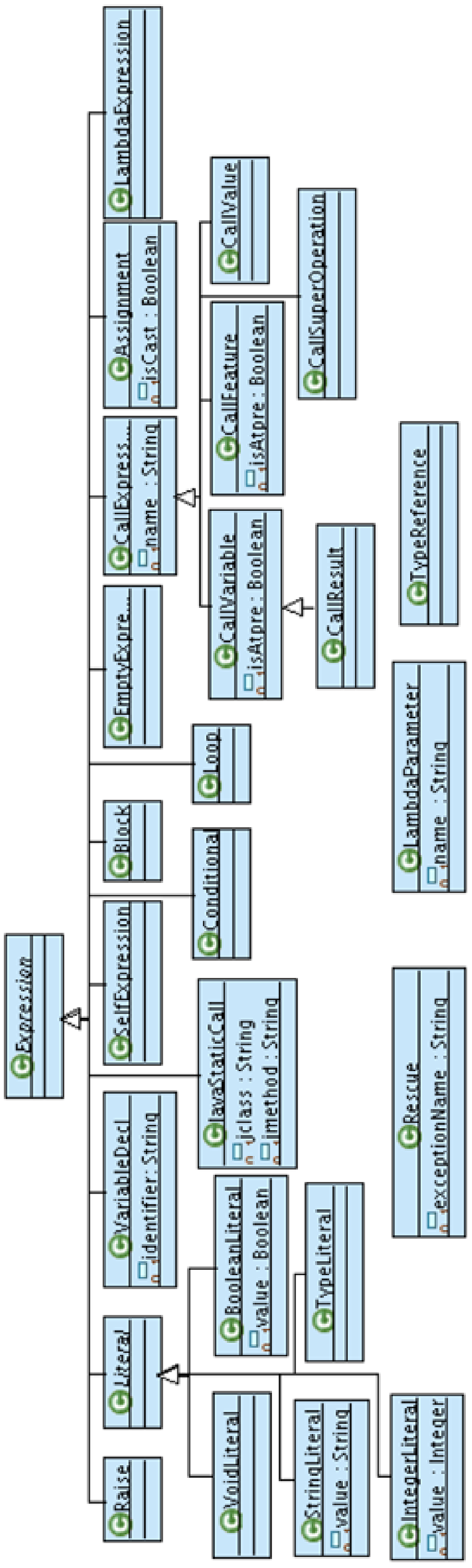


Figure 7.1 – Kermeta's Action Language (from (Drey et al. 2009, §3.3))

$$\begin{array}{rcl}
\mathsf{Exp} & ::= & \textbf{null} \mid \mathsf{scalarExp} \mid \mathsf{instExp} \mid \mathsf{collExp} \\
\mathsf{ScalarExp} & ::= & \mathsf{literal} \mid \mathsf{instExp}\ \textbf{instanceof}\ \mathsf{pClassN} \\
\mathsf{InstExp} & ::= & \textbf{self} \mid \mathsf{lhs} \\
\mathsf{Lhs} & ::= & \mathsf{varN} \mid \mathsf{paramN} \\
 & \mid & \mathsf{target.propN} \\
\mathsf{Target} & \mid & \textbf{super} \mid \mathsf{instExp} \\
\mathsf{CollExp} & \mid & \mathsf{exp.nativeExp} \\
\mathsf{NativeExp} & \mid & \mathit{isEmpty()} \mid \mathit{size()} \mid \mathit{at}(\mathsf{exp}) \\
 & & \\
\mathsf{LocalN} & ::= & \mathsf{varN} \mid \mathsf{paramN} \mid \textbf{self} \\
\mathsf{PClassN} & ::= & (\mathsf{pkgN}\ \mathsf{classN})
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{Body} & ::= & [\mathsf{Stm}]^{+} \\
\mathsf{Stm} & ::= & \mathit{lab}\text{: } \mathsf{Stmt} \\
\mathsf{Stmt} & ::= & \mathsf{condStmt} \\
 & \mid & \mathsf{assignStmt} \mid \mathsf{castStmt} \\
 & \mid & \mathsf{newInstStmt} \mid \mathsf{CollStmt} \\
 & \mid & \mathsf{returnStmt} \mid \mathsf{CallStmt} \\
\mathsf{CondStmt} & ::= & \textbf{if}\ \mathsf{exp} \\
\mathsf{AssignStmt} & ::= & \mathsf{lhs} = \mathsf{exp} \\
\mathsf{CastStmt} & ::= & \mathsf{varN}\ \textbf{?=}\ \mathsf{exp} \\
\mathsf{NewInstStmt} & ::= & \mathsf{varN} = \mathsf{pClassN}\boldsymbol{.new} \\
\mathsf{ReturnStmt} & ::= & \textbf{return} \mid \textbf{return}\ \mathsf{exp} \\
\mathsf{CallStmt} & ::= & \mathsf{call} \mid \mathsf{varN} = \mathsf{call} \\
\mathsf{Call} & ::= & \mathsf{target}\ \mathsf{.opN}\boldsymbol{(}\mathsf{exp}^{*}\boldsymbol{)} \\
\mathsf{CollStmt} & ::= & \mathsf{exp.nativeStmt} \\
\mathsf{NativeStmt} & ::= & \mathit{add}(\mathsf{exp}) \mid \mathit{del}(\mathsf{exp})
\end{array}$$

Figure 7.2 – Expressions (left) and Statements (right)

Kermeta's statements consists in either *local variable declarations* or *statements* with an actual dynamic effect. We first explain how local variables are represented before proceeding to the statements' syntax.

7.2.1 Local Variable Declarations

This formal semantics focuses on the dynamic aspects of Kermeta's AL. We operate several syntactic simplifications that do not change the dynamic semantics but allow us to simplify the presentation without any loss of generality.

The first simplification concerns the variable declarations in an operation's body, which are all shifted at the beginning before any other statement (variables can always be renamed consistently to avoid name clashes), and they are considered to scope to the entire body. This classical simplification does not affect the operation's behavior but only influences the complexity of the type-checking system (Aho, Sethi, and Ullman 1986).

A *local name* LocalN is a name whose scope is local to an operation's body: it is either a local variable VarN, or one of the operation's parameter ParamN. Additionally, we consider the special variable **self** whose value is directly attached to the considered operation. The following definition capture the local type declarations for LocalN.

Definition 7.1 (Local Type Environment)**.** *Given a metamodel* MM $\in \mathcal{M}$*, the* local (type) environment *is a function that associates to each operation in MM a function mapping local declarations to their types.*

$$\lambda_{\mathsf{MM}} : \mathsf{QOpN} \nrightarrow \mathsf{LocalN} \nrightarrow \mathsf{MType}$$

As usual, the attached metamodel MM will be omitted, since statements are always structurally attached to an operation of a class in MM. Notice also that if a operation is abstract, no local variables are defined:

$$\forall \mathsf{qop} \in \mathsf{QOpN}, \mathrm{abs}_{\mathsf{MM}}(\mathsf{qop}) \implies \mathsf{Dom}\,(\lambda_{\mathsf{MM}}(\mathsf{qop})) = \varnothing$$

7.2.2 Syntax

Starting from Kermeta's AL and removing the constructions reminded in Sec. 7.1, we obtain the core AL presented in Fig. 7.2 in the form of a BNF grammar. Using a grammar was a choice made for two main reasons: first, traditional SOS rules are usually expressed on top of BNF sentences; and second, the definition of the core

AL is more compact in this form (as compared to the original metamodel in the Manual). The non-terminal Body of Fig. 7.2 is directly linked with the metamodel's class *Body* in Fig. 5.3, which formally establishes the connection between the SL and the AL. Furthermore, the grammar presented here follows as much as possible the textual concrete syntax of Kermeta, which make it easier to retrieve the corresponding constructions.

The Body consists in a non-empty sequence of statements Stm: we already discarded from operations functions the case where an operation's body is empty (cf. definition in Sec. 6.3.5). Each statement is associated with a label $\mathsf{lab} \in \mathsf{Lab}$ that uniquely identifies the statement through the entire metamodel. Consequently, we can consider that an operation's body is represented by the sequence of its labels, given by the function $\text{labs}_{\mathsf{MM}}$. A *label encloser* function *op* associates a label to its enclosing operation in the metamodel[1].

$$\begin{aligned} \text{labs}_{\mathsf{MM}} &: \mathsf{QOpN} \longrightarrow \langle\!\langle \mathsf{Lab} \rangle\!\rangle \\ op_{\mathsf{MM}} &: \mathsf{Lab} \nrightarrow \mathsf{QOpN} \end{aligned}$$

The core AL is divided in two syntactic groups: expressions Exp are side-effect free and simply evaluate to a value; they are used inside statements Stmt that actually alter the model under execution. This simplification is not new (Aho, Sethi, and Ullman 1986; Pollet 2004; Stark, Borger, and Schmid 2001) and is convenient to clarify the semantics of OO languages.

Expressions are of four kinds: the **null** expression; *scalar* expressions ScalarExp; *instance* expressions InstExp; or *collection* expressions.

A scalar expression is either a literal Literal or an **instanceof** expression. The terminal Literal abstracts from the actual representation of literal and associated operations. An **instanceof** expression consists of an instance expression and a declared class in the form of a package name and a class name, thus reflecting in the grammar the definition of PClassN.

Instance expressions have values that designate "assignable" entities: either **self**, or a left-hand side expression. Left-hand side expressions Lhs are either a variable / parameter access, or a property access at the level of the superclass or by navigating through an instance.

Collection expressions consists in an instance expression that designates a collection, and a native expression, which is one of the following: *size*() that returns the size of a collection; *isEmpty*() that returns true iff a collection is empty; or *at*(exp) that returns an element in an ordered collection at the position given by the expression.

The syntactic set LocalN formally captures the definition of *local names* (as already used in Section 7.2.1, although it is not, strictly speaking, necessary for defining core AL's grammar.

Statements are of five kinds: *conditional*, *cast* and *assignment*, *instance creation*, *operation call* management, and *collection* statements. A *conditional* CondStmt is reduced to a conditional branching statement. An *assignment* AssignStmt assigns an expression to a left-hand side expression. A *cast* CastStmt casts an expression to a variable. An *instance creation* NewInstStmt assigns to a variable a fresh instance of a specified class, again through the form of a PClassN. A *return* ReturnStmt is either the simple keyword **return**, or this keyword used with a return expression. An *operation call* CallStmt is either a simple call, or it can return a value which is assigned to a variable. A *collection* statement CollStmt consists in a call expression (before the .) that designates a collection, on which one of the following native statement is applied: *add*(exp) or *del*(exp) respectively adds or removes an element given by the parameter expression (between parentheses).

7.2.3 Control Flow

Notice first that our core AL does not possess any structuring statement, but rather uses only one conditional branching statement CondStmt. This simplification allows us to avoid well-studied issues that traditionally occur

[1] Consider a metamodel $\mathsf{MM} = (p, c, e, \mathsf{prop}, o) \in \mathcal{M}$ where for example $o(\mathsf{pkg})(\mathsf{c})(\mathsf{op}) = (\_, \_, \_, \_, \mathsf{Body})$. From the definition in Fig. 5.5, we consider that $\mathsf{Body} = \langle\!\langle \mathsf{lab}_1, \ldots, \mathsf{lab}_n \rangle\!\rangle$. Then labs and op are reverse functions, i.e. $\mathsf{labs}(\mathsf{pkg}, \mathsf{c}, \mathsf{op}) = \mathsf{Body} \iff \forall i, \mathsf{op}(\mathsf{lab}_i) = (\mathsf{pkg}, \mathsf{c}, \mathsf{op})$.

Lab	`While`		Flattened	`If-then-else`	
	C.F	Original	Representation	Original	C.F
0	$(1, n+1)$	while(...){	if(...){	if(...){	$(1, n+1)$
1	$(2, \perp)$	stm_1	stm_1	stm_1	$(2, \perp)$
	...	...	...	...	...
n	$(0, \perp)$	stm_n	stm_n	stm_n	$(n+m+1, \perp)$
		}		}else{	
n+1	$(n+2, \perp)$	stm_{n+1}	stm_{n+1}	stm_{n+1}	$(n+2, \perp)$
					...
n+m	$(n+m+1, \perp)$	stm_{n+m}	stm_{n+m}	stm_{n+m}	$(n+m+1, \perp)$
				}	
n+m+1		stm_{n+m+1}	stm_{n+m+1}		stm_{n+m+1}

Table 7.1 – Common representation of traditional `while`/`if-then-else` statements with CondStmt.

when using imperative constructions like iterative (`while`, `do`, for) or conditional (`if-then-else`) statements (Aho, Sethi, and Ullman 1986; Pollet 2004; Stark, Borger, and Schmid 2001), such as variable scopes and nested control flows. Instead, statements contained in such constructions are flattened.

Definition 7.2 (Control Flow)**.** *The control flow function* $\mathrm{nxt}_{\mathsf{MM}}$ *attached to a metamodel MM is a function that associates to each label of the metamodel the next labels to proceed.*

$$\mathrm{nxt}_{\mathsf{MM}} : \mathsf{Lab} \longrightarrow \mathsf{Lab} \times \mathsf{Lab}_{\perp}$$

Consider a labeled statement $\mathsf{lab} : \mathsf{stmt} \in \mathsf{Stm}$. Most statements have exactly one following statement, the following one in the declaration order: in such case, $nxt(\mathsf{lab}) = (\mathsf{lab}', \perp)$. If $\mathsf{stmt} \in \mathsf{ReturnStmt}$ is a return statement, the control flow escapes from the operation and no following statement is expected: $nxt(\mathsf{lab}) = (\perp, \perp)$. If $\mathsf{stmt} \in \mathsf{CondStmt}$ is a conditional statement, two following statements are possible, depending on the value of the condition: $nxt(\mathsf{lab}) = (\mathsf{lab}', \mathsf{lab}'')$, where lab' (resp. lab'') is the statement to which to proceed if evaluated to true (resp., false).

Table 7.1 shows how our single branching statement CondStmt acts as a normal form for the classical `while` and `if-then-else`: they have a common representation but differ only in the definition of their associated control flow function nxt (columns **CF** showing nxt(lab), where lab is given in first column). Other iterative statements like `do` or `for` can syntactically be reduced to a `while` (cf. (Aho, Sethi, and Ullman 1986)).

7.2.4 Example

Figure 7.3 shows how the body of the *fire* operation is transformed into our core AL (the *accept* operation uses the same statements, namely conditionals, loops and assignments). First, all declarations are shifted (and renamed if necessary) to the beginning of the body. Second, Kermeta's loops are translated into `while` loops; and **result** assignments are replaced by corresponding **return** statements. Third, loops are flattened according to Table 7.1; and statements are labeled and the corresponding control flow is computed.

Let $\mathsf{L}_{\mathsf{fire}} \subseteq \mathsf{Lab}$ be the set of labels used in the *fire* operation. The *fire* local environment's domain is $\mathsf{Dom}\,(\lambda_{\mathsf{FSM}}(\mathsf{FSM}, \mathsf{State}, \mathsf{fire})) = \{\mathsf{trans}, \mathsf{current}, \mathsf{i}, \mathbf{self}, \mathsf{letter}\}$. If we remember that $o_{\mathsf{FSM}}(\mathsf{FSM})(\mathsf{State})(\mathsf{fire}) = (\perp, \perp, \langle\!\langle(\mathsf{letter}, (0, 1, \perp, \mathsf{String}))\rangle\!\rangle, (0, 1, \perp, \mathsf{State}), b_{\mathsf{fire}})$, then we have the following definitions:

```
var trans: oset Transition [0..*]
var current: Transition
var i: Integer
01: trans := self.out
02: current := trans.at(0)
03: i := 0
04: if(trans == null) {
05:       return null
     }else{
06:       while(i == trans.size() or
             trans.at(i).label == letter){
07:            i := i+1
          }
08:       if(current == null) {
09:            return null
          }else{
10:            return current.tgt
          }
     }
```

```
var trans: oset Transition [0..*]
var current: Transition
var i: Integer
01: trans := self.out
02: current := trans.at(0)
03: i := 0
04: if(trans == null)
05: return null

06: if(i == trans.size() or
          trans.at(i).label == letter)
07: i := i+1

08: if(current == null)
09: return null

10: return current.tgt
```

L_{fire}	nxt
01	$(02, \bot)$
02	$(03, \bot)$
03	$(04, \bot)$
04	$(05, 06)$
05	$(\bot, \bot)$
06	$(07, 08)$
07	$(06, \bot)$
08	$(09, 10)$
09	$(\bot, \bot)$
10	$(\bot, \bot)$

Figure 7.3 – The *fire* operation's body: at the left, declarations and associated initialisations are shifted at the beginning, and loops and *result* statements are transformed; in the middle, statements are flattened, according to Tab. 7.1; at the right, the control flow is explicited.

$$\begin{aligned}
\forall \mathsf{lab} \in \mathsf{L}_{\mathsf{fire}}, op(\mathsf{lab}) &= (\mathsf{FSM}, \mathsf{State}, \mathsf{fire}) \\
\lambda(\mathsf{FSM}, \mathsf{State}, \mathsf{fire})(\mathsf{trans}) &= (0, \star, \mathsf{OSet}, \mathsf{Transition}) \\
\lambda(\mathsf{FSM}, \mathsf{State}, \mathsf{fire})(\mathsf{current}) &= (1, 1, \bot, \mathsf{Transition}) \\
\lambda(\mathsf{FSM}, \mathsf{State}, \mathsf{fire})(\mathsf{i}) &= (1, 1, \bot, \mathsf{Integer}) \\
\lambda(\mathsf{FSM}, \mathsf{State}, \mathsf{fire})(\mathsf{letter}) &= (0, 1, \bot, \mathsf{String}) \\
\lambda(\mathsf{FSM}, \mathsf{State}, \mathsf{fire})(\mathbf{self}) &= (1, 1, \bot, \mathsf{State})
\end{aligned}$$

7.3 Type-Checking System

A Type-Checking system (Cardelli 2004) defines rules ensuring that statements in an operation's body are consistent with the metamodel $\mathsf{MM} \in \mathcal{M}$ and local λ_{MM} declarations.

In order to have a sound type system, the definition of Type has to be extended to take care of the **null** value, which was not part of the Structural Language. From now on, the following definitions replace the ones given in Def. 6.2:

Definition 7.3 (Extended Types). *A type in the set of (basic) types* Type *is either a primitive type, or a class name or an enumeration name declared in the scope of a package, or the special type* **Null**, *wich contains only one value,* **null**. *The extended order* $\preccurlyeq_{\mathsf{Type}}$ *is redefined for making* **Null** *specialising any other type.*

$$\begin{aligned}
\mathsf{Type} &\triangleq \mathsf{PClassN} \cup \mathsf{DataType} \cup \{\mathbf{Null}\} \\
\forall \mathsf{t}, \mathsf{t}' \in \mathsf{Type}, \mathsf{t} \preccurlyeq_{\mathsf{Type}} \mathsf{t}' &\overset{\triangle}{\iff} \mathsf{t} = \mathbf{Null} \vee \mathsf{t} \preccurlyeq_{\mathsf{Class}} \mathsf{t}' \vee \mathsf{t} \preccurlyeq_{\mathsf{Prim}} \mathsf{t}'
\end{aligned}$$

7.3.1 Expressions

A judgement "$\lambda \bullet \mathsf{qop} \vdash^{\mathsf{MM}}_{\mathsf{Exp}} \mathsf{exp} \rhd \mathsf{t}$" means that in MM, the expression $\mathsf{exp} \in \mathsf{Exp}$ appears in the body of the operation $\mathsf{qop} = (\mathsf{pkg}, \mathsf{c}, \mathsf{op}) \in \mathsf{QOpN}$ with local declarations λ, and is of type $\mathsf{t} \in \mathsf{CType}$.

The judgement $\vdash_{\mathsf{Exp}}$ is defined by structural induction. The following rules just transfer into specific rules for each kind of expression: scalar expressions should be typed by a DataType; instance expressions should be typed by a class PClassN. The **null** expression has obviously the type **Null**.

$$\mathsf{Exp} ::= \textbf{null} \mid \mathsf{scalarExp} \mid \mathsf{instExp} \mid \mathsf{collExp}$$

$$\frac{}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Exp}} \textbf{null} \rhd (\bot, \textbf{Null})}$$

$$\frac{\begin{array}{ccc}\lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Scalar}} & \mathsf{scalarExp} \rhd (\bot, \mathsf{t}) \\ \mathsf{t} & \in & \mathsf{DataType}\end{array}}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Exp}} \mathsf{scalarExp} \rhd (\bot, \mathsf{t})} \qquad \frac{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Inst}} \mathsf{instExp} \rhd \mathsf{ct} \in \mathsf{CType}}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Exp}} \mathsf{instExp} \rhd \mathsf{ct}} \qquad \frac{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Coll}} \mathsf{collExp} \rhd \mathsf{ct} \in \mathsf{CType}}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Exp}} \mathsf{collExp} \rhd \mathsf{ct}}$$

Scalar Expressions. The type of a literal expression is straightforward. The type of an **instanceof** expression is Boolean if the type of its instance expression is a class type that specialises the declared class.

$$\mathsf{ScalarExp} ::= \mathsf{literal} \mid \mathsf{instanceExp}\ \textbf{instanceof}\ \mathsf{className}$$

$$\frac{}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Scalar}} \mathsf{lit} \rhd \tau_{\mathbf{M},\mathsf{MM}}(\mathsf{lit})} \qquad \frac{\begin{array}{ccc}\lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Inst}} & \mathsf{instExp} \rhd \mathsf{c} \\ \mathsf{c} & \preccurlyeq & (\mathsf{pkgN}, \mathsf{classN})\end{array}}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Scalar}} \mathsf{instExp}\ \textbf{instanceof}\ (\mathsf{pkgN}, \mathsf{classN}) \rhd (\bot, \mathsf{Boolean})}$$

Instance Expressions. The type of **self** is the declared type in the local environment λ_{MM} in op's scope. The type of a left-hand side expression is computed by a specific judgment $\vdash_{\mathsf{Lhs}}$.

$$\mathsf{instExp} ::= \textbf{self} \mid \mathsf{lhs}$$

$$\frac{(\_, \mathsf{ct}) = \lambda(\mathsf{qop})(\textbf{self})}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Inst}} \textbf{self} \rhd \mathsf{ct}} \qquad \frac{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Lhs}} \mathsf{lhs} \rhd \mathsf{ct}}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Inst}} \mathsf{lhs} \rhd \mathsf{ct}}$$

The type of a variable/parameter access is also the declared type in λ_{MM} in op's scope.

$$\mathsf{Lhs} ::= \mathsf{varN} \mid \mathsf{paramN} \mid \ldots$$

$$\frac{\begin{array}{ccc}\mathsf{name} & \in & \mathsf{VarN} \cup \mathsf{ParamN} \\ (\_, \_, \mathsf{ct}) & = & \lambda(\mathsf{qop})(\mathsf{name})\end{array}}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Lhs}} \mathsf{name} \rhd \mathsf{ct}}$$

The type of a property access depends on the target expression. First, the class c' corresponding to the target is computed. Then, it depends of the fact that propN is ambiguous or not in the context of c' (given by from value). If not, then there must be a class c'' in the inheritance hierarchy of c that has propN in its scope, i.e either declares it or inherits it (which fact is encapsulated in π). If it is ambiguous, then there must exist a disambiguation class c'' from which the property declaration is retrieved (also through π).

$$\mathsf{Lhs} ::= \ldots \mid \textbf{super}.\mathsf{propN} \mid \mathsf{instExp.propN}$$

$$\frac{\begin{array}{rcl} \mathrm{from}(\mathsf{pkg})(\mathsf{c})(\mathsf{propN}) & = & \bot \\ \mathsf{c}' & \in & \mathrm{super}(\mathsf{pkg},\mathsf{c}) \\ \mathsf{propN} & \in & \mathsf{Dom}\,(\pi(\mathsf{pkg})(\mathsf{c}')) \\ (\_,\_,(\_,\mathsf{ct}),\_) & = & \pi(\mathsf{pkg})(\mathsf{c}')(\mathsf{propN}) \end{array}}{\lambda\bullet(\mathsf{pkg},\mathsf{c},\mathsf{op}) \vdash_{\mathsf{Lhs}} \textbf{super}.\mathsf{propN} \rhd \mathsf{ct}} \qquad \frac{\begin{array}{rcl} \lambda\bullet(\mathsf{pkg},\mathsf{c},\mathsf{op}) & \vdash_{\mathsf{Inst}} & \mathsf{instExp} \rhd \mathsf{c}' \\ \mathrm{from}(\mathsf{pkg})(\mathsf{c}')(\mathsf{propN}) & = & \bot \\ \mathsf{c}'' & \in & \mathrm{super}(\mathsf{pkg},\mathsf{c}) \\ \mathsf{propN} & \in & \mathsf{Dom}\,(\pi(\mathsf{pkg})(\mathsf{c}'')) \\ (\_,\_,(\_,\mathsf{ct}),\_) & = & \pi(\mathsf{pkg})(\mathsf{c}'')(\mathsf{propN}) \end{array}}{\lambda\bullet(\mathsf{pkg},\mathsf{c},\mathsf{op}) \vdash_{\mathsf{Lhs}} \mathsf{instExp}\ .\ \mathsf{propN} \rhd \mathsf{ct}}$$

$$\frac{\begin{array}{rcl} \mathrm{from}(\mathsf{pkg})(\mathsf{c})(\mathsf{propN}) & = & \mathsf{c}' \\ (\_,\_,(\_,\mathsf{ct}),\_) & = & \pi(\mathsf{pkg})(\mathsf{c}')(\mathsf{propN}) \end{array}}{\lambda\bullet(\mathsf{pkg},\mathsf{c},\mathsf{op}) \vdash_{\mathsf{Lhs}} \textbf{super}.\mathsf{propN} \rhd \mathsf{ct}} \qquad \frac{\begin{array}{rcl} \lambda\bullet(\mathsf{pkg},\mathsf{c},\mathsf{op}) & \vdash_{\mathsf{Inst}} & \mathsf{instExp} \rhd \mathsf{c}' \\ \mathrm{from}(\mathsf{pkg})(\mathsf{c}')(\mathsf{propN}) & = & \mathsf{c}'' \\ (\_,\_,(\_,\mathsf{ct}),\_) & = & \pi(\mathsf{pkg})(\mathsf{c}'')(\mathsf{propN}) \end{array}}{\lambda\bullet(\mathsf{pkg},\mathsf{c},\mathsf{op}) \vdash_{\mathsf{Lhs}} \mathsf{instExp}\ .\ \mathsf{propN} \rhd \mathsf{ct}}$$

Collection Expressions. Let instExp.nativeExp ∈ CollExp. First, instExp must have a collection type with an actual collection. Then, the type of *isEmpty()* is Boolean, the type of *size()* is Integer. The type of *at* is instExp basic type if the associated expression's type is Integer and instExp collection kind specialises List (i.e. must be ordered to enable indexed access).

$$\mathsf{CollExp} ::= \mathsf{exp}.\mathit{isEmpty()} \mid \mathsf{exp.size()} \mid \mathsf{exp}.\mathit{at(}\mathsf{exp}\mathit{)}$$

$$\frac{\begin{array}{rcl} \lambda\bullet\mathsf{qop} & \vdash_{\mathsf{Exp}} & \mathsf{exp} \rhd (\mathsf{C},\_) \\ \mathsf{C} & \neq & \bot \end{array}}{\lambda\bullet\mathsf{qop} \vdash_{\mathsf{Exp}} \mathsf{exp}.\mathit{isEmpty()} \rhd (\bot,\mathsf{Boolean})} \quad \frac{\begin{array}{rcl} \lambda\bullet\mathsf{qop} & \vdash_{\mathsf{Exp}} & \mathsf{exp} \rhd (\mathsf{C},\_) \\ \mathsf{C} & \neq & \bot \end{array}}{\lambda\bullet\mathsf{qop} \vdash_{\mathsf{Exp}} \mathsf{exp}.\mathit{size()} \rhd (\bot,\mathsf{Integer})} \quad \frac{\begin{array}{rcl} \lambda\bullet\mathsf{qop} & \vdash_{\mathsf{Exp}} & \mathsf{exp}' \rhd (\bot,\mathsf{Integer}) \\ \lambda\bullet\mathsf{qop} & \vdash_{\mathsf{Exp}} & \mathsf{exp} \rhd (\mathsf{C},\mathsf{t}) \\ \mathsf{C} & \neq & \bot \\ \mathsf{C} & \preccurlyeq & \mathsf{List} \end{array}}{\lambda\bullet\mathsf{qop} \vdash_{\mathsf{Exp}} \mathsf{exp}.\mathit{at(}\mathsf{exp}'\mathit{)} \rhd (\bot,\mathsf{t})}$$

7.3.2 Statements

A judgement "$\lambda\bullet\mathsf{qop} \vdash^{\mathsf{MM}}_{\mathsf{Stm}} \mathsf{stm}$" means that the statement stm ∈ Stm appears inside the body of the operation qop ∈ QOpN, and is well-formed.

A statement stm is well-formed if its inner statement stmt is well-formed and its labels are consistent, i.e. stm's execution proceeds to statements within the same operation's body.

$$\mathsf{Stm} ::= \mathit{lab:}\ \mathsf{stmt}$$

$$\frac{\begin{array}{rcl} \lambda\bullet\mathsf{qop} & \vdash_{\mathsf{Stmt}} & \mathsf{stmt} \\ \mathrm{nxt}(\mathsf{lab}) & = & (\mathsf{lab}',\mathsf{lab}'') \\ op(\mathsf{lab}') & = & \mathsf{qop} \\ op(\mathsf{lab}'') & = & \mathsf{qop} \end{array}}{\lambda\bullet\mathsf{qop} \vdash_{\mathsf{Stm}} \mathsf{lab} : \mathsf{stmt}}$$

The previous rule uses a judgement $\vdash_{\mathsf{Stmt}}$ defined by structural induction. A conditional statement is well-formed if its expression's type is Boolean.

$$\mathsf{CondStmt} ::= \mathsf{if\ exp}$$

$$\frac{\lambda\bullet\mathsf{qop} \vdash_{\mathsf{Exp}} \mathsf{exp} \rhd (\bot,\mathsf{Boolean})}{\lambda\bullet\mathsf{qop} \vdash_{\mathsf{Stmt}} \textbf{if}\ \mathsf{exp}}$$

An instance creation statement is well-formed if the created instance's type specialises the class type declaration.

$$\mathsf{NewInstStmt} ::= \mathsf{var} = \textbf{new}\ \mathsf{PClassN}$$

$$\frac{\begin{array}{rcl}\lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Exp}} & \mathsf{var} \rhd (\bot, \mathsf{c}) \\ \mathsf{c} & \in & \mathsf{PClassN} \\ \mathsf{c} & \preccurlyeq & (\mathsf{pkgN}, \mathsf{classN})\end{array}}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Stmt}} \mathsf{var} = (\mathsf{pkgN},\ \mathsf{classN}).\boldsymbol{new}}$$

A return statement with (resp. without) an expression is well-formed if it appears inside the body of an operation whose return type specialises its expression's type (resp. is void).

$$\mathsf{ReturnStmt} ::= \textbf{return} \mid \textbf{return}\ \mathsf{exp}$$

$$\frac{\text{type}(\mathsf{qop}) = (\bot, \bot, \bot)}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Stmt}} \textbf{return}} \qquad \frac{\begin{array}{rcl}\text{type}(\mathsf{qop}) & = & (\bot, \bot, \mathsf{t}) \\ \lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Exp}} & \mathsf{exp} \rhd \mathsf{t}' \\ \mathsf{t}' & \preccurlyeq & \mathsf{t}\end{array}}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Stmt}} \textbf{return}\ \mathsf{exp}}$$

Assignment and Casting statements are similar. An assigment statement is well-typed if the type of the right-hand side expression specialises the type of the left-hand side expression. A cast statement is well-typed if the types of the left-hand side and the right-hand side are comparable (i.e. they have a "common supertype" (Drey et al. 2009, §2.9.2.1).

$$\mathsf{AssignStmt} ::= \mathsf{lhs} = \mathsf{exp} \qquad \mathsf{CastStmt} ::= \mathsf{varN}\ \textbf{?=}\ \mathsf{exp}$$

$$\frac{\begin{array}{rcl}\lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Exp}} & \mathsf{lhs} \rhd \mathsf{t} \\ \lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Exp}} & \mathsf{exp} \rhd \mathsf{t}' \\ \mathsf{t}' & \preccurlyeq & \mathsf{t}\end{array}}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Stmt}} \mathsf{lhs} = \mathsf{exp}} \qquad \frac{\begin{array}{rcl}\lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Exp}} & \mathsf{varN} \rhd \mathsf{t} \\ \lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Exp}} & \mathsf{exp} \rhd \mathsf{t}' \\ \mathsf{t}' \preccurlyeq \mathsf{t} & \vee & \mathsf{t} \preccurlyeq \mathsf{t}'\end{array}}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Stmt}} \mathsf{varN}\ \textbf{?=}\ \mathsf{exp}}$$

An Operation Call statement is typed in two steps. The first step assumes another judgement $\vdash_{\mathsf{Call}}$ explained hereafter. A call without assignment is well-typed if the call expression is well-typed and the operation has no return type; and a call with assignment is well-typed if the call expression is well-typed and the return type of the operation specialises the type of the assigned variable.

$$\mathsf{CallStmt} ::= \mathsf{target.opN(exp^*)} \qquad \mathsf{CallStmt} ::= \mathsf{varN} = \mathsf{target.opN(exp^*)}$$

$$\frac{\begin{array}{rcl}\lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Call}} & \mathsf{target.opN(exp^*)} \\ \text{type}(\mathsf{qop}) & = & \bot\end{array}}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Stmt}} \mathsf{target.opN(exp^*)}} \qquad \frac{\begin{array}{rcl}\lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Call}} & \mathsf{target.opN(exp^*)} \\ \lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Exp}} & \mathsf{varN} \rhd \mathsf{t} \\ \text{type}(\mathsf{opN}) & \preccurlyeq & \mathsf{t}\end{array}}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Stmt}} \mathsf{varN} = \mathsf{target.opN(exp^*)}}$$

The following set of rules deals with the call itself (for the judgement $\vdash_{\mathsf{Call}}$). First, the type of the target expression should denote a class (which is obviously the case for **super**). Second, all expressions for effective parameters are also well-typed. Third, the type-checking follows the same schema as for property access because it depends on the target and the possible multiply inherited method name: if the method name is ambiguous, then the method call is well-typed if there exists a method with the same name in the inheritance hierarchy of the disambiguated class; if it is not, then the method call is well-typed if there exists one class in the inheritance hierarchy that declares a method with the same name. The class from which the method name is looked for depends on the target: if the target is **super**, then the method name is looked for from the superclass; otherwise, it is looked from the class to which the instance target is typed. Finally, the types of the effective parameters expressions must specialise the formal parameters' types (in order).

$$\mathsf{Call} ::= \mathsf{instExp.opN(exp^*)}$$

$$\frac{\begin{array}{rcl}\forall \mathsf{i}, \lambda \bullet (\mathsf{pkg},\mathsf{c},\mathsf{op}) & \vdash_{\mathsf{Exp}} & \mathsf{exp_i} \rhd \mathsf{t_i}\\ \lambda \bullet (\mathsf{pkg},\mathsf{c},\mathsf{op}) & \vdash_{\mathsf{Exp}} & \mathsf{instExp} \rhd \mathsf{c}'\\ \mathrm{from}(\mathsf{pkg},\mathsf{c}',\mathsf{opN}) & = & \perp\\ \mathsf{opN} & \in & \mathsf{Dom}\,(\omega(\mathsf{pkg})(\mathsf{c}'))\\ \mathrm{partypes}(\mathsf{pkg},\mathsf{c}',\mathsf{opN}) & = & \langle\!\langle(\_,\_,\mathsf{ct}'_1),\ldots,(\_,\_,\mathsf{ct}'_n)\rangle\!\rangle\\ \forall i,\ \mathsf{ct_i} & \preccurlyeq & \mathsf{ct}'_i\end{array}}{\lambda \bullet (\mathsf{pkg},\mathsf{c},\mathsf{op}) \vdash_{\mathsf{Call}} \mathsf{instExp.opN(exp_1,\ldots,exp_n)}}
\qquad
\frac{\begin{array}{rcl}\forall \mathsf{i}, \lambda \bullet (\mathsf{pkg},\mathsf{c},\mathsf{op}) & \vdash_{\mathsf{Exp}} & \mathsf{exp_i} \rhd \mathsf{t_i}\\ \lambda \bullet (\mathsf{pkg},\mathsf{c},\mathsf{op}) & \vdash_{\mathsf{Exp}} & \mathsf{instExp} \rhd \mathsf{c}'\\ \mathrm{from}(\mathsf{pkg},\mathsf{c}',\mathsf{opN}) & = & \mathsf{c}''\\ \mathsf{opN} & \in & \mathsf{Dom}\,(\omega(\mathsf{pkg})(\mathsf{c}''))\\ \mathrm{partypes}(\mathsf{pkg},\mathsf{c}',\mathsf{opN}) & = & \langle\!\langle(\_,\_,\mathsf{ct}'_1),\ldots,(\_,\_,\mathsf{ct}'_n)\rangle\!\rangle\\ \forall i,\ \mathsf{ct_i} & \preccurlyeq & \mathsf{ct}'_i\end{array}}{\lambda \bullet (\mathsf{pkg},\mathsf{class},\mathsf{op}) \vdash_{\mathsf{Call}} \mathsf{instExp.opN(exp_1,\ldots,exp_n)}}$$

$$\mathsf{Call} ::= \mathbf{super}\mathsf{.opN(exp^*)}$$

$$\frac{\begin{array}{rcl}\forall \mathsf{i}, \lambda \bullet (\mathsf{pkg},\mathsf{c},\mathsf{op}) & \vdash_{\mathsf{Exp}} & \mathsf{exp_i} \rhd \mathsf{ct_i}\\ \mathrm{from}(\mathsf{pkg},\mathsf{c},\mathsf{opN}) & = & \perp\\ \mathsf{c}' & \in & \mathrm{super}(\mathsf{pkg},\mathsf{c})\\ \mathsf{opN} & \in & \mathsf{Dom}\,(\omega(\mathsf{pkg})(\mathsf{c}'))\\ \mathrm{partypes}(\mathsf{pkg},\mathsf{c}',\mathsf{opN}) & = & \langle\!\langle(\_,\_,\mathsf{ct}'_1),\ldots,(\_,\_,\mathsf{ct}'_n)\rangle\!\rangle\\ \forall i,\ \mathsf{ct_i} & \preccurlyeq & \mathsf{ct}'_i\end{array}}{\lambda \bullet (\mathsf{pkg},\mathsf{c},\mathsf{op}) \vdash_{\mathsf{Call}} \mathbf{super}\mathsf{.opN(exp_1,\ldots,exp_n)}}
\qquad
\frac{\begin{array}{rcl}\forall \mathsf{i}, \lambda \bullet (\mathsf{pkg},\mathsf{c},\mathsf{op}) & \vdash_{\mathsf{Exp}} & \mathsf{exp_i} \rhd \mathsf{t_i}\\ \mathrm{from}(\mathsf{pkg},\mathsf{c},\mathsf{opN}) & = & \mathsf{c}'\\ \mathsf{opN} & \in & \mathsf{Dom}\,(\omega(\mathsf{pkg})(\mathsf{c}'))\\ \mathrm{partypes}(\mathsf{pkg},\mathsf{c}',\mathsf{opN}) & = & \langle\!\langle(\_,\_,\mathsf{ct}'_1),\ldots,(\_,\_,\mathsf{ct}'_n)\rangle\!\rangle\\ \forall i,\ \mathsf{ct_i} & \preccurlyeq & \mathsf{ct}'_i\end{array}}{\lambda \bullet (\mathsf{pkg},\mathsf{c},\mathsf{op}) \vdash_{\mathsf{Call}} \mathbf{super}\mathsf{.opN(exp_1,\ldots,exp_n)}}$$

The two collection statements are typed the same way. They are well-formed if their application expression's type has an actual collection kind, and if their parameter expression's basic type specialises the application expression's one.

$$\mathsf{CollStmt} ::= \mathsf{exp.add(exp')} \mid \mathsf{exp.del(exp')}$$

$$\frac{\begin{array}{rcl}\lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Exp}} & \mathsf{exp} \rhd (\mathsf{C},\mathsf{t})\\ \lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Exp}} & \mathsf{exp}' \rhd (\perp,\mathsf{t}')\\ \mathsf{C} & \neq & \perp\\ \mathsf{t}' & \preccurlyeq & \mathsf{t}\end{array}}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Stmt}} \mathsf{exp.\_\_\_(exp')}}$$

7.4 Semantics

A Structural Operational Semantics Sos specifies the semantics by means of a labeled transition system describing the abstract execution of a transformation. A *semantic domain* interprets the syntactic constructions, and is further enriched with the necessary information to describe the dynamics of the execution, resulting in an *execution state*, or *configuration* that represents the state of the transition system. Then, the transitions are simply rewriting rules between configurations (Winskel 1993).

7.4.1 Semantic Domain and Operations

The *semantic domain* is formally a set $(\mathbb{D}, \{op_i\}_{i \in I})$ that gives the meaning of all syntactic constructions of the language, i.e. a way to unambiguously interpret what the language describes. It generally comes with operations that facilitate the manipulation of this domain's element.

Definition 7.4 (Semantic Domain — Target)**.** *The semantic domain* $\mathbb{D}$ *is the set of pairs consisting of a model and a local (store) environment. A local store environment* $\mathbb{L}$ *is a function mapping local names to values. A* target *is either a local name, or a pair consisting of an object and a property name.*

$$\begin{array}{rcl}\mathbb{L} & \triangleq & \mathsf{LocalN} \nrightarrow \mathbb{V}\\ \mathbb{D} & \triangleq & \mathbb{M} \times \mathbb{L}\\ \mathbb{T} & \triangleq & \mathsf{LocalN} \cup (\mathbb{O} \times \mathsf{PropN})\end{array}$$

$$\begin{array}{rcl} \mathfrak{Default}^{\mathsf{MM}}_{\mathsf{Coll}} : \mathsf{Collection} & \longrightarrow & \mathbb{V} \\ \mathsf{Bag} & \mapsto & [] \\ \mathsf{Set} & \mapsto & \varnothing \\ \mathsf{List} & \mapsto & \langle\,\rangle \\ \mathsf{OSet} & \mapsto & \langle\!\langle\,\rangle\!\rangle \end{array} \qquad \begin{array}{rcl} \mathfrak{Default}^{\mathsf{MM}}_{\mathsf{Prim}} : \mathsf{PrimType} & \longrightarrow & \mathbb{V} \\ \mathsf{Boolean} & \mapsto & \bot \\ \mathsf{Integer} & \mapsto & 0 \\ \mathsf{Real} & \mapsto & 0.0 \\ \mathsf{String} & \mapsto & \text{``''} \end{array}$$

$$\begin{array}{rcl} \mathfrak{Default}^{\mathsf{MM}}_{\mathsf{Enum}} : \mathsf{PEnumN} & \longrightarrow & \mathbb{V} \\ (\mathsf{pkg}, \mathsf{enum}) & \mapsto & \mathsf{fst}(e(\mathsf{pkg})(\mathsf{enum})) \end{array} \qquad \begin{array}{rcl} \mathfrak{Default}^{\mathsf{MM}}_{\mathsf{Class}} : \mathsf{PClassN} & \longrightarrow & \mathbb{V} \\ (\mathsf{pkg}, \mathsf{class}) & \mapsto & \mathbf{null} \end{array}$$

Figure 7.4 – Functions computing default values for all collection types

A *target* $t \in \mathbb{T}$ designates a element in the semantic domain that carries a value: it is either a local name, stored in the local environment; or a property within an object, stored inside the model. We then use a functional notation to access transparently these elements from the semantic domain: if $d = (\mathsf{M}, l) \in \mathbb{D}$ is an element of the semantic domain, we note $d(t)$ the value stored for t, i.e. $d(t) \triangleq l(t)$ if $t \in \mathsf{LocalN}$, and $d(t) \triangleq \sigma^o_{\mathsf{M}}(\mathsf{p})$ if $t = (o, \mathsf{p}) \in \mathbb{O} \times \mathsf{PropN}$. Similarly, we note $d[t \mapsto v]$ the update of t by v in d, i.e. $d[t \mapsto v] \triangleq l[t \mapsto v]$ if $t \in \mathsf{LocalN}$, and $d[t \mapsto v] \triangleq \sigma^o_{\mathsf{M}}[p \mapsto v]$ if $t = (o, \mathsf{p}) \in \mathbb{O} \times \mathsf{PropN}$.

7.4.1.1 Default Values and Instances

When calling an operation (i.e. executing a CallStmt statement), it is necessary to build a new environment where local variables are associated with default values. Similarly, when creating a new instance (i.e. executing a NewInstStmt statement), it is necessary to build a "new" valid object of the correct type.

The function $\mathfrak{Default}_{\mathsf{MM}}$ associates to each collection type the corresponding default value, by following the structural definition of CType (and as usual, we omit to mention the concerned metamodel MM when clear from context):

$$\begin{array}{rcl} \mathfrak{Default}_{\mathsf{MM}} : \mathsf{CType} & \longrightarrow & \mathbb{V} \\ (\bot, \mathsf{pt}) & \mapsto & \mathfrak{Default}^{\mathsf{MM}}_{\mathsf{Prim}}(\mathsf{pt}) \text{ if } \mathsf{pt} \in \mathsf{PrimType} \\ (\bot, \mathsf{e}) & \mapsto & \mathfrak{Default}^{\mathsf{MM}}_{\mathsf{Enum}}(\mathsf{e}) \text{ if } \mathsf{e} \in \mathsf{PEnumN} \\ (\bot, \mathsf{c}) & \mapsto & \mathfrak{Default}^{\mathsf{MM}}_{\mathsf{Class}}(\bot, \mathsf{c}) \text{ if } \mathsf{c} \in \mathsf{PClassN} \\ (\mathsf{C}, \_) & \mapsto & \mathfrak{Default}^{\mathsf{MM}}_{\mathsf{Coll}}(\mathsf{C}) \text{ if } \mathsf{C} \in \mathsf{Collection} \end{array}$$

The definition distinguishes the cases when the collection type has a collection or not. If there is a collection, the default value is simply the corresponding "empty" collection (emty bag for Bag, empty set for Set and so on) independently from the basic type. If there is no collection, then it depends on the basic type. For *primitive types*, the default value is predefined independently from any metamodel: it is the false value for Boolean, the zero value for Integer, and so on; for *enumeration types*, the default value is the first enumeration literal declared for this enumeration; for *class types*, the default value for a class is simply the **null** value. Figure 7.4 gives the definitions of all intermediary default functions.

The function $\mathfrak{Initialise}_{\mathsf{MM}}$ builds a fresh "initial" instance, valid for a given class type $(\mathsf{pkg}, \mathsf{c}) \in \mathsf{PClassN}$: its type is $(\mathsf{pkg}, \mathsf{c})$ and its state is a function s_0 that associates to each accessible property its default value.

$$\begin{array}{rcl} \mathfrak{Initialise}_{\mathsf{MM}} : \mathsf{PClassN} & \longrightarrow & \mathsf{PClassN} \times \mathbb{State} \\ (\mathsf{pkg}, \mathsf{c}) & \mapsto & ((\mathsf{pkg}, \mathsf{c}), s_0) \\ \text{where} & & \left\{ \begin{array}{rcl} \mathsf{Dom}\,(s_0) & = & \mathsf{Dom}\,(\pi(\mathsf{pkg})(\mathsf{c})) \\ s_0(\mathsf{pkg}, \mathsf{c}, \mathsf{propN}) & = & \mathfrak{Default}_{\mathsf{MM}}(\text{type}(\pi(\mathsf{pkg}, \mathsf{c}, \mathsf{propN}))) \end{array} \right. \end{array}$$

7.4.1.2 Operation Call Initialisation

Calling an operation requires to build a new local environment in order to execute the called operation's body. The function $\mathfrak{Start}$ has as result a domain that corresponds to the starting state of a given operation: it builds a "new" local store environment and keeps the old model.

$$\begin{array}{rcl} \mathfrak{Start}\colon \mathsf{QOpN} \longrightarrow \mathbb{V} \times \langle \mathbb{V} \rangle \nrightarrow \mathbb{D} & \nrightarrow & \mathbb{D} \\ \mathfrak{Start}(\mathsf{pkg}, \mathsf{c}, \mathsf{op})(v_{\mathbf{self}}, \langle v_1, \ldots, v_n \rangle)(m, l) & \mapsto & (m, l') \end{array}$$

where $\mathsf{Dom}\,(l') = \mathsf{Dom}\,(\lambda(\mathsf{pkg}, \mathsf{c}, \mathsf{op}))$ and

$$l' = \left\{ \begin{array}{rcl} \mathsf{this} & \mapsto & v_{\mathsf{this}} \\ \mathsf{p_i} & \mapsto & v_i \ \forall \mathsf{p_i} \in \text{parnames}(\mathsf{pkg}, \mathsf{c}, \mathsf{op}) \\ \mathsf{varN} & \mapsto & \mathfrak{Default}(\mathsf{ct}) \text{ with } \lambda((\mathsf{pkg}, \mathsf{c}, \mathsf{varN})) = (\_, \_, \mathsf{ct}) \end{array} \right\}$$

Basically, the new local store environment respects the local declarations of the operation, and maps parameters to the corresponding value (in the order of their declaration) and variables to their default value.

7.4.1.3 Expression / Target Evaluation

As most of the statements are composed of expressions, it is necessary to provide a mechanism to properly evaluate them. Recall first that an expression is either the **null** expression, a scalar expression or an instance expression. Since expressions are designed to be side-effect free, it is possible to define a function that computes the associated value of an expression, when evaluated in the context of an model.

We first assume the existence of two functions whose effect is abstracted from any particular implementation that directly depends on the execution platform: the function $[\![\bullet]\!]_{\mathrm{Lit}}$ computes the value of literals and associated operators; the function $[\![\bullet]\!]_{\mathrm{Conv}}$ adequately converts primitive types between each others (e.g. $[\![\mathsf{Real}]\!]_{\mathrm{Conv}}(1)$ converts the integer value 1 into the corresponding real value 1.0).

$$\begin{array}{rcl} [\![\bullet]\!]_{\mathrm{Lit}} & : & \mathsf{Literal} \nrightarrow \mathbb{V} \\ [\![\bullet]\!]_{\mathrm{Conv}} & : & \mathsf{PrimType} \longrightarrow \mathbb{V} \nrightarrow \mathbb{V} \end{array}$$

The function $[\![\bullet]\!]_{\mathrm{Lit}}$ is undefined if one of its parameters is not properly initialised or undefined. Consider as an example, the addition operator: it is defined through $[\![\bullet]\!]_{\mathrm{Lit}}$ to correctly handle operands of compatible types (e.g. adding an integer value with a real value returns a real value). These definitions are classic and well-known, but cumbersome to be fully defined.

Let us consider now an expression taking place in a statement $\mathsf{stm} \in \mathsf{Stm}$ inside an operation $\mathsf{qop} \in \mathsf{QOpN}$. Notice first that expressions and statements are supposed to be well-typed. From the grammar defining expressions in Fig. 7.2, we notice that evaluating expressions requires determining the target of an instance expression. As a consequence, the function $[\![\bullet]\!]$ for evaluating expressions' values and the function $[\![\bullet]\!]_{\mathrm{T}}$ for evaluating instance expressions' target are mutually recursive.

The target of a local name (either a variable or a parameter name, or **self**) is the local name itself. The target of property access is defined only if its associated target denotes a valid instance in the domain. In this case, the target is composed of the object corresponding to the instance expression's target (either **super** or

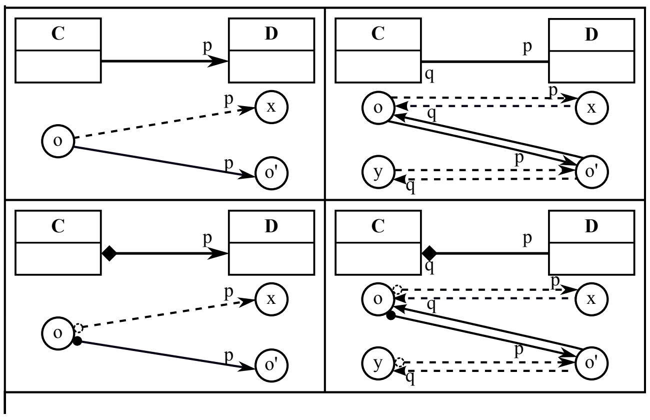


Figure 7.5 – Assignment of objects in a reference. Dashed/plain arrows represent the situation before/after assignment. Plain circles denote current object's containers; empty ones a container's reset.

a general instExp) and of the property given by the expression. The class where the property is declared is computed based on the class returned by the type-checking of the target expression.

$$
\begin{array}{rcl}
[\![\bullet]\!]_{\mathrm{T}} : \mathsf{InstExp} \longrightarrow \mathbb{D} & \nrightarrow & \mathbb{T} \\
[\![\mathbf{self}]\!]_{\mathrm{T}}(d) & \mapsto & \mathbf{self} \\
[\![\mathsf{varN}]\!]_{\mathrm{T}}(d) & \mapsto & \mathsf{varN} \\
[\![\mathsf{paramN}]\!]_{\mathrm{T}}(d) & \mapsto & \mathsf{paramN} \\
[\![\mathbf{super}.\mathsf{propN}]\!]_{\mathrm{T}}(d) & \mapsto & (o, \mathrm{decl}((\mathsf{pkg}, \mathsf{c}'), \mathsf{propN})) \text{ if } o = d(\mathbf{self}) \\
 & & \text{ and } \tau \bullet \mathsf{qop} \vdash_{\mathsf{Exp}} \mathbf{super}.\mathsf{propN} \rhd (\mathsf{pkg}, \mathsf{c}') \\
[\![\mathbf{instExp}.\mathsf{propN}]\!]_{\mathrm{T}}(d) & \mapsto & (o, \mathrm{decl}((\mathsf{pkg}, \mathsf{c}'), \mathsf{propN})) \text{ if } o = [\![\mathsf{instExp}]\!](d) \\
 & & \text{ and } \tau \bullet \mathsf{qop} \vdash_{\mathsf{Exp}} \mathsf{instExp}.\mathsf{propN} \rhd (\mathsf{pkg}, \mathsf{c}')
\end{array}
$$

The evaluation of the **null** expression and of literal expressions is straightforward, considering the dedicated function $[\![\bullet]\!]_{\mathrm{Lit}}$. The value of the **instanceof** test depends on the value of its instance expression: it is true if the type of the instance expression specialises the class declaration. Notice that if this value is undefined, then the result is also undefined. The value of an instance expression is the value associated in the domain with the instance expression's target.

$$
\begin{array}{rcl}
[\![\bullet]\!] : \mathsf{Exp} \longrightarrow \mathbb{D} & \nrightarrow & \mathbb{V} \\
[\![\mathbf{null}]\!](d) & \mapsto & \mathbf{null} \\
[\![\mathsf{lit}]\!](d) & \mapsto & [\![\mathsf{lit}]\!]_{\mathrm{Lit}} \\
[\![\mathsf{instExp}\ \mathbf{instanceof}\ (\mathsf{pkg}\ \mathsf{c})]\!](d) & \mapsto & \mathrm{type}([\![\mathsf{instExp}]\!](d)) \preccurlyeq (\mathsf{pkg}, \mathsf{c}) \\
[\![\mathsf{instExp}]\!](d) & \mapsto & d([\![\mathsf{instExp}]\!]_{\mathrm{T}}(d))
\end{array}
$$

7.4.1.4 Updating

The assignment statement is crucial to the behavior of the AL: it ensures that object integrity is preserved during updating. Since assignments can transparently update a variable, a reference, an association or a containment, its behavior must ensure the global consistency and the containment uniqueness property of models.

Suppose an assignment statement $\mathsf{lhs} = \mathsf{exp}$ evaluated in a domain $d \in \mathbb{D}$, for which expression exp evaluates to the value $v \in \mathbb{V}$, and left-hand side (LHS) lhs refers to the target $t \in \mathbb{T}\mathsf{arget}$ whose type is type (formally speaking, $[\![\mathsf{exp}]\!](d) = v$, $[\![\mathsf{lhs}]\!]_{\mathrm{T}}(d) = t$ and $\lambda \bullet \mathsf{op} \vdash_{\mathsf{Exp}} \mathsf{lhs} \rhd \mathsf{type}$). The effect of the assignment depends on both the *type* and the *nature* of the LHS involved (Fleurey 2006, Appendix A).

(i) $\mathsf{type} \in \mathsf{DataType}$ If the target's type is a $\mathsf{DataType}$, it does not deal with containment and t's value is simply replaced by v after properly converting it in case of numerical values.

(ii) $\mathsf{type} \notin \mathsf{DataType}$ If not, then t represents an object o and there is two cases:

- **(ii-1)** $t \in \mathsf{LocalN}$ either t refers to a local name and the assignment has the usual effect of replacing the current object's value by the the object denoted by v;
- **(ii-2)** $t = (o, \mathsf{p}) \in \mathbb{O} \times \mathsf{PropN}$ is a property access and it depends on the collection nature of p. The case without collection is depicted in Fig. 7.5: in this case, v is assigned to t, the container of the previous object $x \triangleq d(t)$ pointed by o is reset; and in case of bidirectional reference, the opposite property q is set to o to preserve consistency. If there is a collection, then this process is repeated for each object within the collection.

The definition of the assignment function $[\![\bullet, \bullet]\!]$ reflects these remarks and addresses the situation with collection values: here, the update is done on each target object t' of all objects contained within the collection value v, i.e. $t' \triangleq (o', \mathrm{opp}(p))\ \forall o' \in \mathrm{objs}(v)$, where the function $\mathrm{objs} : \mathbb{V} \rightarrow \wp(\mathbb{O})$ retrieves all objects contained in a collection value.

$$
\begin{array}{rcl}
[\![\bullet, \bullet]\!] : \mathsf{Lhs} \times \mathsf{Exp} & \nrightarrow & \mathbb{D} \longrightarrow \mathbb{D} \\
(\mathsf{lhs}, \mathsf{exp})(d) & \mapsto & \begin{cases} d[t \mapsto [\![\mathsf{t}, v]\!]_{\mathrm{Conv}}] & \text{if (i)} \\ d[t \mapsto (v)] & \text{if (ii} - 1) \\ d\begin{bmatrix} t & \mapsto & v \\ t' & \mapsto & (\_, o) \\ x & \mapsto & (\_, \bot) \end{bmatrix} & \text{if (ii} - 2) \end{cases}
\end{array}
$$

$$
\begin{array}{rcl}
\mathrm{objs} : \mathbb{V} & \longrightarrow & \wp(\mathbb{O}) \\
v & \mapsto & \begin{cases} v & \text{if } \mathrm{type}(v) \in (\_, \mathsf{PClassN}) \\ \varnothing & \text{otherwise} \end{cases}
\end{array}
$$

7.4.2 Configuration

A *configuration* $\gamma \in \Gamma$ consists of the label denoting the statement under execution, a stack storing the information between operation calls, and the semantic domain. A *stack* is a sequence whose elements comprise the label where to resume after completing the execution of a call, the local environment of the call, and eventually the variable to which the result of the call needs to be assigned.

$$
\begin{array}{rcl}
\Gamma & \triangleq & \mathsf{Lab} \times \langle \mathbb{E}\mathsf{nv} \rangle \times \mathbb{D} \\
\mathbb{E}\mathsf{nv} & \triangleq & (\mathsf{Lab} \times \mathbb{L}) \cup (\mathsf{Lab} \times \mathbb{L} \times \mathsf{VarN})
\end{array}
$$

Suppose now that a Kermeta transformation is launched from the platform, i.e. a **main** operation $\mathsf{op}_{\mathsf{main}}$ inside a **main** class $\mathsf{C}_{\mathsf{main}} = (\mathsf{pkg}_{\mathsf{main}}, \mathsf{c}_{\mathsf{main}})$ (cf. Appendix A.2) is called with correct effective parameter value list (v_{param}), where $\mathsf{param} \in \mathsf{Dom}\,(\lambda(\mathsf{pkg}_{\mathsf{main}}, \mathsf{c}_{\mathsf{main}}, \mathsf{op}_{\mathsf{main}}))$. From this information, we build an *initial* configuration $\gamma_0 \in \Gamma$ that starts the execution, which consists of the following: the first label of $\mathsf{op}_{\mathsf{main}}$'s (non-empty) body, an empty stack environment, and a domain where the model m_0 contains only one object of type $\mathsf{C}_{\mathsf{main}}$ for which all properties are initialised to their default value, and a local environment where each parameter param is bound to the corresponding value v_{param} and each variable to its default value.

$$\gamma_0 = (\mathsf{lab}_0, \langle\,\rangle, (m_0, l_0)) \text{ with } \left\{ \begin{array}{rcl} m_0 & = & \mathfrak{Initialise}(\mathsf{pkg}_{\mathsf{main}}, \mathsf{c}_{\mathsf{main}}) \\ \forall \mathsf{param} \in \mathsf{Dom}\,(l_0), l_0(\mathsf{param}) & = & v_{\mathsf{param}} \\ \forall \mathsf{var} \in \mathsf{Dom}\,(l_0), l_0(\mathsf{var}) & = & \mathfrak{Default}(\mathsf{ct}_{\mathsf{var}}) \\ \text{with } \lambda(\mathsf{pkg}_{\mathsf{main}}, \mathsf{c}_{\mathsf{main}}, \mathsf{op}_{\mathsf{main}})(\mathsf{var}) & = & (\_, \_, \mathsf{ct}_{\mathsf{var}}) \\ \langle\!\langle \mathsf{lab}_0, \ldots, \mathsf{lab}_\mathsf{n} \rangle\!\rangle & = & \mathrm{labs}((\mathsf{pkg}, \mathsf{c}, \mathsf{op})) \end{array} \right.$$

7.4.3 Semantic Rules

The rewriting system defining the Sos for Kermeta takes the form of a set of rewriting rules as below:

$$\frac{C}{\gamma \xrightarrow{\mathsf{stm}} \gamma'}$$

It means that the configuration $\gamma \in \Gamma$ is rewritten in γ' when executing the statement $\mathsf{stm} \in \mathsf{Stm}$ under the (possibly empty) set of conditions C. We consider that all conditions in C are performed atomically before another semantic rule can fire. Notice that inside the rules, we explicitly recall the label to bind the statement with the control flow. From the initial configuration γ_0, the rewriting system proceeds infinitely, or stops when there is no more statements to execute[2].

7.4.3.1 Conditional Statement

A conditional statement CondStmt only changes the label to the adequate one, according to the boolean value of its expression.

$$\frac{\begin{array}{rcl} v & = & [\![\mathsf{exp}]\!](d) \\ (\mathsf{lab}', \mathsf{lab}'') & = & \mathrm{nxt}(\mathsf{lab}) \\ v & \Longrightarrow & \mathsf{lab}_{\mathsf{res}} = \mathsf{lab}' \\ \neg v & \Longrightarrow & \mathsf{lab}_{\mathsf{res}} = \mathsf{lab}'' \end{array}}{(\mathsf{lab}, S, d) \xrightarrow{\mathsf{lab:\ if\ exp}} (\mathsf{lab}_{\mathsf{res}}, S, d)}$$

7.4.3.2 Instance Creation Statement

An instance creation statement NewInstStmt adds to the model a fresh initial object, and associates it to the variable in the local environment. The execution proceeds to the next label.

$$\frac{\begin{array}{rcl} o & \notin & \mathsf{Dom}\,(m) \\ m' & = & m \cup (o \mapsto \mathfrak{Initialise}(\mathsf{pkg}, \mathsf{c})) \\ l' & = & l[\mathsf{var} \mapsto o] \\ (\mathsf{lab}', \_) & = & \mathrm{nxt}(\mathsf{lab}) \end{array}}{(\mathsf{lab}, S, (m, l)) \xrightarrow{\mathsf{lab:\ var} = \mathbf{new}\ (\mathsf{pkg\ c})} (\mathsf{lab}', S, (m', l'))}$$

7.4.3.3 Assignment and Cast Statements

An assignment AssignStmt has to preserve the integrity of objects. Since we already defined a function for this purpose, the rule simply applies it and proceeds to the next label.

$$\frac{\begin{array}{rcl} (m', l') & = & [\![\mathsf{lhs}, \mathsf{exp}]\!](m, l) \\ (\mathsf{lab}', \_) & = & \mathrm{nxt}(\mathsf{lab}) \end{array}}{(\mathsf{lab}, S, (m, l)) \xrightarrow{\mathsf{lab:\ lhs} = \mathsf{exp}} (\mathsf{lab}', S, (m', l'))}$$

[2]Actually, the rewriting system can stop because no rule fires: this is not supposed to happen for an Sos, whose rules are triggered by the statements of an Abstract Syntax Tree.

A cast statement CastStmt essentially behaves like an assignment: it assigns the right expression to the variable, except that it has to carefully handle the dynamic types. In the case of primitive values, a cast statement simply behaves as a conversion. In the case of instances, the right-hand side's dynamic type must specialise the left-hand side's static type.

$$\frac{\begin{array}{rcl} \mathsf{qop} & = & \mathrm{op}(\mathsf{lab}) \\ \lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Exp}} & \mathsf{varN} \rhd \mathsf{t} \in (\bot, \mathsf{PrimType}) \\ v & = & [\![\mathsf{exp}]\!](d) \\ d' & = & [\![\mathsf{varN}, [\![\mathsf{t}]\!]_{\mathrm{Conv}}(v)]\!](d) \end{array}}{(\mathsf{lab}, S, d) \xrightarrow{\mathsf{lab:\ varN?{=}exp}} (\mathsf{lab}', S, d')} \qquad \frac{\begin{array}{rcl} \mathsf{qop} & = & \mathrm{op}(\mathsf{lab}) \\ \lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Exp}} & \mathsf{varN} \rhd \mathsf{t} \in (\bot, \mathsf{PClassN}) \\ v & = & [\![\mathsf{exp}]\!](d) \\ \tau(v) & \preccurlyeq & \mathsf{t} \\ d' & = & [\![\mathsf{varN}, v]\!](d) \end{array}}{(\mathsf{lab}, S, d) \xrightarrow{\mathsf{lab:\ varN?{=}exp}} (\mathsf{lab}', S, d')}$$

7.4.3.4 Operation Management

We include in this Section three kind of statements: the operation calls, the return statement and the collection statements. The two first manipulate the stack, whereas the last only delegates the execution to predefined operations of the semantic domain.

A return statement ReturnStmt proceeds to the label stored in the top element of the stack and changes the current local environment with the stored one, then removes the top element. If the return statement has an expression, it is evaluated in the context of the stored local environment and assigned to the variable stored in the top element.

$$\frac{s = (\mathsf{lab}', l')}{(\mathsf{lab}, s :: S, (m, l)) \xrightarrow{\mathsf{lab:}\ \textbf{return}} (\mathsf{lab}', S, (m, l'))} \qquad \frac{\begin{array}{rcl} s & = & (\mathsf{lab}', l', \mathsf{var}) \\ (m', l'') & = & [\![\mathsf{var}, \mathsf{exp}]\!](m, l') \end{array}}{(\mathsf{lab}, s :: S, (m, l)) \xrightarrow{\mathsf{lab:}\ \textbf{return}\ \mathsf{exp}} (\mathsf{lab}', S, (m', l''))}$$

An operation call without assignment of the form $\mathsf{target.op}(\mathsf{exp}_1, \ldots, \mathsf{exp}_n)$ proceeds as follows. First, the target's value is computed and ensured by the type-checking system to denote an instance. Then, all effective parameters' values are computed. Then, the operation's body is looked up, based on the dynamic type of the target. The *Start* function then builds the new domain based on the parameters values and the old domain. The label where to continue the execution is saved in the stack as well as the current environment.

$$\frac{\begin{array}{rcl} v_{\mathsf{this}} & = & [\![\mathsf{target}]\!](m, l) \\ \forall i,\ v_i & = & [\![\mathsf{exp_i}]\!](m, l) \\ (\bot, (\mathsf{pkg}, \mathsf{c})) & = & \tau(v_{\mathsf{this}})(m, l) \\ (\_, params, \_, b) & = & \omega(\mathsf{pkg})(\mathsf{c})(\mathsf{op}) \\ (m', l') & = & \mathfrak{Start}(\mathsf{pkg}, \mathsf{c}, \mathsf{op})(v_{\mathsf{this}}, \langle\!\langle v_1, \ldots, v_n \rangle\!\rangle)(m, l) \\ (m', l') & = & [\![\mathsf{lhs}, \mathsf{exp}]\!](m, l) \\ \langle\!\langle \mathsf{lab}_0, \ldots \rangle\!\rangle & = & \mathrm{labs}(\mathsf{pkg}, \mathsf{c}, \mathsf{op}) \\ (\mathsf{lab}', \_) & = & \mathrm{nxt}(\mathsf{lab}) \\ (\mathsf{lab}', l) & = & s_0 \end{array}}{(\mathsf{lab}, S, (m, l)) \xrightarrow{\mathsf{lab:\ target.op(exp_1, \ldots, exp_n)}} (\mathsf{lab}_0, s_0 :: S, (m', l'))}$$

An operation call with assignment of the form $\mathsf{varN} = \mathsf{target.op}(\mathsf{exp}_1, \ldots, \mathsf{exp}_n)$ proceeds the same as previously, except that it has to save the variable into the store.

$$\frac{\begin{array}{rcl} v_{\text{this}} & = & [\![\mathsf{target}]\!](m,l) \\ \forall i,\ v_i & = & [\![\mathsf{exp_i}]\!](m,l) \\ (\bot, (\mathsf{pkg},\mathsf{c})) & = & \tau(v_{\text{this}})(m,l) \\ (\_, params, \_, b) & = & \omega(\mathsf{pkg})(\mathsf{c})(\mathsf{op}) \\ (m',l') & = & \mathfrak{Start}(\mathsf{pkg},\mathsf{c},\mathsf{op})(v_{\text{this}}, \langle\!\langle v_1,\ldots,v_n\rangle\!\rangle)(m,l) \\ (m',l') & = & [\![\mathsf{lhs},\mathsf{exp}]\!](m,l) \\ \langle\!\langle \mathsf{lab}_0,\ldots\rangle\!\rangle & = & \mathrm{labs}(\mathsf{pkg},\mathsf{c},\mathsf{op}) \\ (\mathsf{lab}',\_) & = & \mathrm{nxt}(\mathsf{lab}) \\ (\mathsf{lab}',l,\mathsf{varN}) & = & s_0 \end{array}}{(\mathsf{lab}, S, (m,l)) \xrightarrow{\mathsf{lab:\ varN = target.op(exp_1,\ldots,exp_n)}} (\mathsf{lab}_0, s_0 :: S, (m',l'))}$$

A collection statement CollStmt first computes the collection corresponding from the call expression and the updating element from the parameter expression, then updates the collection accordingly to the native statement and proceeds to the next statement. Notice that the type checking system ensures that the element can effectively update the collection, since its type specialises the collection's type.

$$\frac{\begin{array}{rcl} v & = & [\![\mathsf{exp}]\!](d) \\ v' & = & [\![\mathsf{exp}']\!](d) \\ d' & = & d[v \mapsto v \oplus v'] \\ (\mathsf{lab}',\_) & = & \mathrm{nxt}(\mathsf{lab}) \end{array}}{(\mathsf{lab}, S, d) \xrightarrow{\mathsf{lab:\ exp.}\boldsymbol{add}\mathsf{(exp)}} (\mathsf{lab}', S, d')} \qquad \frac{\begin{array}{rcl} v & = & [\![\mathsf{exp}]\!](d) \\ v' & = & [\![\mathsf{exp}']\!](d) \\ d' & = & d[v \mapsto v \ominus v'] \\ (\mathsf{lab}',\_) & = & \mathrm{nxt}(\mathsf{lab}) \end{array}}{(\mathsf{lab}, S, d) \xrightarrow{\mathsf{lab:\ exp.}\boldsymbol{del}\mathsf{(exp)}} (\mathsf{lab}', S, d')}$$

7.5 Discussions

This Section starts by discussing some syntactic aspects of Kermeta's AL in comparison with our core AL, and then compares with related works in the domain of ALs in particular and model transformations in general.

7.5.1 Syntactic aspects & Restrictions

Returning from an operation's body (`result`). As the running example presented in Chapter 5 and the way it is transformed in Fig. 7.3 suggest, Kermeta's keyword `result` is equivalent in our formalisation to `return`.

File Dependency (`require`). We did not take into account the way Kermeta deals with file dependency since it is a pure static consideration: it only affects how a complete Kermeta specification is obtained, which can be handled by adequate pre-transformations.

Derived Properties are in MOF a special kind of property whose value is computed from other properties. As a matter of fact, it is not *per se* a concrete element as it depends on others. The value of derived properties usually depends on other properties' value: Kermeta allows to attach a `getter` for retrieving the value, and eventually a `setter` to modify it, so that their value is automatically updated if other values they depend on are. We did not address this feature, since it is possible to fully mimic this capacity by using normal operations (which is the way they are implemented in).

Value Casting. Kermeta possesses two constructions for value casting. The *conditional cast statement* `?=` is part of our AL (cf. Sec. 7.2.2 and 7.4). Our semantics distinguishes the casting of values (that are statically decidable) from the casting of instances (that can sometimes fail). In this latter case, Kermeta assigns the special value `void` (or equivalently in our AL, **null**). However, this scenario is not taken into account because the core AL does not handle exceptions and the semantic rules only rewrite correct configurations. Notice also that because primitive types are meta-represented in Kermeta (cf. Discussion Sec. 6.6), only the second rule on instances matters; the first was given for soundness and completeness purpose.

Kermeta also proposes the `asType()` operation (cf. (Drey et al. 2009, §2.9.2.2)) that avoids using an intermediate variable. The only difference resides in the error cases handling: when impossible to cast, `asType` raises an exception instead of silently failing by assigning `void`.

Collection API. Without a proper representation of generic classes and associated functionals, we equipped our core AL with only the basic capabilities for manipulating collections: testing collection emptiness or computing size, adding/removing an element, and accessing an element at a given index within an ordered collection. It is still possible to extend collections capabilities with new ones by following the same process: defining a dedicated operation on Abstract Datatypes (as defined in Section 5.4.2) that is later used within semantic rules. As a concrete example, we already defined downward/upward conversion operators in Definition 5.4: these operators can be directly used to handle the semantics of specialised cast operations in Kermeta's Collection API for converting a collection to another (these operations are named `asXxx()`, where `Xxx` stands for the name of a collection, e.g. `asSequence()`).

7.5.2 Related Works

Two transformation languages are directly comparable to Kermeta's AL: Epsilon (Kolovos et al. 2012) and xOCL (Clark, Sammut, and Willans 2008). They include OO features as well as queries and more complicated behavior like State Machines, but to the best of our knowledge, a formal specification in terms of Action Semantics only exists for UML (Yang, Michaelson, and Pooley 2008). Approaches like ASMkw and GB take advantage of the target workspace to perform the transformations, which require from the user knowledge about it (cf. the work of Combemale, Crégut, et al. (2009) for a comprehensive review). Our approach contrasts with these by formally specifying the framework itself, making formal any DSML whose semantics is expressed with Kermeta directly in the Kermeta workspace, instead of relying on target languages. Fleurey outlined in his pioneering work about Kermeta (Fleurey 2006) almost the same AL subset as ours. Nevertheless, his work is questionable in several outcomes: the structuring concepts used for the AL lack a formal counterpart; he uses a big-step semantics, which is not directly executable; and his AL subset lacks a formal type system.

Part III

Formal Verification of Kermeta

In this Part, we show how a possible implementation of the reference semantics can be performed in Maude. Maude appears to be an interesting verification domain: from a specification viewpoint, the algebraic specifications and the rewriting logic constitute an adequate formalism to embed the set-theoretic and structural operational semantics formalisation without too much effort; from a verification viewpoint, Maude offers several capabilities, from lightweight analysis techniques such as simulation and reachability analysis, to heavier techniques like model-checking and theorem-proving.

Two Chapters constitute this Part: Chapter 8 is an introduction to Maude, and Chapter 9 describes how the semantic bridges between the formal semantics to Maude is performed.

8 Maude In a Nutshell

We have selected Maude as a semantic target for the formal verification of Kermeta model transformations. This preliminary Chapter presents the basics of Maude by first describing the fundamental material upon which Maude is built (Equational and Rewriting Logics in Section 8.1, then presenting the specific constructions used for Object Orientation in Section 8.2, on which our own Kermeta specification is based. Section 8.3 finishes the Chapter by demonstrating how Maude can be used for specifying the semantics of a simple imperative programming language, summarising the methodology for expressing an operational semantics as a rewriting system.

8.1 Equational & Rewriting Logic

Maude is a wide spectrum programming language based on two logics: the *Membership Equational Logic* is an equational theory on top of which a *Rewriting Logic* allows to represent changes within terms. As a consequence, Maude offers an equational style for functional programming with rewriting logic computations. This combination, paired with an efficient rewriting engine and metalanguage capabilities, has the capability of being able to represent many models of computation using non-determinism (like those based on concurrency) and time, making Maude an interesting platform for creating executable environments for different logics, but also for programming and transformation languages.

This Section presents the basics of Maude, namely the logics it is based on. Since in this Thesis, Maude is used as a backend for formal verification, i.e. as a tool for model-checking model transformations in Kermeta, we follow a pragmatic presentation guided by small examples, rather than presenting the rich and sometimes complex mathematical foundations of the language (the interested reader can start from Clavel et al. 2007).

8.1.1 Functional Modules for Equational Theories

Membership Equational Logic (MEL) is a Horn logic whose atomic sentences are declared inside *functional modules* using **fmod** … **endfm**. A functional module computationally describes an *equational theory* (Σ, E). Σ specifies the *algebraic signature* of the theory: basic "types", called *sorts* are declared with the keywords **sort** or **sorts**; whereas *operators*, introduced with keyword **op**, are declared together with their signature, i.e. their arguments' and result's types. Additionally, sorts include an order relation, declared using the keyword **subsort**, which roughly corresponds to subtypes: it enables overloading operators with different signatures. By combining sorted variables and operators, it becomes possible to define terms, that may possess many different sorts due to the subsort and operator overloading mechanisms. Fortunately, under some convenient requirements, a term has a least sort.

Two types of sentences exist in MEL: *membership assertions*, introduced with the keyword **mb**, have the form $t : S$ and mean that term t has sort S; and *equalities*, introduced with the keyword **eq**, have the form $t = t'$ and state that terms `t` and `t'` have the same meaning. Both sentences admit a *conditional* version (introduced respectively with keywords **cmb** and **ceq**): such a sentence is computed only if the conditional holds;

and conditions are conjunctions (noted `/\`) of equations and memberships.

As an example, consider the module defining the natural numbers in Peano notation (extracted from Clavel et al. 2007).

```
fmod NATURAL is
   sorts NzNat Nat .
   subsort NzNat < Nat .
   op 0    : -> Nat [ctor] .
   op s_   : Nat -> NzNat [ctor] .
   op _+_ : Nat Nat -> Nat [assoc comm id:0] .
   vars M N : Nat .
   eq 0 + N = N .
   eq s N + M = s (N + M) .
endfm
```

The sorts `Nat` and `NzNat` define natural and non-null naturals. The Peano notation relies on two constructors (denoted with the attribute `ctor`): the constant `0` has no parameter (before `->`), and the successor operation `s_` has one `Nat` parameter, used where the placeholder `_` appears. The addition operation `_+_` is defined, and its behavior specified using equations specifying that the addition with a successor is the successor of the addition. Notice that `+` is defined with three axioms: `assoc`, `comm` and `id` stipulate that `+` is respectively associative, commutative, and admits `0` as identity element (i.e. `0` is idempotent).

In a functional module, computations occur by using equations as simplification rules from left to right, until a canonical form is reached. Consider for example the term `s 0 + s 0`, the equivalent of $1 + 1$ in usual notation. Applying the second equation reduces to `s (0 + s 0)`; then applying the first equation reduces to `s s 0`. This last term is canonical: it is solely composed of operators declared as constructors. Maude performs simplifications *modulo* axioms (for addition, commutativity, associativity and identity): these axioms are handled at low level to avoid non-termination and non-determinism (typically, an equation such as `eq N + M = M + N` for the addition, expressing commutativity, is non-terminating). Termination and determinism ensure the existence and uniqueness of canonical forms (Bouhoula, Jouannaud, and Meseguer 2000).

Maude allows *parameterised*, or *generic*, modules i.e. modules using other modules as parameters whose requirements are also defined by an equational theory. Consider for example building a datatype for sets with the following module.

```
 1 fth TRIV is
 2    sort Elt .
   endfth
 4
   fmod SET{X :: TRIV} is
 6    sort Set{X} .
      subsort X\$Elt < Set{X} .
 8    op empty : -> Set{X} [ctor] .
      op _,_   : Set{X} Set{X} -> Set{X} [ctor assoc comm id:empty] .
10    vars E : Set{X} .
      eq E, E = E .
12 endfm

14 view Nat from TRIV to NAT is
      sort Elt to Nat
16 endv
```

The module SET provides a sort Set{X} of sets over a given sort of elements, specified by the `Elt` sort in the `TRIV` theory. It defines the `empty` constant for empty sets and an associative, commutative and `empty`-idempotent operator `\_,\_` for set union. This way, thanks to the `subsort` declaration in Line 7, a singleton can be simply defined with the element itself. Line 9 provides the syntax for sets: elements are separated by commas. Notice without the equation of Line 11, this specification actually defines a multiset: by removing duplicate elements, we ensure that a comma-separated list of elements is indeed a set.

The paramter `X :: TRIV` provides a name `X` for the formal generic parameter; it has to be instantiated with a sort that fulfills the requirements of the theory `TRIV`[1]. Sorts and operators of the parameter can be used within

[1]The theory `TRIV` is defined in Maude's `prelude`, which is automatically loaded when Maude starts. This theory defines only a

the parameterised theory in a qualified form X$Elt. To instantiate a parameterised module, a *view* (introduced by the keyword view) maps sorts and operations in such a way that the requirement axioms are provable in the target module. In our example, the view Nat maps TRIV to NAT by mapping the sort Elt in TRIV to the sort Nat in NAT. After that, it becomes possible to refer to the set of natural numbers with the module expression SET{Nat}, and to use the corresponding instantiated sort name Set{Nat}.

8.1.2 System Modules for Rewrite Theories

Rewrite Logic (RL) is a logic whose atomic sentences are declared inside *system modules* using **mod** ... **endm**. A system module is basically a functional module allowing a new kind of sentences, namely (conditional) *rewriting rules* of the form `rl [lab] : t => t'` or, in its conditional form, `crl [lab] : t => t' if Cond`.

Computationally, a rule specifies a local concurrent transition that can take place in a system if the pattern of the rule's left-hand side `t` matches a piece of the system's state, and, if present, the rule's condition `Cond` is satisfied. When that happens, the matched fragment is transformed into the corresponding instance of the right-hand side `t'`. Note that each rule is `[lab]`elled.

Usually, a system comprises a *static* and a *dynamic* part, just like DSLs do. As a small illustration, consider a blackboard where a set of natural numbers is written, and one is allowed to erase any two of them to replace them by their quotient (adapted from Clavel et al. 2007). We will simply instantiate our previous view for sets of natural numbers `SET{Nat}` to directly reuse the predefined addition `_+_` and the quotient `_quo_` operations on natural numbers.

```
mod BOARD is
  pr Set{Nat} .
  vars N M : Nat .
  rl [replace] : N, M => (N + M) quo 2 .
endfm
```

The static part simply consists of the set definition, imported here using **pr** (which means `protected`). The desired behaviour is defined by the `replace` rule. Note that the rewriting actually occurs modulo the equations and attributes attached to the set definition (i.e. modulo associativity, commutativity, identity and idempotency of `_,_`, meaning for example that `6 3` and `3 6` represent the same set, as expected).

The **rewrite** command can then be used to execute the system, for example on the following set `6, 3, 2`, using the Maude interpreter, which applies the system rules until none is applicable.

```
Maude> rewrite 6, 3, 2 .
result NzNat: 4
```

As one would do it by hand, the result highly depends on the way pairs of numbers are non-deterministically chosen: system rules are not required to be terminating, nor confluent. In our case, subterm `3 2` (or its commutative equivalent `2 3`) has been first rewritten into term `2`, then the resulting term `6 2` or `2 6` has been rewritten into `4`.

The **search** command allows the exploration of the computation graph. In our case, it shows that there is only two final results.

```
Maude> search 6, 3, 2 =>! N:Nat .
Solution 1 (state 4)
N --> 4
Solution 2 (state 5)
N --> 3
No more solutions.
```

sort `Elt` with no constraints

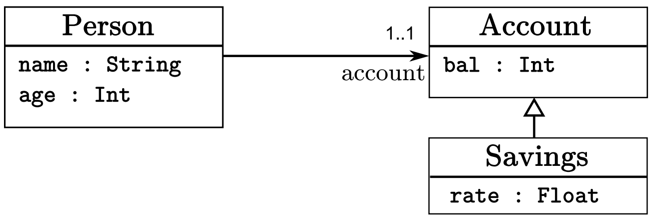


Figure 8.1 – The Bank Account Class Diagram and a simple Object Diagram.

8.2 Object-Oriented Maude

Maude allows the specification of object-oriented systems by means of object-oriented modules introduced by `omod` ... `endom`. Such modules have a dedicated syntax to declare classes containing typed "attributes". Class inheritance can be defined with the usual meaning, simply by exploiting the subsort relation. We show here on an example how class diagrams can easily be mapped into Maude's dedicated syntax. Then, we elaborate more on how object diagrams can be represented in Maude, since our formalisation of Kermeta is directly inspired from Maude's syntax for describing objects.

Consider the small class diagram depicted in Figure 8.1. A *Person*, characterised by its *name* and *age*, possesses exactly one *Account*, which has a *bal*ance. Furthermore, a special kind of account, *Savings*, is characterised by its *rate*.

```
(omod BANK-ACCOUNT is
   pr STRING .
   class Person  | name    : String,
                   age     : Nat,
                   account : Oid .
   class Account | bal     : Int .
   class Savings | rate    : Float .
   subclass Savings < Account .
endom)
```

The module `BANK-ACCOUNT` encapsulates the class diagram, and declares three classes `Person`, `Account` and `Savings` with the keyword `class`. Each class possesses a set of (comma-separated) typed attributes. In particular, `account`, which corresponds to a reference in the class diagram, is typed with the sort `Oid`: this corresponds to *object identifiers*. Subclasses are simply declared with the keyword `subclass`, using the subsort relation between class names.

Consider now the object diagram in Figure 8.1. The person named *John* is associated to an account whose balance equals 120. This would corresponds to the following Maude system:

```
   < J  : Person  | name : ''John'', age : ''42'', account: A-1 >
   < A1 : Account | bal : 120 >
```

At runtime, the state of an object-oriented system basically consists of a soup of objects, that capture the state of each object in the system. When Maude starts, it automatically imports the module `CONFIGURATION` that contains all the necessary definitions. Here is an extract:

```
mod CONFIGURATION is
   sorts Object Msg Configuration .
   subsorts Object Msg < Configuration .

   op none : -> Configuration [ctor] .
   op __    : Configuration Configuration -> Configuration [ctor assoc comm id: none] .

   sort Oid Cid .
   sorts Attribute AttributeSet .
   subsorts Attribute < AttributeSet .

   op none : -> AttributeSet [ctor] .
   op _,_   : AttributeSet AttributeSet -> AttributeSet [ctor assoc comm id: none] .
   op <_:_|_> : Oid Cid AttributeSet -> Object [ctor object] .
endm
```

As we can see on Lines 2–6, a `Configuration` is a multiset[2] of objects and messages that represents a snapshot of a possible system state. Configurations are formed by multiset union (using the empty syntax `__`). Lines 8–14 declare an object syntax corresponding to the notation we used earlier for expliciting `BANK-ACCOUNT` Object Diagram. Four sorts are introduced: `Oid` are object identifiers; `Cid` are class identifiers, `Attribute` corresponds to an element of an object's state, and `AttributeSet` just gathers attributes in a multiset structure. Objects interact in different ways, including message passing (we do not insist on messages since we will not use them).

Internally, object-oriented modules are just syntactic sugar: when computations occur over object-oriented modules, Maude simply translates them into plain system modules. For example, the previous `BANK-ACCOUNT` module is equivalently represented with the following:

```
sort Person .
subsorts Person < Cid .
op Person : -> Person .
op name_  : String -> Attribute .
op age_   : Int    -> Attribute .

sort Account .
subsorts Account < Cid .
op Account : -> Account .
op bal_   : Int    -> Attribute .

sort Savings .
subsorts Savings < Cid .
subsort Savings < Account .
op Savings : -> Savings .
op rate_  : Float  -> Attribute .
```

The reader is refered to (Clavel et al. 2007; Durán 1999) for the details on how the translation is performed.

8.3 Example: The SIMPLE Language

After the tutorial introduction to Maude, this Section demonstrates on a simple example one possible translation from mathematical definitions of languages (syntax and semantics) to Maude specifications. We have chosen SIMPLE, a toy imperative programming language often used in Maude classes[3]. The purpose is at the same time to illustrate basic translation mechanisms, but also to explain basic Maude constructions by means of a well-known language.

The rest of the Section proceeds as follows: SIMPLE syntax and semantics are first explained using the classical mathematical tools (a BNF grammar and a structural operational semantics), then the Maude equivalent is presented, the code being extensively commented to help reading.

8.3.1 Syntax

As its name indicates, the SIMPLE programming language contains the main features encountered in most imperative languages. A SIMPLE program is a sequence of statements (separated by ;) among the following: an empty statement skip, a variable assignment, a while...do...od loop, a conditional if...then...else...fi (the form without **else** can be emulated by using **skip**). Statements are build over BooleanExpressions and numeral Expressions, that both contains only a minimal set of (unary and binary operators). A Variable can be seen as a string; and a Numeral represents integers. Figure 8.2 summarises SIMPLE's BNF syntax.

In Maude, `Numerals` are non-empty strings of `Digits` (i.e. the symbols `0` ... `9`). The concatenated representation (so-called *empty syntax* _ _) imposes to use spaces between digits.

[2]Notice that the definition for `Configuration` is very similar to the definition of `SET` at the end of Section 8.1.1: except for the syntax (for sets, elements were separated by a comma whereas configurations use an empty syntax), a `Configuration` does not possess an equation to erase duplicate elements as for sets: it is therefore a multiset.

[3]During the ISR 2012 Summer School (International School on Rewriting), SIMPLE was used as a tutorial language. Thanks to Marwane EL KHARBILI, we noticed that it is actually a simplified version of Winskel's `IMP` language (Winskel 1993).

Assignable	::=	Variable
Expression	::=	Assignable | Numeral | - Expression
	|	Expression + Expression
	|	Expression - Expression
	|	Expression * Expression
BooleanExpression	::=	**true** | **false**
	|	BooleanExpression < BooleanExpression
	|	BooleanExpression == BooleanExpression
	|	**not** BooleanExpression
	|	BooleanExpression **and** BooleanExpression
BasicProgram	::=	Assignable **:=** Expression
Program	::=	**skip** | BasicProgram | Program **;** Program
	|	**if** BooleanExpression **then** Program **else** Program **fi**
	|	**while** BooleanExpression **do** Program **od**

Figure 8.2 – Syntax for **SIMPLE**. Terminals are in **bold**. A Variable is represented as a String; and a Numeral represents integers.

```
fmod NUMERAL is
   sort Digit .
   ops 0 1 2 3 4 5 6 7 8 9 : -> Digit [ctor] .
   sort Numeral .
   subsort Digit < Numeral .
   op _ _ : Numeral Digit -> Numeral [ctor prec 2] .
endfm
```

Then, it becomes possible to define `Expression`s. For variables, we simply use predefined identifiers (from module `QID`).

```
mod EXPRESSIONS is
    *** Use "Quoted Identifiers" for variables.
    ***  The module QID in the standard prelude declares a sort Qid
    ***  of LISP-like quoted identifiers.  These are just strings preceded
    ***  by a closing single-quote (’).  For example, ’a, ’b, ’ab, etc.
    ***  We rename the sort name "Qid" to "Variable", as this name is more
    ***  appropriate for defining a programming language.
    ***
   pr QID * (sort QID to Variable) .
   pr NUMERAL .
   sort Expression Assignable .
   *** Variables can be assigned integer values
   ***
   ***    <Assignable>   ::=   <Variable>
   ***
   subsort Variable < Assignable .
   *** All assignables and numerals are expressions:
   ***
   ***    Expression   ::=   Assignable | Numeral
   ***
   subsort Assignable Numeral < Expression .
   *** Binary infix addition operation.
   ***
   ***    Expression   ::=   Expression + Expression
   ***    Expression   ::=   Expression - Expression
   ***    Expression   ::=   Expression * Expression
   ***
   op _+_ : Expression Expression -> Expression [ctor prec 20] .
   op _*_ : Expression Expression -> Expression [ctor prec 16] .
   op _-_ : Expression Expression -> Expression [ctor prec 19] .
   *** Unary prefix minus operation.
   ***
   ***    Expression   ::=   - Expression
   ***
   op  -_  : Expression -> Expression [ ctor prec 3 ] .
endfm
```

It is interesting to notice, for example in Lines 16 and 21, how the BNF construction is translated into subsorting in Maude. This is not surprising, since BNF has a sort-based semantics. Notice also the use of `prec`edence for arithmetic operators to allow correct parsing (e.g., multiplication has priority over addition and substraction).

Defining `BooleanExpression`s does not differ much, since the same constructions (infix binary operators and unary operators) are used.

```
fmod BOOLEAN_EXPRESSIONS is
   pr EXPRESSIONS .
   sort BooleanExpression .
   ***  Constants:
   ***
   ***    BooleanExpression  ::=  true | false
   ***
   ops  true false  : -> BooleanExpression  [ctor] .
   *** Comparison: equality test and less-than
   ***
   ***    BooleanExpression  ::=  Expression == Expression
   ***    BooleanExpression  ::=  Expression < Expression
   ***
   op  _==_  : Expression Expression -> BooleanExpression [ctor prec 25] .
   op  _<_   : Expression Expression -> BooleanExpression [ctor prec 25] .

   *** Logical operators
   ***
   ***    BooleanExpression  ::=  BooleanExpression and BooleanExpression
   ***    BooleanExpression  ::=  not BooleanExpression
   ***
  op  _and_  : BooleanExpression BooleanExpression -> BooleanExpression [ctor prec 30] .
  op  not_   : BooleanExpression -> BooleanExpression [ctor prec 29] .
endfm
```

Defining `Program`s follows again the same principles: each BNF construction is translated according to the previous schemas. Notice here that we encounter a small disagreement: `fi` is already a Maude keyword, making it impossible to use in SIMPLE's syntax; instead, it is replaced by another usual conditional terminator `endif`.

```
fmod  PROGRAMS  is
   pr BOOLEAN_EXPRESSIONS .

   *** Basic programs are assignments.
   ***
   sort  BasicProgram .

   *** Assignment
   ***
   ***   BasicProgram  ::=  Variable := Expression
   ***
   op  _:=_  : Assignable Expression -> BasicProgram [ctor prec 50] .

   sort  Program .
   *** All basic programs are programs.
   ***
   ***   <Program>  ::=  <BasicProgram>
   ***
   subsort  BasicProgram < Program .

   *** The "do nothing" program.
   ***
   ***   Program  ::=  skip
   ***
   op  skip : -> Program  [ctor] .

   *** Sequential composition.
   ***
   ***  Program  ::=  Program ; Program
   ***
```

$$\frac{}{(\sigma, (\mathsf{stm}\ ;\ \mathsf{S})) \xrightarrow{\mathsf{stm} \triangleq \mathbf{skip}} (\sigma, \mathsf{S})}$$

$$\frac{(\sigma, p) \longrightarrow (\sigma', \_)}{(\sigma, (\mathsf{stm}\ ;\ \mathsf{S})) \xrightarrow{\mathsf{stm} \triangleq \mathsf{p}\ ;\ \mathsf{p'}} (\sigma', (\mathsf{p'}\ ;\ \mathsf{S}))} \qquad \frac{\begin{array}{rcl}[\![\mathsf{exp}]\!]_\sigma & = & n \in \mathbb{N} \\ \sigma' & = & \sigma[\mathsf{var} \mapsto n]\end{array}}{(\sigma, (\mathsf{stm}\ ;\ \mathsf{S})) \xrightarrow{\mathsf{stm} \triangleq \mathsf{var}{:=}\mathsf{exp}} (\sigma', \mathsf{S})}$$

$$\frac{[\![\mathsf{c}]\!]_\sigma = \top}{(\sigma, (\mathsf{stm}\ ;\ \mathsf{S})) \xrightarrow{\mathsf{stm} \triangleq \mathbf{if}\ \mathsf{c}\ \mathbf{then}\ \mathsf{p}\ \mathbf{else}\ \mathsf{p'}\ \mathbf{fi}} (\sigma, (\mathsf{p}\ ;\ \mathsf{S}))} \qquad \frac{[\![\mathsf{c}]\!]_\sigma = \bot}{(\sigma, (\mathsf{stm}\ ;\ \mathsf{S})) \xrightarrow{\mathsf{stm} \triangleq \mathbf{if}\ \mathsf{c}\ \mathbf{then}\ \mathsf{p}\ \mathbf{else}\ \mathsf{p'}\ \mathbf{fi}} (\sigma, (\mathsf{p'}\ ;\ \mathsf{S}))}$$

$$\frac{[\![\mathsf{c}]\!]_\sigma = \top}{(\sigma, (\mathsf{stm}\ ;\ \mathsf{S})) \xrightarrow{\mathsf{stm} \triangleq \mathbf{while}\ \mathsf{c}\ \mathbf{do}\ \mathsf{p}\ \mathbf{od}} (\sigma, (\mathsf{p}\ ;\ \mathsf{stm}\ ;\ \mathsf{S}))} \qquad \frac{[\![\mathsf{c}]\!]_\sigma = \bot}{(\sigma, (\mathsf{stm}\ ;\ \mathsf{S})) \xrightarrow{\mathsf{stm} \triangleq \mathbf{while}\ \mathsf{c}\ \mathbf{do}\ \mathsf{p}\ \mathbf{od}} (\sigma, \mathsf{S})}$$

Figure 8.3 – **SIMPLE** Semantics

```
   op  _;_  : Program Program -> Program [ctor assoc prec 60] .

   *** Conditionals & Loops
   ***  Program ::=  if BooleanExpression then Program else Program fi
   ***  Program ::=  while BooleanExpression do Program od
   ***
   *** Since "fi" is a keyword in Maude, we slightly change the syntax
   *** and use "endif" instead.
   ***
   op  if_then_else_endif : BooleanExpression Program Program -> Program [ctor prec 65] .
   op  while_do_od        : BooleanExpression Program -> Program [ctor prec 62] .
endfm
```

Thanks to the flexible syntax of Maude, translating BNF grammars into equivalent Maude constructions is easy. Like for **SIMPLE**, it is often necessary to slightly adapt the syntactic constructions in Maude due to predefined keywords, or difficulty to correctly parse grammar sentences.

8.3.2 Semantics

The semantics of **SIMPLE** follows the intuitive meaning of such constructions in an imperative language like Pascal (Wirth 1971). A program manipulates integer and boolean values, which constitutes the definition of the set $\mathbb{V}$ of values. An environment $\sigma\colon \mathsf{Variable} \longrightarrow \mathbb{V}$ stores the variables values along the program's execution. A function $[\![\bullet]\!]_\sigma\colon \mathsf{BooleanExpression} \cup \mathsf{Expression} \longrightarrow \mathbb{V}$ allows the evaluation of both types of expressions, whose precise definition is straightforward. Notice that for enabling variables' value retrieving, this function needs to depend on σ. A configuration for the operational semantics has the form (σ, S), where $\mathsf{S} \in \mathsf{Program}$.

Operational rules are described in Figure 8.3. Executing **skip** just do nothing. Executing a BasicProgram modifies the value associated to the Assignable in σ, after having evaluated the Expression. Executing a sequence simply executes the first Program, then the other. Executing a conditional or a loop just behaves the normal way, by transferring the control adequately according the the BooleanExpression evaluation.

The first task to perform when translating such a semantics into Maude is to take care of the extra material necessary for expressing the operational semantics, namely the environment and the evaluation function. Fortunately, since **SIMPLE**'s semantic domain only consists of boolean and integer values, we can simply build up over native **Bool** and **Int** native Maude sorts. Notice how the construction for store updating is conveniently translated in Maude (Lines 70 – 71) following its mathematical definition (cf. 5.4.1).

```
fmod STORE is
   *** Importing necessary modules
   ***
   pr PROGRAMS .
   pr INT .

   *** Declaring the sort for the Store
   ***
   sort Store .

   *** Initial values for stored variables.
   ***
   op  initial  : -> Store .


   *** "Look up" the value of a variable.
   ***  In fact, rather than declaring this as an operation
   ***  of type  Store Variable -> Int ,
   ***  we generalise this by saying we can evaluate any Expression,
   ***  not just a variable.
   ***
   op  _[[_]]  : Store Expression -> Int [prec 75] .


   *** "Update" a state (by assigning to a variable).
   ***
   op  _<-_ : Store BasicProgram -> Store [prec 70] .

   var S : Store .
   vars V V' : Variable .
   vars E E' : Expression .

   *** Initially, all variables store 0:
   ***
   eq  initial [[ V ]]  =  0 .


   *** evaluate binary operations, by evaluating their operands
   *** and combining the results (by addition, multiplication, etc.)
   *** evaluate unary minus by evaluating the operand, then taking the minus
   ***
   eq  S [[ E + E' ]]  =  (S [[ E ]]) + (S [[ E' ]]) .
   eq  S [[ E * E' ]]  =  (S [[ E ]]) * (S [[ E' ]]) .
   eq  S [[ E - E' ]]  =  (S [[ E ]]) - (S [[ E' ]]) .
   eq  S [[ (- E) ]]   =  - (S [[ E ]]) .

   ***  evaluate digits in the obvious way...
   ***
   eq  S [[ 0 ]]  =  0 .
   eq  S [[ 1 ]]  =  1 .
   eq  S [[ 2 ]]  =  2 .
   eq  S [[ 3 ]]  =  3 .
   eq  S [[ 4 ]]  =  4 .
   eq  S [[ 5 ]]  =  5 .
   eq  S [[ 6 ]]  =  6 .
   eq  S [[ 7 ]]  =  7 .
   eq  S [[ 8 ]]  =  8 .
   eq  S [[ 9 ]]  =  9 .

   var  N : Numeral .
   var  D : Digit .

   ***  ...and use the place system to evaluate numerals
   ***
   eq  S [[ N D ]]  =  10 * (S[[ N ]]) + (S[[ D ]]) .

   *** an assignment updates the value associated with the variable ...
   *** and only that variables (others don't change).
   ***
   eq  S <- V := E [[ V ]]  =  S [[ E ]] .
   ceq S <- V := E [[ V' ]] =  S [[ V' ]] if V =/= V' .
endfm
```

The following module SEMANTICS implements **SIMPLE**'s semantics following the rules of Figure 8.3. The evaluation of BooleanExpressions is handled simply by overloading the operator _[[_]], and mapping the effect of boolean operators to their corresponding ones within the Maude built-in **Bool** sort (notice for example, how the equality test is translated as a membership in Line 19). The configuration definition for operational rules is exactly the same as the mathematical one (except that it uses ; as a separator instead of a comma). Notice how rules with preconditions are translated into conditional equations.

```
 1 fmod SEMANTICS is
 2    protecting  PROGRAMS .
      including   STORE .
 4
      ***  To evaluate boolean expressions
 6    ***  Results are of Maude's built-in sort Bool,
      ***  from the built-in module BOOL
 8    ***
      op  _[[_]] : Store BooleanExpression -> Bool [prec 75] .
10

12    var  S : Store .
      vars E1 E2 : Expression .
14    vars T1 T2 : BooleanExpression .

16    *** evaluation of boolean expressions
      ***
18    eq  S [[ E1 < E2 ]]   =  (S [[ E1 ]]) < (S [[ E2 ]]) .
      eq  S [[ E1 == E2 ]]  =  (S [[ E1 ]]) is (S [[ E2 ]]) .
20    eq  S [[ T1 and T2 ]] =  (S [[ T1 ]]) and (S [[ T2 ]]) .
      eq  S [[ not T1 ]]    =  not (S [[ T1 ]]) .
22    eq  S [[ true ]]      =  true .
      eq  S [[ false ]]     =  false .
24

26    ***  Run program P in a store S to obtain new store S ; P .
      ***  ErrorStore is used to allow for non-terminating programs.
28    ***
      op  _;_ : Store Program -> Store [prec 70] .
30

32    var T : BooleanExpression .
      var P1 P2 : Program .
34
      ***  skip has no effect on Stores
36    ***
      eq  S ; skip  =  S .
38
      ***  Sequential composition:
40    ***  do first program, then do the next
      ***
42    eq  S ; (P1 ; P2)  =  (S ; P1) ; P2 .

44    ***  Conditionals:
      ***  if test is true, execute the then-clause...
46    ***
      ceq   S ; if T then P1 else P2 endif  =  S ; P1   if  S[[T]] .
48
      ***  ... otherwise, execute the else-clause
50    ***
      ceq   S ; if T then P1 else P2 endif  =  S ; P2   if  not(S[[T]]) .
52
      ***  While-loops:
54    ***  if guard is true, execute the body, and repeat ...
      ***
56    ceq   S ; while T do P1 od  =  S ; P1 ; while T do P1 od  if  S[[T]] .

58    ***  ... otherwise, exit the loop (do nothing)
      ***
60    ceq   S ; while T do P1 od  =  S   if  not(S[[T]]) .
   endfm
```

9 KMV: Verifying Kermeta with Maude

We showed in Chapter 3 that no tools are currently available for formally analysing or verifying metaprogrammed model transformations. A natural possibility would consist of implementing such a tool from scratch, thus building an *ad hoc* tool specialised for our formally specified Action Language subset. However, this task is difficult, error-prone, and requires time to reach the maturity necessary to handle our subset correctly so that it reasonably scales for the size of model transformations that represent the current practice.

This Chapter shows how a bridge from the reference semantics of Kermeta (presented in Part II) into Maude (Clavel et al. 2007) can be defined. Maude has been chosen as a verification domain for three reasons:

- Maude's mathematical background (Membership Equational Logics and Rewriting Logics) is very close to the mathematical tools we used for specifying the semantics of Kermeta: this reduces the semantic gap between the reference semantics and this implementation, allowing one to manually check the implementation's correctness;
- Maude's specifications are executable: this implementation offers a lightweight executable framework for the formalised subset of the Action Language, and a good basis for handling more features;
- Maude comes equipped with two verification tools, namely model-checking and theorem-proving: this enables transformation designers to address a larger variety of properties within a single tool.

After providing an overview of the implementation strategy, we organise the Chapter the same way the formal specification of Part II was built: the Structural and Action Languages are described in Sections 9.2 and 9.3, respectively. Each one is illustrated with the FSM example, to show how the specifications are used at the model level, and how to execute a specification with Maude. Section 9.5 describes KMV, as a tool integrated into Eclipse, as well as its current limitations.

9.1 Overview

Instead of providing a novel Maude specification for metamodels and models, which takes a long time and is subject to errors, we decided to base our tool on existing specifications that were already tested and validated. We first explain how we selected an adequate specification, then the assumptions for our tool to work regarding the Kermeta specification considered as input.

9.1.1 mOdCL & Maudeling

Among all existing specifications for handling metamodelling in Maude (Boronat and Meseguer 2008; Mokhati and Badri 2009; Rivera, Durán, and Vallecillo 2009; Rusu 2011), KMV is built on top of two existing Maude specifications: mOdCL and Maudeling[1]. This solution was preferred for the following reasons:

[1]Both tools are developed by the Atenea Team, hosted at the University of Málaga: http://atenea.lcc.uma.es/.

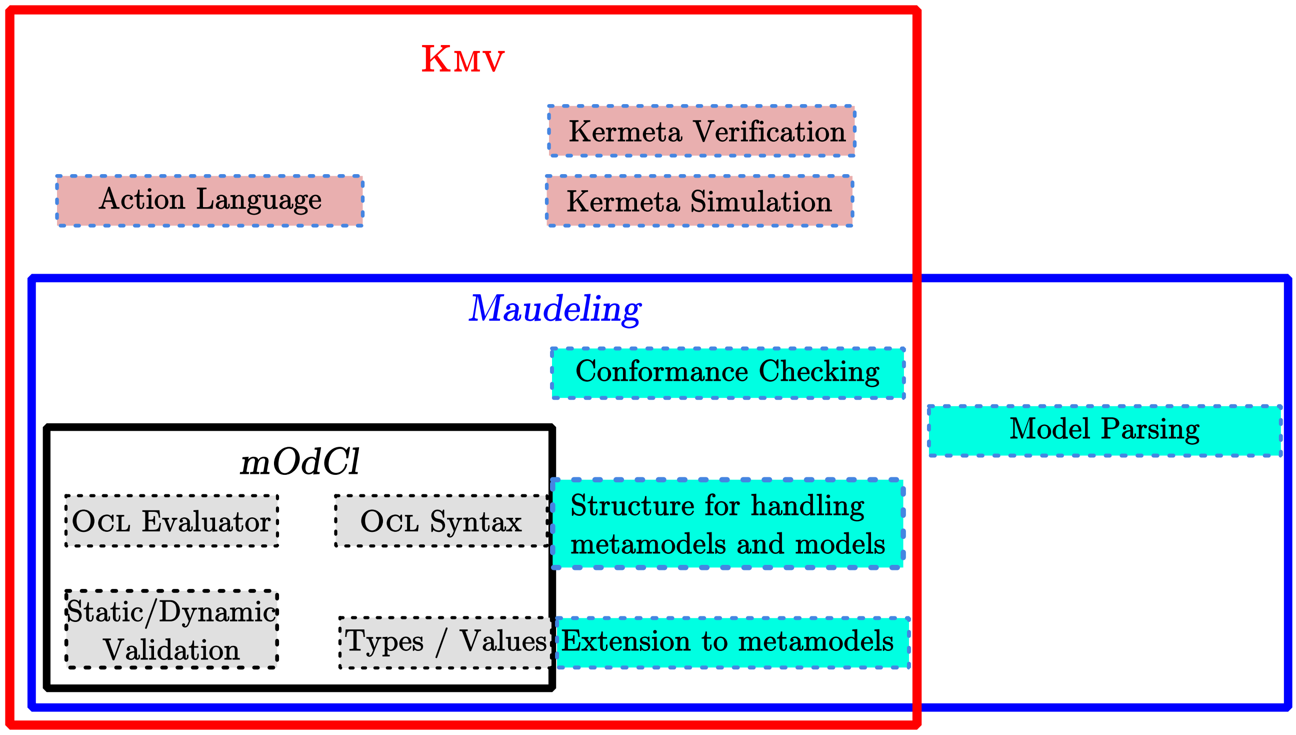


Figure 9.1 – Dependencies between mOdCL, Maudeling and KMV: each colored box represents one tool with its components. mOdCL is included into Maudeling, and both tools are reused by KMV.

Encompassing current limitations We showed in Section 8.2 that the built-in object-oriented Maude models are not sufficient for our purpose, i.e. representing metamodels, packages and MDE-specific constructions (like multiplicities, MOF properties and their typing information). On the contrary, mOdCL and Maudeling propose dedicated constructions that simplify the representation of Kermeta's constructions.

Avoiding Full Maude Some implementations (among which, the ones proposed by Boronat and Meseguer (2008) and Mokhati and Badri (2009)) as well as the built-in object-oriented modules make calls to Full Maude, thus requiring a performance overhead that becomes cumbersome for execution purposes which perform recurrent basic operations on metamodels. In contrast, mOdCL and Maudeling work at the level of Core Maude, enhancing the overall performance of our tool dramatically.

Figure 9.1 depicts the interactions between mOdCL, Maudeling and Kermeta. mOdCL complies with the OCL syntax (Object Management Group 2010), and provides an expression evaluator that successfully passes existing OCL benchmarks (e.g. the one proposed by Kuhlmann et al. 2013). The OCL evaluator allows on the one hand to validate the static syntax of UML models, i.e. what is called in metamodelling, *model constraints* (Durán, Gogolla, and Roldán 2011), but also to dynamically validate such constraints on an executable UML specification whose behaviour is defined by means of Activity and Sequence Diagrams (Object Management Group 2011a), with multithread support (Durán and Roldán 2011).

However, the language for representing OCL expressions is limited regarding metamodelling activities as perceived in MOF: several constructions in MOF need to be manipulated as first-class concepts, instead of being directly represented as built-in components. For example, the data types and collection types need to appear syntactically in a MOF specification to enable conformance checking, as well as the notion of metamodel. For these reasons, Maudeling extends the mOdCL type and value system to support these features, without breaking the underlying evaluation capabilities. Maudeling also defines the structure of metamodels in a way that is fully compatible with MOF, making the previous notions explicit. Besides, Maudeling offers a conformance checking predicate that can serve as a basis for checking conformance for Kermeta specifications, and a parser to create Maude representation of MOF models from XMI descriptions, the native representation of the Eclipse implementation of MOF, ECore (Steinberg et al. 2009).

KMV is built on top of the mOdCL and the Maudeling specification in Maude. For specifying Kermeta's Structural Language, KMV extends the specification already available in Maudeling:

- Kermeta requires the *from* clause to disambiguate properties and operations names that are multiply inherited, and enabling a full static type checking of the Action Language (cf. discussion in Section 5.2.1);
- Obviously, Kermeta requires the definition of operations (cf. Section 6.3.5) present in MOF but not specified in Maudeling because transformations were defined using Graph-Based Transformations techniques;
- This new elements imposes that the conformance checking is modified accordingly, to check for example that operations defined as abstract do not possess a body and are contained in an abstract class.

The previous elements are purely syntactic and describes the *structure* of model specifications in Kermeta. KMV's originality resides in the *dynamic* part, i.e. the control flow and execution rules necessary to perform transformations' execution, as described in Section 9.3.

9.1.2 Assumptions

Both Kermeta and Maude, through the Maude Development Tool[2], are integrated within Eclipse. As a natural continuation, KMV is itself developed within Eclipse as a plugin that can be integrated in the Kermeta Perspective.

Figure 9.2 depicts the interactions between KMV and Eclipse, highlighting the assumptions made by KMV regarding Kermeta input models that it is capable of analysing:

Preprocessing On the one hand, Maude relies on algebraic specifications, meaning that each term is unique up to an entire Maude specification; on the other hand, Kermeta makes use of names whose scope depends on their location (e.g. the scope of a class name is the entire metamodel; the scope of a property name is its containing class, which prevents operation parameters to be named identically). KMV has to maintain a bi-directional correspondence between Kermeta names and Maude terms, that is computed during a preprocessing phase prior to any analysis.

Compiled Validated Models KMV assumes that analysed models successfully pass the *compilation* and *validation* phases without raising errors: indeed, models are expected to be *conforming* and *type correct*. KMV currently works directly on the Abstract Syntax Tree used as an internal representation for Kermeta specifications in order to avoid computing information (like the qualified names of entities that are used with the `require`/`uses` clauses) that the compiler already does. By doing so, KMV is able to handle aspects, since the Abstract Syntax Tree represents the full specification where the aspects are already resolved.

The *preprocessing* assumption results both from historical reasons (to enforce the reuse of mOdCL and Maudeling) and from the inevitable abstractions necessary when a mathematical reference semantics has to be implemented in a real-life programming language: by reusing the target language's built-in representation mechanisms (namely, algebraic terms), one gives priority to the tool scalability and performance over the strict compliance with the mathematical definitions. This point is further discussed in Section 9.5.2.

The *compiled validated models* assumption seems to be reasonable to ensure the rapid prototyping of KMV, and can be improved in the future. However, it acts as an abstraction layer that prevents KMV from having to evolve accordingly to the Kermeta platform: this decouples KMV from the possible future evolutions of the

[2]Maude Development Tool (MDT) Website: http://mdt.sourceforge.net/

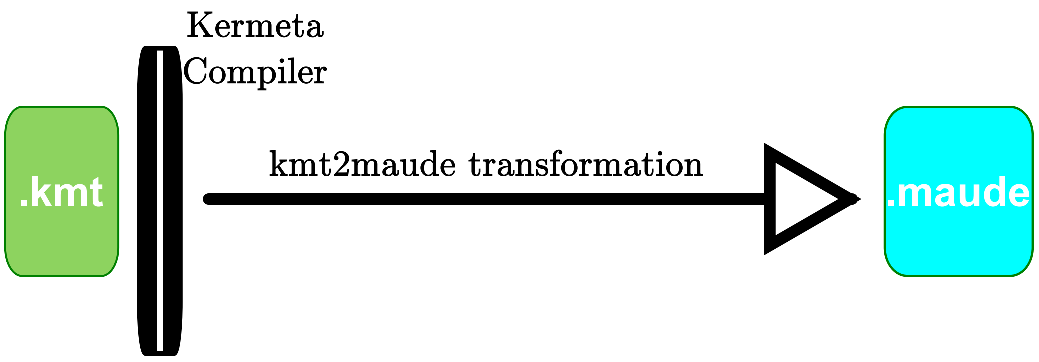


Figure 9.2 – The `kmt2maude` transformation assumes that a Kermeta specification successfully compiles.

Kermeta platform, forcing only the *Translation* transformation from a model of the Abstract Syntax Tree into the entry format of KMV to be maintained.

We now proceed to the description of the implementation of the formal semantics in Maude. It is not possible to formally ensure the correctness of this kind of implementation for two reasons: the reference semantics, although mathematical (and thus, having a "formal" semantics, as the meaning commonly understood for mathematics themselves), is not supported by a tool, which is required for a mechanical proof of correction; any proof would be complicated anyway by the fact that implementation choices, abstractions and "tricks" that a good programmer will adopt to enhance the readability or the performance of the implementation are always difficult to formalise. However, we will follow *literate programming* principles (Knuth 1992) to facilitate a manual checking of the correctness of the implementation.

9.2 Specifying Kermeta's Structural Language

The structural part of KMV is an extension of Maudeling, which is itself built on top of mOdCL (cf. Section 9.1.1 and Figure 9.1): KMV adds to Maudeling Kermeta's specificities for metamodelling, namely operation declarations and the *from* clause.

The reference semantics of Part II relies on a set-theoretical framework that makes an extensive use of partial functions: Definitions 6.4 and 6.5 have defined metamodels and models as a tuple of partial functions that captured the meaning of their constituting parts. Unfortunately, Maude does not support such a mechanism for *creating* tuples. For example for a package function $p\colon \mathsf{PkgN} \rightarrow\!\!\!\!\!+ \wp(\mathsf{PkgN}) \times \wp(\mathsf{ClassN}) \times \wp(\mathsf{EnumN}) \in \mathcal{P}$, that associates to a package its subpackages, classes and enumerations, it is not possible to define an algebraic operation that straightforwardly mimic this set-theoretic construction, something like `op p : @Package -> Set{@Package} Set{@Class} Set{Enumeration}`.

A simple solution would consist of creating an extra sort, say `@PackageResult`, that represents p's image, and then define appropriate accessors accordingly. This way, $p \in \mathcal{P}$ would then have an equivalent Maude constructor `op p` defined as follows:

```
sort @PackageResult .
op packageResult   : Set{@Package} Set{@Class} Set{Enumeration} -> @PackageResult .
op p : @Package -> @@PackageResult .
```

This solution is theoretically perfectly acceptable, but leads to numerous problems: if a codomain (or some parts of it), like `packageResult`, is shared by several partial functions, it becomes quickly ambiguous to parse; furthermore, semantic rewritings, as defined in Chapter 7, often occur in limited places in the codomain, forcing to instanciate many variables to represent a valid term of the codomain, which in return forces to match many information that is ultimately never really used.

Instead, we adopted another solution that is better regarding both the performance and the readability (and also appears to be more convenient to manipulate in Maude): we defined "*projector*" operators, just like the auxiliary functions that we defined over qualified names (cf. Section 6.3). As an example, to retrieve the set of subpackages for the package `pkg`, one will invoke an operator as follows: `subpackages(pkg)`.

Before jumping to the systematic description of the components of a metamodel, we detail how the common components for names, types and values are built.

9.2.1 Names, Types, and Values

We provided a formal definition for names (Definition 6.1), types (Definition 6.2) and values (Definition 6.3). As we recalled earlier, we need a meta-representation of types, and then a explicit link between values and their syntactic type (cf. Definition 6.6). This will give the possibility to perform (ontological) conformance based on this meta-representation.

Names were defined in Definition 6.1 as a sorted set $\mathsf{Name} \triangleq (\mathsf{Name}_e)$ where $\mathsf{e} \in \mathsf{Element}$ ranges over the set of named metamodel elements. To represent them in Maude, we define a sort for each indexed set (i.e. $\mathsf{Name}_{\mathsf{PkgN}}$ becomes `@Package`, $\mathsf{Name}_{\mathsf{ClassN}}$ becomes `@Class`, and so on). The **subsort** declarations follows the MOF inheritance relations between the classes: for example, Line 13 expresses the fact that all metamodel elements inherits from *NamedElement*; and Line 14 establishes that *Class*es and *DataType*s are different sorts of *Classifier*s. Note that the sort `@Operation` is declared as a `@StructuralFeature`, in order to handle its return type.

```
1    sort @NamedElement .
2    sort @Metamodel .
     sort @Package .
4    sort @Classifier .
     sort @StructuralFeature .
6    sort @Attribute .
     sort @Reference .
8    sort @Operation .
     sort @Parameter .
10   sort @DataType .
     sort @Enumeration .
12
     subsorts @Metamodel @Package @Classifier @StructuralFeature < @NamedElement .
14   subsorts @Class @DataType < @Classifier .
     subsorts @Attribute @Reference @Parameter @Operation < @StructuralFeature .
16   subsort  @Enumeration < @DataType .

18   op name : @NamedElement -> String .
```

A string, representing the name as it is specified in Kermeta, is associated to any term typed as `@NamedElement` (cf. example below). Notice that most of the sort names are prefixed by `@` to avoid conflict with already defined sorts in Maude.

Example 9.1 (Declaring named elements)**.** Let us see how to declare the *FSM* package, the *Label* class and its *label* attribute, as well as how the correspondence with Kermeta's names is ensured.

```
op FSM : -> @Package .
eq name(FSM) = "FSM" .

sort Label@FSM .
subsort Label@FSM < @Class .
op Label@FSM : -> Label@FSM .
eq name(Label@FSM) = "Label" .

op label@Label@FSM  : -> @Attribute .
eq name (label@Label@FSM) = "label" .
```

A first important point concerns the naming conventions. To obtain an unique term name, we use a form of qualified names by concatenating nested elements names, separated with the special character `@`: this way, the term representing the *label* attribute becomes `label@Label@FSM`, where we recognise the name of the attribute, concatenated with the name of its containing class, with the name of its containing package. Given the name constraints in Kermeta, this schema ensures term names unicity while still easily reversible.

We showed how things are declared for an attribute, a class and a package, but other metamodel elements follow the same pattern: a model element is declared as a constant operator (Lines 1, 6 and 9) for building terms of the appropriate sort. Then, the term is linked to the original Kermeta name using an equation (Lines 2, 7 and 10). Classes are treated slightly differently in order to benefit from the built-in subsorting mechanism: each class sort is declared to be a subsort of `@Class`. □

Specifying types and values is straightforward. First, recall from Definition 6.2 that syntactic types are built on top of DataTypes, defined as the union of PrimType and EnumN (i.e. a primitive type is also a datatype). This is simply translated using constants with well-chosen names (enumerations will be defined later).

```
op @String   : -> @DataType .
op @Int      : -> @DataType .
op @Bool     : -> @DataType .
op @Float    : -> @DataType .
```

Then, we need to build collection and multiple types, by defining collections and multiplicities (cf. Definition 6.2). To facilitate their manipulation, and ensure better compatibility with UML-based models (as well as ECore models), things are separated to allow a finer manipulation, with the help of projector functions. Remember that upperbounds can represent a finite, undefined value $\star$ encoded as the value `-1`, and this supposes to redefine the associated order `<=Card` on naturals.

```
op lowerBound : @StructuralFeature -> Int .
op upperBound : @StructuralFeature -> Int .
op isOrdered  : @StructuralFeature -> Bool .
op isUnique   : @StructuralFeature -> Bool .

op * : -> Int .
eq * = -1 .

op maxCard : Int Int -> Int [comm] .
eq maxCard(-1, I1) = -1 .
eq maxCard(I1, I2) = max(I1, I2) [owise] .
op minCard : Int Int -> Int [comm] .
eq minCard(-1, I1) = I1 .
eq minCard(I1, I2) = min(I1, I2) [owise] .

op _<=Card_ : Int Int -> Bool .
eq I1 <=Card -1 = true .
eq -1 <=Card I2 = (I2 == -1) .
eq I1 <=Card I2 = I1 <= I2 [owise] .

op isMany : @StructuralFeature -> Bool .
eq isMany(SF) =   (2 <=Card upperBound(SF)) .
op isRequired : @StructuralFeature -> Bool .
eq isRequired(SF) = (1 <=Card lowerBound(SF)) .
```

Two extra operators `isMany` and `isRequired` will be useful for the conformance: `isMany` indicates that the multiplicity requires a collection (because its upperbound is greater than 2); whereas `isRequired` indicates that at least one element is necessary.

Example 9.2 (Multiplicity Types)**.** In the FSM, the *label* attribute in class *Label* has multiplicity 1:1, and is represented as a Set collection (since it should be unique).

```
    eq lowerBound (label@Label@FSM) = 1 .
    eq upperBound (label@Label@FSM) = 1 .
    eq isOrdered (label@Label@FSM) = false .
    eq isUnique (label@Label@FSM) = true .
```

□

9.2.2 Metamodel

We follow a systematic presentation for each metamodel element: after recalling the corresponding definition from Section 6.3, we show how the projector operators are defined with respect to the formal definition, then provide sample examples extracted from the FSM example. Appendix A.4 provides the full Maude specification for our running example.

A generic list datatype (with empty syntax) is used for representing the codomain elements of of partial functions using powersets (those constructed with $\wp\cdot$). This definition will enable a transparent treatment of all components in the metamodel, e.g. when defining the conformance. This generic datatype is then instantiated with sort `@NamedElement` (which all others are subsort of) and renamed `MyList`[3].

```
fmod MGLIST{X :: TRIV} is
    sort EmptyList MGNeList{X} MGList{X} .
    subsort EmptyList MGNeList{X} < MGList{X} .
    subsort X$Elt < MGNeList{X} .

    op nil : -> EmptyList [ctor] .
    op __ : X$Elt MGList{X} -> MGList{X} [ctor prec 25] .

    op append : EmptyList EmptyList -> EmptyList .
    op append : MGList{X} MGList{X} -> MGList{X} .

    op occurs : X$Elt MGList{X} -> Bool .

    op reverse : EmptyList -> EmptyList .
    op reverse : MGList{X} -> MGList{X} .
endfm $
```

9.2.2.1 Package

This is the first time projector functions are used: we will detail here how the translation between the mathematical definition and the Maude specification is performed. Since it is quite natural, we will only sketch these details for future definitions, and count on the intuition based on the projector names to properly map both representations.

Section 6.3.1 defined what a package function $p \in \mathcal{P}$ looks like: it maps an existing package (name) pkg to the set of subpackages P, classes C and enumerations E, i.e. $p(\mathsf{pkg}) = (P, C, E)$. For each component in the codomain, a projector operator (named `subPackages`, `classes` and `enumerations`) is introduced:

```
   op subPackages  : @Package -> MyList . --- Of @Package
   op classes      : @Package -> MyList . --- Of @Class
   op enumerations : @Package -> MyList . --- Of @Enumeration
```

These operators map terms sorted over `@Package` to the list structure `MyList`. Let $\mathsf{pkg} \in \mathsf{PkgN}$ such that $\mathsf{pkg} \in \mathsf{Dom}\,(p)$, and `pkg : @Package`. The equivalence between the mathematical notation and the Maude definitions is

$$p(\mathtt{pkg}) = (\mathtt{P},\mathtt{C},\mathtt{E}) \overset{\triangle}{\iff} \begin{cases} \mathtt{subpackage(pkg)=P} \\ \mathtt{classes(pkg)=C} \\ \mathtt{enumerations(pkg)=E} \end{cases}$$

Additionally, we define three *helper operators*, which are not strictly required for defining a metamodel, but help computing information about the package structure: `allSubPackages` retrieves all subpackages recursively (i.e. finds the subpackages within subpackages of a given package); `allClasses` retrieves all classes declared within a package similarly; and `superPackage` is the inverse of the `subPackage` operator.

[3]This implementation choice is also the consequence of the language's learning curve: `MyList` is defined as a view from a list-like module, whose effective parameter is `@NamedElement`, which makes it usable for any metamodel element. This also marks the very difference between *mathematical specification*, whose goal is conciseness and precision, and *implementation*, whose goal is convenience and efficiency. Lists of `@NamedElements` are seamlessly treated at the expense of not catching mistyped lists. However, since we consider type-correct Kermeta specification, this has a minor impact.

```
op allSubPackages : MyList -> MyList . --- Of @Package (both) .
eq allSubPackages(nil) = nil .
eq allSubPackages(P) = (subPackages(P) allSubPackages(subPackages(P))) .
eq allSubPackages((P L)) = (allSubPackages(P) allSubPackages(L)) .

op allClasses : @Package -> MyList . --- Of @Class
eq allClasses(P) = (classes(P) classesAux(allSubPackages(P))) .

op classesAux : MyList -> MyList . --- Of @Package, @Class
eq classesAux(nil) = nil .
eq classesAux(P L) = (classes(P) classesAux(L)) .

op superPackage : @Package -> Maybe{@Package} .
```

Here, `allSubPackages` and `allClasses` are computed by structural induction, taking advantage of the empty syntax for list (i.e. the list constituted by `E1` and `E2` is simply noted `(E1 E2)`).

Example 9.3 (The FSM Package)**.** It is obviously named FSM, and has no sub- or super-package.

```
eq name(FSM) = "FSM" .
eq superPackage(FSM) = null .
eq subPackages(FSM) = nil .
eq classes(FSM) = __(FSM@FSM, Label@FSM, State@FSM, Transition@FSM) .
```

□

9.2.2.2 Class

Section 6.3.3 defined the structure of a class function $c \in \mathcal{C}$. It maps an existing class (name) c within a package (name) pkg to a boolean (indicating if c is abstract) and the set of c's superclasses: $c(\mathsf{pkg})(\mathsf{c}) = (\mathsf{abs}, \{(\mathsf{pkg}, \mathsf{c}_1), (\mathsf{pkg}, \mathsf{c}_n)\})$. The corresponding Maude specification simply follows the same scheme as for packages, by defining the corresponding projectors `superTypes` and `isAbstract`.

```
op superTypes : @Class -> MyList . --- Of @Class
op isAbstract : @Class -> Bool .
op package : @Classifier -> Maybe{@Package} .
op references : @Class -> MyList . --- Of @Reference
op attributes : @Class -> MyList . --- Of @Attribute
```

Additionally, we need to handle the fact that a class is eventually contained within a package (operator `package`, with as result a term of sort `Maybe{@Package}` indicating the optional enclosing package), and that references and attributes are also enclosed within a class (operators `references` and `attributes` respectively). Similarly to the previous Section, we can establish the following equivalences between both notations:

$$\mathsf{Dom}\left(\mathrm{prop}(\mathsf{pkg})(\mathsf{c})\right) = \mathsf{LREFS} \cup \mathsf{LATTS} \overset{\triangle}{\Longleftrightarrow} \begin{cases} \mathsf{references}(C) = \mathsf{LREFS} \\ \mathsf{attributes}(C) = \mathsf{LATTS} \end{cases}$$

$$c(\mathsf{pkg})(\mathsf{c}) = (abs, C) \overset{\triangle}{\Longleftrightarrow} \begin{cases} \mathtt{isAbstract}(\mathtt{c}) = abs \\ \mathtt{package}(\mathtt{c}) = \mathtt{pkg} \\ \mathtt{superType}(\mathtt{c}) = \mathtt{C} \end{cases}$$

The *order relation* $\prec_{\mathsf{Class}} \subseteq \mathsf{PClassN} \times \mathsf{PClassN}$, induced by the inheritance hierarchy, needs to be captured by an operator `subTypeOf`: it follows a definition by structural induction over `@Classifiers`, and needs to computes the superclass transitive closure, which is captured by the operator `allSuperTypes`.

```
op subTypeOf : @DataType @DataType -> Bool .
eq subTypeOf(DT, @DataType) = true .
eq subTypeOf(DT, DT) = true .
eq subTypeOf(DT, DT2) = false [owise] .

op subTypeOf : @Class @Class -> Bool .
eq subTypeOf(C, C2) = superTypeOf(C2, C) .

op allSuperTypes : MyList -> MyList . --- Of @Class (both)
eq allSuperTypes(nil) = nil .
eq allSuperTypes(C) = (superTypes(C) allSuperTypes(superTypes(C))) .
eq allSuperTypes(C L) = (allSuperTypes(C) allSuperTypes(L)) .
```

Example 9.4 (The Label and *FSM* Classes)**.** We illustrate the declaration of classes on *Label* and *FSM*. Note the translation for *FSM*, which inherits from *Label*.

```
eq name(Label@FSM) = "Label" .
eq isAbstract(Label@FSM) = true .
eq package(Label@FSM) = FSM .
eq superTypes (Label@FSM) = nil .
eq references (Label@FSM) = nil .
eq attributes (Label@FSM) = label@Label@FSM .

eq name(FSM@FSM) = "FSM" .
eq isAbstract(FSM@FSM) = false .
eq package(FSM@FSM) = FSM .
eq superTypes (FSM@FSM) = Label@FSM .
eq references (FSM@FSM) = __(states@FSM@FSM, transitions@FSM@FSM) .
eq attributes (FSM@FSM) = alphabet@FSM@FSM .
```

□

9.2.2.3 Enumeration

Section 6.3.2 defined the structure of an enumeration function $e \in \mathcal{E}$. It maps an enumeration (name) enum within a package (name) pkg to an order list of enumeration literals: $e(\mathsf{pkg})(\mathsf{enum}) = \langle\!\langle \mathsf{elit}_1, \ldots, \mathsf{elit}_n \rangle\!\rangle$. We just introduce an operator for that purpose.

```
subsort @EnumerationInstance < @DataTypeInstance .
op literals : @Enumeration -> List{@EnumerationInstance} .
```

Here, the sort `@EnumerationInstance` represents our enumerations literals, i.e. elements belonging to $\mathbb{E}$, declared to be a subsort of `@DataTypeInstance`, the sort representing elements of PrimType.

Example 9.5 (The Kind Enumeration)**.** The KIND enumeration within the FSM just declares three values for the possible kinds of states. Therefore, after creating a sort for representing the enumeration itself, three operators create exactly one constant for each enumeration value. The usual information for metamodel components is also required: the enumeration `name`, the containment information (`metaAux` and `package` and `literals`). The `defaultValue` is set to the first enumeration found on the list.

```
sort Kind@FSM .
subsorts Kind@FSM < @EnumerationInstance .
op Kind@FSM : -> @Enumeration .
op NORMAL@Kind@FSM : -> Kind@FSM .
op START@Kind@FSM : -> Kind@FSM .
op STOP@Kind@FSM : -> Kind@FSM .
eq metaAux( X:Kind@FSM ) = Kind@FSM .
eq name( Kind@FSM ) = "Kind" .
eq package( Kind@FSM ) = FSM .
eq defaultValue( Kind@FSM ) = NORMAL@Kind@FSM .
eq literals( Kind@FSM ) = __( NORMAL@Kind@FSM , START@Kind@FSM , STOP@Kind@FSM ) .
```

□

9.2.2.4 Property

Section 6.3.4 defined the structure of a property function prop $\in \mathcal{P}rop$. It maps a property (name) prop to a boolean indicating if it is contained, a disambiguation clause, a multiple type, and an eventual opposite: $\mathsf{prop}(\mathsf{pkg})(\mathsf{c})(\mathsf{prop}) = (cnt, from, \mathsf{mt}, \mathsf{opp})$. It also satisfies well-formedness rules that constrain the multiple type according to the property's nature, and the opposite to adequately be represented in both sides.

```
op type : @StructuralFeature -> @Classifier .
op type : @Attribute -> @DataType [ditto] .
op type : @Reference -> @Class [ditto] .
op from : @StructuralFeature -> Maybe{@Class} .
op opposite : @Reference -> Maybe{@Reference} .
op isContainment : @Reference -> Bool .

op isContainer : @Reference -> Bool .
eq isContainer(REF)= isContainment(opposite(REF)) .
```

We delay the opposite checking to the conformance checking: it is easier to specify it directly with a model at hand. However, the constraint on features' types is in Maude elegantly expressed using operators overloading: an `@Attribute`'s (resp. `@Reference`)`type` is always a `@DataType` (resp. `@Class`). Notice the use of `Maybe{@Class}`: it corresponds to the notation $\mathsf{Class}_\perp$ extending a set with one extra element $\perp$. The module on the right proceeds the same way by introducing a (generic) sort `Maybe{X}` containing any term of sort `X`.

Example 9.6 (References in the Transition Class)**.** We illustrate the declarations of properties with the *Transition* class, that only contains references (it equally applies to attributes).

```
op src@Transition@FSM : -> @Reference .
eq name (src@Transition@FSM) = "src" .
eq opposite (src@Transition@FSM) = out@State@FSM .
eq type (src@Transition@FSM) = State@FSM .
eq lowerBound (src@Transition@FSM) = 1 .
eq upperBound (src@Transition@FSM) = 1 .
eq containingClass (src@Transition@FSM) = Transition@FSM .
eq isOrdered (src@Transition@FSM) = true .
eq isUnique (src@Transition@FSM) = true .
eq isContainment (src@Transition@FSM) = false .

op tgt@Transition@FSM : -> @Reference .
eq name (tgt@Transition@FSM) = "tgt" .
eq opposite (tgt@Transition@FSM) = in@State@FSM .
eq type (tgt@Transition@FSM) = State@FSM .
eq lowerBound (tgt@Transition@FSM) = 1 .
eq upperBound (tgt@Transition@FSM) = 1 .
eq containingClass (tgt@Transition@FSM) = Transition@FSM .
eq isOrdered (tgt@Transition@FSM) = true .
eq isUnique (tgt@Transition@FSM) = true .
eq isContainment (tgt@Transition@FSM) = false .
```

Notice how it becomes necessary, due to the subsorting mechanism, to define the value of all operators relevant for `@Reference`: because `@Reference` is a subsort of `@StructuralFeature`, the multiplicity and collection must receive an equational definition. □

9.2.2.5 Operation

Section 6.3.5 defined the structure of an operation function $o \in \mathbb{O}$. It maps an operation (name) op within a class (name) c to a boolean indicating if it is abstract, a disambiguation clause, the (ordered) list of its parameters if any, a multiple type corresponding to the result's type, and its body: $o(\mathsf{pkg})(\mathsf{c})(\mathsf{op}) = (abs, \mathsf{from}, paramlist, \mathsf{mt}, body)$.

```
sort @Operation @Parameter .
subsort @Operation @Parameter < @StructuralFeature .
op isAbstract : @Operation -> Bool .
op from : @StructuralFeature -> Maybe{@Class} .
op containingOperation : @Parameter -> @Operation .
op parameters : @Operation -> MyList . --- of @Parameter
```

We declared sorts `@Operation` and `@Parameter` as subsorts of `@StructuralFeature` for defining their multiplicity type (which corresponds to the result type for `@Operation`). We define a pair of operators `containingOperation` and `parameters` as usual for contained metamodel elements. Since a *from* clause can be optional, the operator `from` builts a term of sort `Maybe{@Class}` that corresponds to our mathematical notation $\mathsf{Class}_\perp$. Operation bodies will be defined in Section 9.3, when dealing with AL's specification.

9.2.2.6 Metamodel

From Definition 6.4 $\mathsf{MM} \in \mathcal{M}$, a metamodel is defined by gathering all previous metamodel functions: $\mathsf{MM} = (p, c, e, \mathsf{prop}, o)$. We introduce a sort `@Metamodel` to represent metamodels explicitly: it is required by the object-oriented style in Maude (cf. Section 8.2).

```
sort    @Metamodel .
subsort @Metamodel < @NamedElement .
op packages : @Metamodel -> MyList . --- Of @Package
```

Example 9.7 (The FSM Metamodel)**.** We can finally define the entire *FSM* Maude representation (the full definition is available in Appendix A.4.

```
mod FSM-MM is
   op FiniteStateMachine : -> @Metamodel .
   eq name(FiniteStateMachine) = "FSM" .
   eq packages (FiniteStateMachine) = FSM .
   *** Other package, class, enumerations,
   *** properties and operations definitions
endfm
```

□

Several helper operations are defined to retrieve information by traversing a metamodel's structure back and forth. The operator `metamodel` attaches a package, a classifier or a feature to its container metamodel; note that for `@Classifier` and `@StructuralFeature`, it is computed from the metamodel's declarations. Operators `allPackages`, `allClasses`, `allAttributes`, `allReferences` and `allCcontainingReferences` retrieve the corresponding information from a metamodel, by also recursively examining substructures (e.g., subpackages for `allPackages`, or superclasses for `allReferences`).

```
op metamodel : @Package -> @Metamodel .

var MM  : @Metamodel .
var CLF : @Classifier .
var SF  : @StructuralFeature .

op metamodel : @Classifier -> Maybe{@Metamodel} .
eq metamodel(CLF) = if (package(CLF) == null) then null else metamodel(package(CLF)) fi .
eq metamodel(CLF) = metamodel(package(CLF)) .

op metamodel : @StructuralFeature -> @Metamodel .
eq metamodel(SF) = metamodel(package(containingClass(SF))) .

op allPackages : @Metamodel -> MyList . --- Of @Package
eq allPackages(MM) = (packages(MM) allSubPackages(packages(MM))) .

op allClasses : @Metamodel -> MyList . --- Of @Class
eq allClasses(MM) = classesAux(allPackages(MM)) .
```

9.2.3 Model

Section 6.4.1 defined accessible features for properties π_{MM} and operations ω_{MM} for a metamodel MM: it gathered features by traversing up the class inheritance hierarchy. We defined two specialised operators`allAttributes` and `allReferences` (and also `allContainingReferences`, doing the same job as `allReferences` but filtering out non-containment references); and another one, `allStructuralFeatures`, that just gather both.

```
op allReferences : MyList -> MyList . --- Of @Class, @Reference
eq allReferences(nil) = nil .
eq allReferences(C) = (references(C) allReferences(superTypes(C))) .
eq allReferences(C L) = (allReferences(C) allReferences(L)) .

op containmentReferences : @Class -> MyList . --- Of @References
eq containmentReferences(C) = containmentReferences(allReferences(C)) .
op containmentReferences : MyList -> MyList . --- Of @References (both)
eq containmentReferences(nil) = nil .
eq containmentReferences(REF L)
  = if (isContainment(REF))
    then (REF containmentReferences(L))
    else containmentReferences(L) fi .

op allAttributes : MyList -> MyList . --- Of @Class, @Attribute
eq allAttributes(nil) = nil .
eq allAttributes(C) = (attributes(C) allAttributes(superTypes(C))) .
eq allAttributes(C L) = (allAttributes(C) allAttributes(L)) .
```

```
op allOperations : MyList -> MyList . --- Of @Class, @Operation
eq allOperations(nil) = nil .
eq allOperations(C) = (operations(C) allOperations(superTypes(C))) .
eq allOperations(C L) = (allOperations(C) allOperations(L)) .

op allStructuralFeatures : @Class -> MyList . --- Of @StructuralFeature
eq allStructuralFeatures(C) = (allAttributes(C) allReferences(C) allOperations(C)) .
```

Section 6.4.2 defined the structure of models as a function associating an object to its type (i.e. its class) and its state, i.e. the value of each accessible property. We explain how models are represented by starting from the innermost constituant elements. The first step is to represent pairs of attribute/value or reference/value. The following constructor operators define the corresponding notions of `@ReferenceInstance` and `@AttributeInstance` using the classical notation in Object-Oriented Maude:

```
sort @AttributeInstance @ReferenceInstance @StructuralFeatureInstance .
subsort @AttributeInstance @ReferenceInstance < @StructuralFeatureInstance .

op _:_ : @Attribute OCL-Type -> @AttributeInstance [ctor ditto] .
op _:_ : @Reference OCL-Type -> @ReferenceInstance [ctor ditto] .
```

The second step consists in gathering several such pairs to build a full model's state. This is done using set constructions of the previous sorts, using `#` as a separator.

```
sort Set{@AttributeInstance} .
subsort @AttributeInstance < Set{@AttributeInstance} < Set{@StructuralFeatureInstance} .
op _#_ : Set{@AttributeInstance} Set{@AttributeInstance} -> Set{@AttributeInstance} [ctor ditto] .

sort Set{@ReferenceInstance} .
subsort @ReferenceInstance < Set{@ReferenceInstance} < Set{@StructuralFeatureInstance} .
op _#_ : Set{@ReferenceInstance} Set{@ReferenceInstance} -> Set{@ReferenceInstance} [ctor ditto] .

sort EmptySet .
subsorts EmptySet < Set{@ReferenceInstance} Set{@AttributeInstance} .
op _#_ : EmptySet EmptySet -> EmptySet [ctor ditto] .
op empty : -> EmptySet [ctor ditto] .
```

The third step consists in defining how an object is represented. We declare a new sort `@Object` and just reuse the classical notation already available for objects in Object-Oriented Maude:

```
sort @Object .
op <_:_|_> : Oid @Class @Set{StructuralFeature} -> @Object [ctor] .

sort Set{@Object} .
subsort @Object < Set{@Object} < Configuration .
op __ : Set{@Object} Set{@Object} -> Set{@Object} [ctor ditto] .
op none : -> Set{@Object} [ctor ditto] .
```

Example 9.8 (An initial State)**.** We illustrate how to build an object with the initial state of our FSM model $\mathsf{M}_{\mathsf{abc}}$. This state was named 1 and its mathematical representation was the following:

$$\begin{aligned}\sigma^1(\mathsf{label}) &= \text{"1"}\\ \sigma^1(\mathsf{kind}) &= \mathsf{START}\\ \sigma^1(\mathsf{in}) &= \langle\!\langle b\rangle\!\rangle\\ \sigma^1(\mathsf{out}) &= \langle\!\langle a\rangle\!\rangle\\ \sigma^1(\mathsf{fsm}) &= abc\end{aligned}$$

Recalling that each property name is represented with all the enclosing information (e.g., the attribute *label* of class *Label*, inherited by *State*, is named `label@Label@FSM`), we obtain the following definition:

```
< 'one : State@FSM | label@Label@FSM : "1" #
                     fsm@State@FSM : 'fsm #
                     kind@State@FSM : START@Kind@FSM #
                     in@State@FSM : Set {'b} #
                     out@State@FSM : Set {'a} >
```

Notice that object identifiers are terms of sort `Oid` that are required to be prefixed by `'`. □

Finally, it remains to define the structure of a model: we introduce a new sort `@Model`, and define the curly braced notation as follows:

```
sort @Model .
op _'{_'} : @Metamodel Set{@Object} -> @Model [ctor] .
```

Example 9.9 (The *FSM* Model)**.** Putting all previous definitions together, we would define the representation of our FSM model as follows:

```
op FSMModel : -> @Model .
eq FSMModel = FiniteStateMachine {
   ...
   < 'one : State@FSM | ... >
   ...
  } .
```

□

Two helper operators retrieve an object's type and value of a specific property, just as the functions $type_{\mathsf{M}}(o)$ and σ^{o}_{M} in the mathematical notation.

```
var C    : @Class .
var OBJ  : @Object .
var O    : Oid .
var SFI  : @StructuralFeatureInstance .
var SFIS : Set{@StructuralFeatureInstance} .
var VALUE : OCL-Type .

op meta : @Object -> @Class .
eq meta(< O : C | SFIS >) = C .

op get : @Object @StructuralFeature -> OCL-Type .
eq get(< O : C | (SF : VALUE # SFIS) >, SF) = VALUE .
eq get(OBJ, SF) = null [owise] .
```

9.2.4 Conformance

The Maude definition of conformance closely follows its mathematical definition. A model conforms to its metamodel if all objects it contains are valid. An object is conform if its constituting attributes and references are valid.

```
op conformsTo : @Model -> Bool .
eq conformsTo(MODEL) = validObject(MODEL, MODEL) .

op validObject : @Model @Model -> Bool .
eq validObject(MM { none }, MODEL) = true .
eq validObject(MM { < O : C | (ATTIS # REFIS) > OBJSET }, MODEL) =
   validReferences(allReferences(C), < O : C | REFIS >, MODEL)) and-then
   validAttributes(allAttributes(C), < O : C | ATTIS >)) and-then
   validObject(MM { OBJSET }, MODEL)) .
eq validObject(MM { OBJSET }, MODEL) = false [owise] .
```

A model conforms to its metamodel if all objects it contains are valid. This is achieved by induction on the list of objects within the model's representation: the empty model is valid; otherwise, it is possible to extract one object `< O : C | (ATTIS # REFIS)>`. This object is valid if all attributes and references it contains are valid. This is achieved using specialised operators `validAttributes` and `validReferences`. Notice that the checking can stop immediately after an error is discovered, thus the use of the boolean operator `and-then`.

A list of attributes is valid if all its attributes are valid (again, a structural induction). An attribute `ATT : VALUE` is valid if its declared multiplicity and collection match those of its value, and if the `VALUE`'s type is a subtype of the declared `ATT`'s type. Notice that because `VALUE` can be a collection value (constituted of many elementary values), we need to perform the check for each of them (notice that an empty collection of value always satisfies the required subtyping).

```
op validAttributes : MyList @Object -> Bool .
eq validAttributes(nil, < O : C | empty >) = true .
eq validAttributes((ATT ATTS), < O : C | (ATT : VALUE # ATTIS) >) =
   ((((((isUnique(ATT) == isUnique(VALUE))
   and-then (isOrdered(ATT) == isOrdered(VALUE)))
   and-then (isMany(ATT) == isMany(VALUE)))
   and-then (lowerBound(ATT) <=Card size(VALUE)))
   and-then (size(VALUE) <=Card upperBound(ATT)))
   and-then (<< VALUE -> asSequence() -> isEmpty() >> or-else
            subTypeOf(meta(VALUE), type(ATT))))
   and-then validAttributes(ATTS, < O : C | ATTIS >)) .
eq validAttributes(ATTS, < O : C | ATTIS >) = false [owise] .
```

Checking the validity of a list of references is almost the same, except that instead of a subtyping checking on datatypes, it happens for references for classes: we introduce the operator `validRefType` for that purpose. Another notable difference resides in handling the opposites. We introduce an extra operator `validOpposites` that checks that all opposite objects are eventually included in a SRC collection value.

```
op validReferences : MyList @Object @Model -> Bool .
eq validReferences(nil, < O : C | empty >, MODEL) = true .
eq validReferences((REF REFS), < O : C | (REF : VALUE # REFIS) >, MODEL) =
   (((((((isUnique(REF) == isUnique(VALUE))
   and-then (isOrdered(REF) == isOrdered(VALUE)))
   and-then (isMany(REF) == isMany(VALUE)) )
   and-then (lowerBound(REF) <=Card size(VALUE)))
   and-then (size(VALUE) <=Card upperBound(REF)))
   and-then validRefType( << VALUE -> asSequence() >>, type(REF), MODEL))
   and-then ((opposite(REF) == null) or-else
     validOpposites(<< VALUE -> asSequence() >>, opposite(REF), O, MODEL)))
   and-then validReferences(REFS, < O : C | REFIS >, MODEL)) .
eq validReferences(REFS, < O : C | REFIS >, MODEL) = false [owise] .

op validRefType : Sequence @Class @Model -> Bool .
eq validRefType(Sequence{mt-ord}, C, MODEL) = true .
eq validRefType(Sequence{O # LO}, C', MM { < O : C | SFS > OBJSET }) =
   (isSubClass(C, C')
   and-then validRefType(Sequence{LO}, C',
     MM { < O : C | SFS > OBJSET })) .
eq validRefType(SEQ, C, MODEL) = false [owise] .

op validOpposites : Sequence @Reference Oid @Model -> Bool .
eq validOpposites(Sequence{mt-ord}, REF, SRC, MODEL) = true .
eq validOpposites(Sequence{O # LO}, REF, SRC,
                  MM { < O : C | (REF : VALUE # SFS) > OBJSET }) =
   (<< VALUE -> asSequence() -> includes(SRC) >>
   and-then validOpposites(Sequence{LO}, REF, SRC,
       MM { < O : C | (REF : VALUE # SFS) > OBJSET })) .
eq validOpposites(SEQ, REF, O, MODEL) = false [owise] .
```

9.3 Specifying Kermeta's Action Language

The Maude specification for the Action Language of Kermeta closely follows the mathematical definitions given in Chapter 7. The specification consists of building a syntax for the elements needed as components of the SOS rules (control flow, local environment, semantic domain and rule configurations), as well as the BNF syntax itself, in order to be able to parse Kermeta statements.

As already mentioned, Maude has a very flexible parser that allows the definition of customised syntaxes for representing and manipulating data structures (Quesada Moreno 1997, 1999). Nevertheless, we tried in our specification to follow as closely as possible the mathematical syntax, customising data structures only to avoid too much ambiguities for the parser: in particular, we have not followed the usual mathematical syntax for tuples because it was raising too much parsing problems and subject to errors when writing even small examples[4]. Therefore, all parenthesised structure has some "variations" (e.g., brackets, chunks, etc.) to remind

[4]Beyond the flexible syntax, Maude's algebraic specifications rely on order-sorted terms, which is very powerful, but at the same

the reader the original syntax while suppressing parsing issues. This kind of parsing issues is fixable in Maude for obtaining a "perfect" syntax, but asks for too much effort only due to technicalities. This was one of the main reasons why Part II uses mathematics instead of a particular tool: the mathematical syntax does not need so much explanation.

We present the elements constituting the Action Language in the same order as in the reference semantics: we start with the *local variables*, then the *statement syntax*, and finish by the *semantic domain* and the *rule configurations*. A last section describes the semantic rules by means of equations.

9.3.1 Local Variables

Local variables were defined in Section 7.2.1: they are located inside a particular operation, and possess a multiple type. In Maude, we introduce a sort `@Variable` and treat variables just like any other structural feature contained within a class: an operation `variables` associates to an operation the list of its variables; and an operation `containingOperation` allows to retrieve the operation a variable is defined into.

```
sort @Variable @LocalVariable .
subsort @LocalVariable < @StructuralFeature .
subsort @LocalVariable < Vid .
op containingOperation : @Variable -> @Operation .
op variables : @Operation -> MyList . --- of @Variable
```

Note that we also use a sort `@LocalVariable`, just as in the BNF of Figure 7.2: `@LocalVariable` will represent a supersort for both variables and parameters, and is declared as a subsort of the mOdCL sort `Vid` in order to be evaluated properly within expressions.

Example 9.10 (Local Variables in the accept operation)**.** Let us consider the operation `accept` in class `FSM`, which contains a variable `toEval` declared as follows: `var toEval : seq String[0..*]`. This variable would be specified as follows (a similar construction is applied for operation parameters):

```
1   sort toEval@accept@FSM@FSM .
2   op toEval@accept@FSM@FSM : -> toEval@accept@FSM@FSM [ctor] .
    subsort toEval@accept@FSM@FSM < @Variable .
4   eq name(toEval@accept@FSM@FSM) = "toEval" .
    eq type(toEval@accept@FSM@FSM) = @String .
6   eq lowerBound(toEval@accept@FSM@FSM) = 0 .
    eq upperBound(toEval@accept@FSM@FSM) = * .
8   eq isOrdered(toEval@accept@FSM@FSM) = true .
    eq isUnique(toEval@accept@FSM@FSM) = false .
10  eq containingOperation(toEval@accept@FSM@FSM) = accept@FSM@FSM .
```

The same construction as for metamodel elements is adopted here: for each variable, a sort named with the "qualified" name of the variable is defined with `@` as a separator (Line 1) and declared as being a subsort of `@Variable` (Line 3), as well as a constructor operator with the same "qualified" name (Line 2). is declared as a subsort of `@Variable`, together with its corresponding constructor (Lines 1–3). Then, several equations capture the variable declarations (name and type in Line 4 and 5, and multiplicity in Lines 6–9). Finally, the sort is bound to its containing operation (Line 10). □

9.3.2 Statement Syntax

The BNF of Section 7.2.2 needs a representation in Maude. Here, it is possible to fully reuse the syntax for mOdCL expressions, since we have clearly separated expressions from statements. Note how some of the constructions

time very tricky from a syntactic viewpoint. Consider for example a binary operator `op '(_,_'): A B -> C` that follows the usual mathematical syntax for pairs. Then, if we need another pair for building a different element D from completely different sorts, we would similarly define `op '(_,_'): D E -> F`. This works perfectly fine in Maude, if sorts `A`, `B`, `D`, `E` are order-sorted, this becomes very ambiguous and very difficult to debug.

do not strictly follow our BNF of Figure 7.2: the conditional statement (Line 16) and the assignment symbol (Lines 17, 18 and 24), which clashes with predefined constructions in Maude; and the parenthesis of operation calls (Lines 22–23), which are too ambiguous, are replaced by chunks. However, we preserved the syntactic groups (Line 7) to facilitate the recognition of the operators, thus helping to match BNF terminals to their representation in Maude.

```
 1 mod KERMETA-AL is
 2    pr KERMETA-STMT-PREP .
      pr KERMETA-SL .
 4    pr METAMODEL-PROP .
      pr MGMAYBE{@Operation} .
 6
      sort    @CondStmt @AssignStmt @InstanceCreationStmt @Call @CallStmt @ReturnStmt @Statement .
 8    sort    @CollItem .
      subsort @Call < @CallStmt .
10    subsort @CondStmt @AssignStmt @InstanceCreationStmt @CallStmt @ReturnStmt < @Statement .

12    ops bag
         set
14       seq
         oset : -> @CollItem .
16    op iff_             : OCL-Exp -> @CondStmt                                  [ctor] .
      op _.:=._           : OCL-Exp OCL-Exp -> @AssignStmt                        [ctor] .
18    op _.:=. new‘(_‘)   : @Variable @Classifier -> @InstanceCreationStmt        [ctor] .
      op _.:=. new‘(_,_‘) : @Variable @CollItem @Classifier -> @InstanceCreationStmt [ctor] .
20    op return           : -> @ReturnStmt                                        [ctor] .
      op return_          : OCL-Exp -> @ReturnStmt                                [ctor] .
22    op _._<>            : OCL-Exp String -> @Call                               [ctor] .
      op _._<_>           : OCL-Exp String List{OCL-Exp} -> @Call                 [ctor] .
24    op _.:=._           : OCL-Exp @Call -> @CallStmt                            [ctor] .
      --- [...]
26 endm
```

9.3.3 Control Flow

In Section 7.2.3, we defined a function $\mathrm{nxt}_{\mathrm{MM}}$ (cf. Definition 7.2) for capturing a transformation's control flow, under the following assumptions: an label uniquely identify each statement of the metamodel; variable declarations inside an operation body are shifted at its beginning; and blocks are flattened.

We define a sort `@Label` constructed in a specific way (Line 1–2) to ensure label uniqueness: we use the sort names for packages, classes and operations, which are unique up to their respective scope, and an integer denoting the statement's rank in an operation body. The $\mathrm{nxt}_{\mathrm{MM}}$ function has a cartesian product as a codomain, translated in Maude with a new constructor `<_,_>` that builds terms of sort `@LabelNxt` from a pair of `@Label`s (Line 5). This way, the corresponding operator `nxt` has the exact same definition as the function $\mathrm{nxt}_{\mathrm{MM}}$. Note that an operation `labels` allows to list all labels of statements contained in a given operation (Line 7), and an operation `statements` defines the statements attached to each label within a metamodel as a map (Line 8).

```
1   sort @Label @LabelNxt .
2   op [_,_,_,_] : @Package @Class @Operation Nat -> @Label [ctor] .
    --- default label
4   op [] : -> @Label .
    op <_,_> : @Label @Label -> @LabelNxt [ctor] .
6   op nxt : @Label -> @LabelNxt .
    op labels : @Operation -> MyList . --- of @Label
8   op statements : @Metamodel -> Map{@Label, @Statement} .
```

The default label `[]` is necessary to handle the case where no actual label is needed. For example for a ReturnStmt, no next statement is needed: we represent that as `nxt([pkg, c, op, _])= <[],[]>`.

Example 9.11 (Statements & Control Flow)**.** We provide now the encoding of the body of the `getStart` operation in class `FSM` (remember that this code is obtained after the restructurations of explained in Section 7.2.3).

```
 1
 2 eq statements(FiniteStateMachine) =
      [FSM, FSM@FSM, getStart@FSM@FSM,  1] |-> i@getStart@FSM@FSM .:=. 0,
 4    [FSM, FSM@FSM, getStart@FSM@FSM,  2] |-> i@getStart@FSM@FSM .:=. i@getStart@FSM@FSM + 1,
      [FSM, FSM@FSM, getStart@FSM@FSM,  3] |-> iff( (i@getStart@FSM@FSM .=. states@FSM@FSM -> size()) or
 6                      (states@FSM@FSM -> at(i@getStart@FSM@FSM) . kind@State@FSM .=. START@Kind@FSM)),
      [FSM, FSM@FSM, getStart@FSM@FSM,  4] |-> iff(i@getStart@FSM@FSM > states@FSM@FSM -> size()),
 8    [FSM, FSM@FSM, getStart@FSM@FSM,  5] |-> return null,
      [FSM, FSM@FSM, getStart@FSM@FSM,  6] |-> return (states@FSM@FSM -> at(i@getStart@FSM@FSM)),
10    --- [mapping for other statements]
   .
```

Note here that the calls for built-in collection operations (`size()` at in Lines 3 and 5 and `at()` at Line 9) is done with `->` instead of the Kermeta dot notation: this comes from mOdCL, which follows the syntax of OCL. The control flow that corresponds to these statements is the following (remember that this code comes from a flattened `size()`do...| loop): only statements with specific flow are defined explicitly, the "normal" statements just jump to the next label (Line 8).

```
1 var N : Nat .
2 eq nxt([FSM, FSM@FSM, getStart@FSM@FSM,  3]) = <[FSM, FSM@FSM, getStart@FSM@FSM,  4],
                                                  [FSM, FSM@FSM, getStart@FSM@FSM,  2]> .
4 eq nxt([FSM, FSM@FSM, getStart@FSM@FSM,  4]) = <[FSM, FSM@FSM, getStart@FSM@FSM,  5],
                                                  [FSM, FSM@FSM, getStart@FSM@FSM,  6]> .
6 eq nxt([FSM, FSM@FSM, getStart@FSM@FSM,  5]) = <[], []> .
  eq nxt([FSM, FSM@FSM, getStart@FSM@FSM,  6]) = <[], []> .
8 eq nxt([FSM, FSM@FSM, getStart@FSM@FSM,  N]) = <[FSM, FSM@FSM, getStart@FSM@FSM,  (N + 1)], []> [owise] .
```

□

9.3.4 Configuration

It remains to define how elements of the sets of semantic domains $\mathbb{D}$ and of configurations Γ (cf. Sections 7.4.1 and 7.4.2, respectively) are defined: the Maude specifications just mimic the mathematical definitions except for Γ:

```
 1 mod KERMETA-DOMAIN is
 2    pr MGmOdCL .
      pr METAMODEL-PROP .
 4    pr KERMETA-SL-CTORS .

 6    sort Domain .
      op <_##_> : @Model Set{VarPair} -> Domain [ctor] .
 8 endm

10 mod KERMETA-CONFIGURATION is
      pr KERMETA-SL .
12    pr STACK{StackEntry} .
      pr KERMETA-DOMAIN .
14    pr OCL-TYPE .

16    sort KConfig .
      op <|_,_,_,_|> : @Label Stack{StackEntry} Domain Nat -> KConfig [ctor] .
18    op isStop : KConfig -> Bool .
      eq isStop(<| [], NOPE, D, N |>) = true .
20    eq isStop(<| LAB, S, D, N |>) = false [owise] .
   endm
```

The sort `KConfig` (the name has been changed to avoid conflicts with the predefined `Configuration` sort) has a slightly different constructor (Line 18): an extra element of sort `Nat` acts like a global variable that enables the creation of fresh identifiers, a functionality needed for the NewInstStmt statement. The sort `Set{VarPair}`corresponds to our local environment set $\mathbb{L}$ (cf. Definition 7.4): imported from mOdCL, it maps `@Variable` to an `OCL-Type` value.

The construction of a `KConfig` term necessitates a `Stack` element: we use a syntax-customised version of this classical datastructure (cf. Martí-Oliet, Palomino, and Verdejo 2005). An element of this stack corresponds to the sort `StackEntry`, which basically corresponds to elements of the set $\mathbb{E}$nv (cf. Section 7.4.2).

```
mod KERMETA-STACK is
   pr KERMETA-SL .
   pr KERMETA-STATEMENTS-MAP .
   pr MGMAYBE{@LocalVariable} .

   sort StackEntry .
   op `(|_,_,_|`) : @Label Set{VarPair} Maybe{@LocalVariable} -> StackEntry .
endm

fmod STACK{X :: TRIV} is
   protecting BOOL .

   sorts NeStack{X} Stack{X} .
   subsort X$Elt  < NeStack{X} < Stack{X} .

   op NOPE : -> Stack{X} [ctor] .
   op _!!!_ : X$Elt Stack{X} -> NeStack{X} [ctor right id: NOPE] .

   var E : X$Elt .
   var S : Stack{X} .

   op isEmpty_ : Stack{X} -> Bool .
   eq isEmpty(NOPE) = true .
   eq isEmpty(S)    = false [owise] .
endfm

view StackEntry from TRIV to KERMETA-STACK is
   sort Elt to StackEntry .
endv
```

Several examples will be given in the next Section, where domains and configurations are used within rules that define the effect of each statement.

9.3.5 Semantics

Once all the machinery is ready for representing the semantics components, comes the time to translate into Maude the semantic rules of Section 7.4.3. We first explain how expressions are evaluated, then specify the rules for executing each statement.

9.3.5.1 Evaluating Expressions

The expressions defined in Section 7.2 share a lot with OCL expressions: a natural idea would then consist in simply reusing mOdCL evaluator. This requires to adapt the evaluator to our datastructures since they differ from the ones at the basis of mOdCL, which are customised for UML.

We provide two operators for evaluating expressions: the first one simply takes an expression to be evaluated in the context of a model; whereas the second one adds an environment that binds local variables with their values.

```
   op <<_;_>> : OclExp @Model -> OclExp .
   eq << EXP:OclExp ; MM { OBJSET } >> = .
      eval-aux(EXP:OclExp, env(empty) OBJSET, none ) .

   --- The second argument are the context variables
   op <<_;_;_>> : OclExp Msg @Model -> OclExp .
   eq << EXP:OclExp ; env(VARS:Set{VarPair}) ; MM { OBJSET } >> =
      eval-aux(EXP:OclExp, env(VARS:Set{VarPair}) OBJSET, none ) .
```

The first one is equivalent to the second, since it evaluates an expression under an empty environment (actually, an environment always maps the local variable `self` to the current object). The second one is very simple: it calls the mOdCL dedicated operator `eval-aux` with the proper values, i.e. it extracts the set of objects `OBJSET` constituting the metamodel and passes it to the mOdCL evaluator.

Example 9.12 (Evaluating Expressions)**.** Let us evaluate two expressions extracted from the condition of the first conditional instruction in the body of the Kermeta operation `getStart`: this expression is actually extracted from the loop that goes through the set of states contained in the FSM, looking for a state whose kind is START.

```
 1 reduce in FSM-MODEL : << states@FSM@FSM -> size() ;
 2    env(putVar(i@getStart@FSM@FSM <- 1, putVar(self <- 'fsm, empty))) ;
      FSMModel >> .
 4 rewrites: 27 in 5479628793ms cpu (0ms real) (0 rewrites/second)
   result NzNat: 3
 6
   reduce in FSM-MODEL :
 8    << (i@getStart@FSM@FSM .=. states@FSM@FSM -> size()) or
         (states@FSM@FSM -> at(i@getStart@FSM@FSM) . kind@State@FSM .=. START@Kind@FSM) ;
10       env(putVar(i@getStart@FSM@FSM <- 0, putVar(self <- 'fsm, empty))) ;
         FSMModel >> .
12 rewrites: 68 in 5506796793ms cpu (0ms real) (0 rewrites/second)
   result Bool: false
14
   reduce in FSM-MODEL :
16    << (i@getStart@FSM@FSM .=. states@FSM@FSM -> size()) or
         (states@FSM@FSM -> at(i@getStart@FSM@FSM) . kind@State@FSM .=. START@Kind@FSM) ;
18       env(putVar(i@getStart@FSM@FSM <- 3, putVar(self <- 'fsm, empty))) ;
         FSMModel >> .
20 rewrites: 68 in 5506796793ms cpu (0ms real) (0 rewrites/second)
   result Bool: true
```

We recognise in the first reduction the three elements required by the evaluation operator: the expression on Line 1; the environment in Line 2 and the current model in Line 3. After a few rewriting, we get the expected answer of `i@getStart@FSM@FSM`, which corresponds to the number of states of our model.

The second reduction evaluates the actual condition of the loop: it compares the kind of the state stored at the position `i@getStart@FSM@FSM` to see if it has a `START@Kind@FSM`. Unfortunately, it appears that it is not at position `0`, but at position `3`, value that is assigned to the variable `i@getStart@FSM@FSM` at the beginning of the environment. □

9.3.5.2 Executing Statements

We define an operation `op exec_ : KConfig -> KConfig .` for representing one execution step: this mimics the trigger of one rule in the SOS of Section 7.4.3, which rewrites a configuration into another one. The translation follows the core principle of an SOS: at each time, exactly one rule should be triggered, the one corresponding to the statement to-be-executed. The conditions also define what is traditionally written in the premisse of SOS rules, so that Maude equations look very like the mathematical definitions. We take advantage of Maude matching capabilities to bind variables with their adequate values to avoid expanding too many elements from the configuration.

9.3.5.2.1 Conditional Statement After having matched the statement's form[5], the rule simply rewrites the configuration into an identical one, except for the label: it depends on the value of the conditional expression `E` (Line 5 and 12) that helps select the appropriate label defined by `nxt` (Line 4 and 11).

```
1 ceq exec( <| LAB, S, < M ## VPSET >, N |> ) = <| LABTHEN, S, < M ## VPSET >, N |>
2 ---------------------------------------------------------------------------------
     if  iff( E ) := statements( meta(M) ) [ LAB ] /\
4       < LABTHEN , LABELSE > := nxt(LAB)          /\
        true = << E ; env(VPSET) ; M >>
6 .
```

[5]From now on, we will not repeat that!

```
 8   ceq exec(<| LAB, S, < M ## VPSET >, N |>) = <| LABELSE, S, < M ## VPSET >, N |>
     ------------------------------------------------------------------------------------
10      if  iff( E ) := statements( meta(M) ) [ LAB ] /\
           < LABTHEN  , LABELSE > := nxt(LAB)        /\
12         false = << E ; env(VPSET) ; M >>
     .
```

9.3.5.2.2 New Instance Creation The rule rewrites into a configuration where the label corresponds to the left-hand side of the value stored in `nxt`, the stack stays the same, and the domain is updated: if the creation concerns a `@DataType` (Lines 1–6), then only the variable in the local store is updated with the new value; if it concerns a simple `@Classifier` (Lines 7–12), then the variable is updated with a new object that is added to the model; if it concerns a `@Classifier` with a collection (Lines 13–17), then again only the variable is updated with the corresponding default value.

```
 1   ceq exec(<| LAB,    S, < MM { OBJSET } ## VPSET >, N |>) =
 2          <| LABNxt, S, < MM { OBJSET } ## (putVar(MYVAR <- defaultValue(CLASSIFIER), VPSET)) >, N |>
        if  MYVAR .:=. new ( CLASSIFIER ) := statements( MM ) [ LAB ] /\
 4         < LABNxt  , LABNULL > := nxt(LAB) /\
           CLASSIFIER :: @DataType
 6   .
     ceq exec(<| LAB,    S, < MM { OBJSET } ## VPSET >, N |>) =
 8          <| LABNxt, S, < MM { complete(< newOid(N) : CLASSIFIER | empty >) OBJSET } ## (putVar(MYVAR <-
              newOid(N), VPSET)) >, (N + 1) |>
        if  MYVAR .:=. new ( CLASSIFIER ) := statements( MM ) [ LAB ] /\
10         < LABNxt  , LABNULL > := nxt(LAB)                         /\
           CLASSIFIER :: @Class
12   .
     ceq exec(<| LAB, S, < MM { OBJSET } ## VPSET >, N |>) =
14         <| LABNxt, S, < MM { OBJSET } ## (putVar(MYVAR <- default(COLLITEM , CLASSIFIER), VPSET)) >, N |>
        if  MYVAR .:=. new ( COLLITEM , CLASSIFIER ) := statements( MM ) [ LAB ] /\
16         < LABNxt  , LABNULL > := nxt(LAB)
     .
```

The operations `defaultValue` (Line 2) and `default` (Line 14) just implement the $\mathfrak{Default}$ function of Section 7.4.1.1. In Line 8, the operation `newOid` creates a fresh Maude object identifier of the form o_{N}, and `N` is incremented to ensure uniqueness (Line 9). The fresh object is then set to its default value through the operation `complete`, which implements the function $\mathfrak{Initialise}$ defined in Section 7.4.1.1, and assigned to `MYVAR`.

```
 1   op newOid : Nat -> Oid .
 2   var N : Nat .
     eq newOid(N) = qid("O" + string(N, 10)) .
```

9.3.5.2.3 Return Statement The rules for the ReturnStmt are complicated by the fact that they are in charge of detecting that the execution terminates, i.e. there is no more statements to execute (cf. Section 7.4.3). Lines 1–11 correspond to “normal” statements whereas Lines 12–21 correspond to execution termination: both are similar in their principle. We aligned the configurations to help the reading: we can see that the top element of the stack is removed, and the execution is transfered to the label `LABNxt` that was contained in it; the old local store `VPSETT` is eventually updated with the value resulting from evaluating the return value `E` (Lines 10 and 20).

```
 1   ceq exec(<| LAB,   (| LABNxt  , VPSETT, null |) !!! S, < M ## VPSET >,   N |>) =
 2          <| LABNxt,                                  S, < M ## VPSETT >, N |>
     ------------------------------------------------------------------------------------
 4      if return :=  statements( meta(M) ) [ LAB ]
     .
 6   ceq exec(<| LAB, (| LABNxt  , VPSETT, MYVAR |) !!! S, < M ## VPSET >,                       N |>) =
            <| LABNxt,                                 S, < M ## putVar( MYVAR <- RES  , VPSETT) >, N |>
 8   ----------------------------------------------------------------------------------------------------
        if  return( E ) :=  statements( meta(M) ) [ LAB ] /\
10                RES := << E ; env(VPSET) ; M >>
     .
12   ceq exec(<| LAB, NOPE, < M ## VPSET >,  N |>) =
            <| [] , NOPE, < M ## VPSET >,  N |>
14   ------------------------------------------------------------------------------------
        if return :=  statements( meta(M) ) [ LAB ]
16   .
```

```
  ceq exec(<| LAB, NOPE, < M ## VPSET >,                        N |>) =
        <| [] , NOPE, < M ## putVar( KRESULT <- RES , VPSET) >, N |>
     if  return( E ) :=  statements( meta(M) ) [ LAB ] /\
                RES := << E ; env(VPSET) ; M >>
  .
  op KRESULT : -> @LocalVariable .
```

When a transformation terminates, we return a KConfiguration with a default label [] and a special variable KRESULT containing the last computed value: this would correspond to the value the Kermeta main operation would have returned. The resulting model is still accessible inside the last KConfiguration.

9.3.5.2.4 Call Statement For CallStmt, the Maude implementation follows the rule split defined in Section 7.4.3.4 two rules for operation calls: one for a simple call (Lines 1–15) and another for storing the value of a call inside a variable (Lines 16–30), except that the implementation has to match whether the parameter list is empty or not. The rules are pretty similar, so we only explain the most complicated one (Lines 23–30).

```
1    ceq exec(<| LAB, S, < MM { < OID : CLASS | SFIS > OBJSET } ## VPSET >, N |>) =
2          <| [ package(containingClass(OP)) , containingClass(OP) , OP , 1 ] , (| LABNxt , VPSET, null |)
               !!! S, < MM { < OID : CLASS | SFIS > OBJSET } ## (putVar(self <- OID, createLocalEnv(OP))) >,
               N |>
       if INST . OPNAME <> := statements( MM ) [ LAB ] /\
4      < LABNxt , LABNULL > := nxt(LAB) /\
       OID := << INST ; env(VPSET) ; MM { < OID : CLASS | SFIS > OBJSET } >> /\ OID :: Oid /\
6      OP := lookup(OPNAME, CLASS)
     .
8    ceq exec(<| LAB, S, < MM { < OID : CLASS | SFIS > OBJSET } ## VPSET >, N |>) =
           <| [ package(containingClass(OP)) , containingClass(OP) , OP , 1 ] , (| LABNxt , VPSET, null |)
               !!! S, < MM { < OID : CLASS | SFIS > OBJSET } ## (putVar(self <- OID, createLocalEnv(OP, LEXP)
               )) >, N |>
10     if INST . OPNAME < LO > := statements( MM ) [ LAB ] /\
       < LABNxt , LABNULL > := nxt(LAB) /\
12     LEXP := eval-EL(LO, env(VPSET), none) /\
       OID := << INST ; env(VPSET) ; MM { < OID : CLASS | SFIS > OBJSET } >> /\ OID :: Oid /\
14     OP := lookup(OPNAME, CLASS)
     .
16   ceq exec(<| LAB, S, < MM { < OID : CLASS | SFIS > OBJSET } ## VPSET >, N |>) =
           <| [ package(containingClass(OP)) , containingClass(OP) , OP , 1 ] , (| LABNxt , VPSET, MYVAR |)
               !!! S, < MM { < OID : CLASS | SFIS > OBJSET } ## (putVar(self <- OID, createLocalEnv(OP))) >,
               N |>
18     if MYVAR .:=. INST . OPNAME <> := statements( MM ) [ LAB ] /\
       < LABNxt , LABNULL > := nxt(LAB) /\
20     OID := << INST ; env(VPSET) ; MM { < OID : CLASS | SFIS > OBJSET } >> /\ OID :: Oid /\
       OP := lookup(OPNAME, CLASS)
22   .
     ceq exec(<| LAB, S, < MM { < OID : CLASS | SFIS > OBJSET } ## VPSET >, N |>) =
24         <| [ package(containingClass(OP)) , containingClass(OP) , OP , 1 ] , (| LABNxt , VPSET, MYVAR |)
               !!! S, < MM { < OID : CLASS | SFIS > OBJSET } ## (putVar(self <- OID, createLocalEnv(OP, LEXP)
               )) >, N |>
       if MYVAR .:=. INST . OPNAME < LO > := statements( MM ) [ LAB ] /\
26     < LABNxt , LABNULL > := nxt(LAB) /\
       LEXP := eval-EL(LO, env(VPSET), none) /\
28     OID := << INST ; env(VPSET) ; MM { < OID : CLASS | SFIS > OBJSET } >> /\ OID :: Oid /\
       OP := lookup(OPNAME, CLASS)
30   .
```

In Line 28, the parameter list is evaluated, returning a list LEXP of expressions. In Line 28, we retrieve the instance OID on which the call is performed. Line 29 performs the dynamic lookup of the appropriate operation OP. The KConfiguration is rewritten with the following elements: the label corresponds to the first statement found in the body of OP (the label is properly build using the helper operations containingClass and package, Line 24); an element formed with the the following label LABNxt, the current local environment VPSET and the left-hand side MYVAR is pushed on the stack (Line 24); and a new local environment is build, binding self to the evaluated caller object OID with proper default values bound to the local variables (Line 25, where createLocalEnv corresponds to the function 𝔖𝔱𝔞𝔯𝔱 defined in Section 7.4.1.2). The lookup operation implements the lookup of the appropriate operation term based on the name OPNAME provided in the call: in Kermeta, we have to look in the entire class hierarchy (due to multiple inheritance) but fortunately, there is no operation overloading so that an operation name in unique inside a class (details can be found in Appendix B: `lookup` is not detailed since it is very long because it uses many intermediate operators for implementing those various aspects).

9.3.5.2.5 Assignment Statement For AssignStmt, we used in Section 7.4.3.3 an auxiliary function $[\![\bullet,\bullet]\!]$ that was defined in Section 7.4.1.4: $[\![\bullet,\bullet]\!]$ filters on the type of the left-hand side to provide the appropriate behaviour, thus ensuring model consistency. We define three rules that basically corresponds to the function cases: when the left-hand side is a local variable (Lines 1–6 corresponding to case (ii-1) in Section 7.4.1.4); when it is a DataType (Lines 7–14 for case (i)) for accessing an attribute; and finally when it is a reference (Lines 15–22 for case (ii-2), the most complicated).

```
1   ceq exec(<| LAB ,    S, < MM { OBJSET } ## ((MYVAR <- OLD) # VPSET) >,      N |>) =
2            <| LABNxt , S, < MM { OBJSET } ## (putVar(MYVAR <- RES, VPSET)) >, N |>
          if  MYVAR .:=. E := statements( MM ) [ LAB ] /\
4            < LABNxt , LABNULL > := nxt(LAB) /\
             RES := << E ; env((MYVAR <- OLD) # VPSET) ; MM { OBJSET } >>
6   .
    ceq exec(<| LAB ,    S, < MM { < O : CLASS | ATT : OLD # SFIS > OBJSET } ## VPSET >, N |>) =
8            <| LABNxt , S, < MM { < O : CLASS | ATT : RES # SFIS > OBJSET } ## VPSET >, N |>
          if  INST . ATT .:=. E := statements( MM ) [ LAB ] /\
10           < LABNxt , LABNULL > := nxt(LAB) /\
             O   := << INST ; env(VPSET) ; MM { < O : CLASS | ATT : OLD # SFIS > OBJSET } >> /\
12           O   :: Oid /\
             RES := << E    ; env(VPSET) ; MM { < O : CLASS | ATT : OLD # SFIS > OBJSET } >>
14  .
    ceq exec(<| LAB ,    S, < MM { < O : CLASS | REF : OLD # SFIS > OBJSET } ## VPSET >, N |>) =
16           <| LABNxt , S, < update(MM { < O : CLASS | REF : RES # SFIS > OBJSET }, O, REF, RES) ## VPSET >,
                 N |>
          if  INST . REF .:=. E := statements( MM ) [ LAB ] /\
18           < LABNxt , LABNULL > := nxt(LAB) /\
             O   := << INST ; env(VPSET) ; MM { < O : CLASS | REF : OLD # SFIS > OBJSET } >> /\
20           O   :: Oid /\ --- not(isMany(REF)) /\ opposite(REF) == null /\
             RES := << E ; env(VPSET) ; MM { < O : CLASS | REF : OLD # SFIS > OBJSET } >>
22  .
```

In the first case (Lines 1–6), if the left-hand side `MYVAR` matches a variable inside the local environment, it is simply updated by `RES`, the value returned by the evaluation of the right-hand side `E` (Line 5). Otherwise, we have to match a dotted expression as a left-hand side. Here, Maude matching mechanism and sorting system becomes handy: we easily know what kind of property is navigated thanks to the sort of the dotted element and the following variable declarations `var ATT : @Attribute` and `var REF : @Reference`. The rest is straightforward: we evaluated `INST` to retrieve the object `O` being navigated (Lines 11–12 and 19–20) and update the navigated property inside the model (Lines 8 and 16–17). Notice the use of the operation `update` for references (Line 16) that implements the specificities corresponding to case (ii-2) depicted in Figure 7.5.

```
1   op update : @Model Oid @Reference OCL-Exp -> @Model .
2
    vars MM : @Metamodel .
4   vars O O' X Y : Oid .
    var SFIS SFISO SFISO' SFISX SFISY : Set{@StructuralFeatureInstance} .
6   var CLASSO CLASSO' CLASSX CLASSY : @Class .
    vars P Q : @Reference .
8   var OBJSET : Set{@Object} .
    var VALO' VALX VALY : OCL-Exp .
10
    ceq update(MM{< O : CLASSO | P : X  # SFISO > OBJSET}, O, REF, O') =
12           MM{< O : CLASSO | P : O' # SFISO > OBJSET}
       if not(isMany(REF)) /\ null = opposite(REF) .
14
    ceq update(MM{< O : CLASSO | P : X    # SFISO >
16                < Y : CLASSY | P : O'   # SFISY >
                  < X : CLASSX | Q : VALX # SFISX > OBJSET}, O, REF, O') =
18           MM{< O : CLASSO | P : O'   # SFISO >
                  < Y : CLASSY | P : null # SFISY >
20                < X : CLASSX | Q : null # SFISX > OBJSET}
       if not(isMany(P)) /\ not(isContainment(P)) /\ Q := opposite(P) .
22
```

```
  ceq update(MM{< O  : CLASSO  | P : X     # SFISO  >
                < O' : CLASSO' | Q : VALO' # SFISO' >
                < Y  : CLASSY  | P : O'    # SFISY  >
                < X  : CLASSX  | Q : VALX  # SFISX  > OBJSET}, O, REF, O') =
             MM{< O  : CLASSO  | P : O'   # SFISO   >
                < O' : CLASSO' | Q : O    # SFISO   >
                < Y  : CLASSY  | P : null # SFISO'  >
                < X  : CLASSX  | Q : null # SFISX   > OBJSET}
     if not(isMany(P)) /\ isContainment(P) /\ Q := opposite(P) .

  ceq update(MM{< O : CLASSO | P : X    # SFISO >
                < Y : CLASSY | P : O'   # SFISY >
                < X : CLASSX | Q : VALX # SFISX > OBJSET}, O, REF, O') =
             MM{< O : CLASSO | P : O'   # SFISO >
                < Y : CLASSY | P : null # SFISY >
                < X : CLASSX | Q : null # SFISX > OBJSET}
     if not(isMany(P)) /\ isContainer(P) /\ Q := opposite(P) .
```

9.4 Simulating a Model Transformation

The purpose of the previous Section was the definition of the operator `exec` for performing one rewriting step of the mathematical rewriting system defined in Section 7.4.2 according to the semantic rules of Section 7.4.3. To obtain an interpreter for Kermeta, and therefore becoming able to simulate Kermeta transformations, it remains to "chain" this operator with itself to execute it recursively starting from an initial configuration, until reaching the last statement (cf. Section 7.4.2). We defined an operator `run` that recursively call itself by applying `exec` until the final `KConfig`urigation is reached (detected through `isStop`, defined in Section 9.3.4), in which case the final `KConfig`urigation is returned (from which the user can extract the resulting model).

```
  op run : KConfig -> KConfig .
  eq run(K) = if isStop(K) then K else run(exec(K)) fi .
```

Example 9.13 (Executing the FSM)**.** Let us experiment with the execution of the FSM with various initial configurations. We check that the FSM does not accept the empty word, but accepts words of the form $(\mathsf{b}\ \mathsf{a})^\star\ \mathsf{c}$.

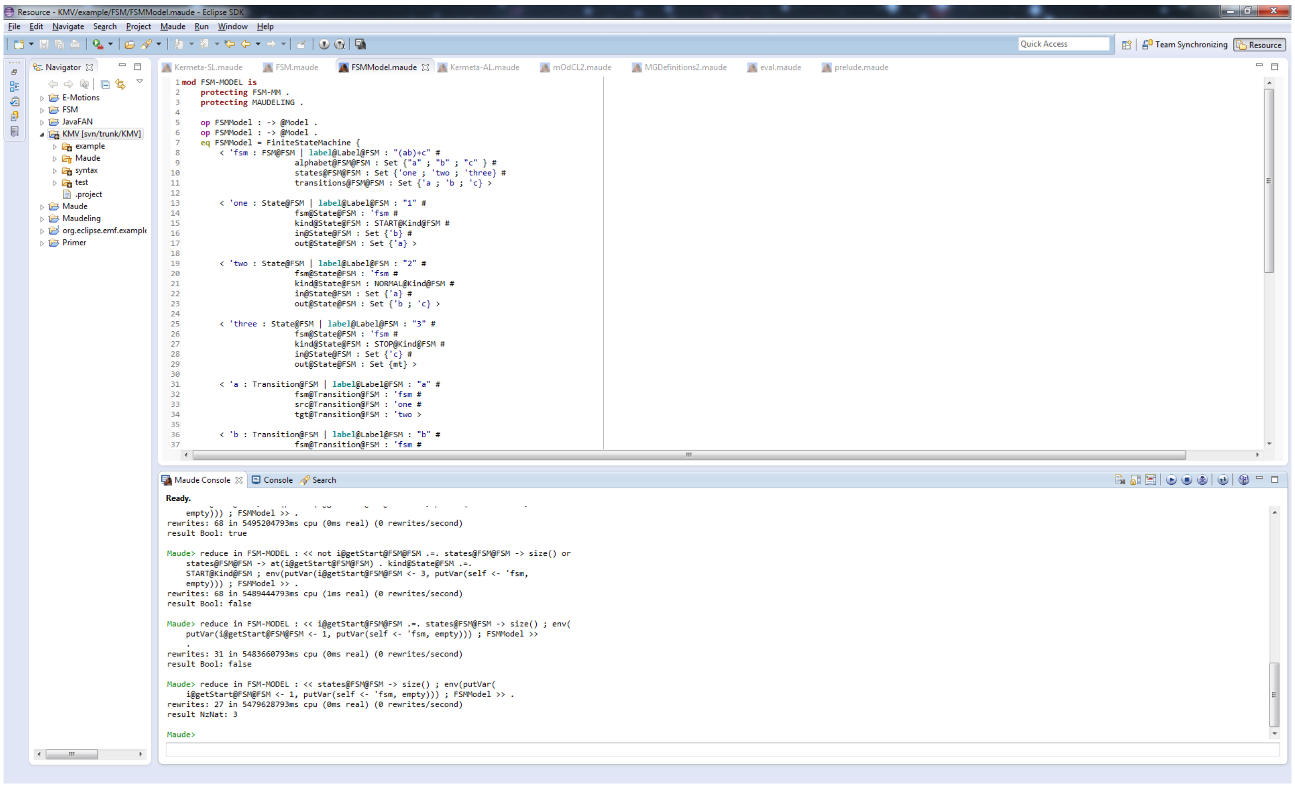
```
reduce in FSM-MODEL : run(<| [FSM, FSM@FSM, accept@FSM@FSM,  1] , (||), < FSMModel,
    putVar(word@accept@FSM@FSM <- Set{} , empty) , 0 |>) .
rewrites: 31 in 5496924793ms cpu (0ms real) (0 rewrites/second)
result Bool: false

reduce in FSM-MODEL : run(<| [FSM, FSM@FSM, accept@FSM@FSM,  1] , (||), < FSMModel,
    putVar(word@accept@FSM@FSM <- Set{"a" ; "b" ; "a" ; "b"} , empty) , 0 |>) .
rewrites: 186 in 5496924793ms cpu (0ms real) (0 rewrites/second)
result Bool: false

reduce in FSM-MODEL : run(<| [FSM, FSM@FSM, accept@FSM@FSM,  1] , (||), < FSMModel,
    putVar(word@accept@FSM@FSM <- Set{"a" ; "b" ; "a" ; "b" ; "c"} , empty) , 0 |>) .
rewrites: 218 in 5496924793ms cpu (0ms real) (0 rewrites/second)
result Bool: true

reduce in FSM-MODEL : run(<| [FSM, FSM@FSM, accept@FSM@FSM,  1] , (||), < FSMModel,
    putVar(word@accept@FSM@FSM <- Set{"a" ; "b" ; "a" ; "b" ; "a" ; "b" ; "a" ; "b" ;
    "c"} , empty) , 0 |>) .
rewrites: 23042 in 5496924793ms cpu (0ms real) (0 rewrites/second)
result Bool: true
```

As expected, the number of rewrites grows with the size of the word, showing that more FSM states are visited during the checking for acceptance. □



```
mod FSM-MODEL is
    protecting FSM-MM .
    protecting MAUDELING .

    op FSMModel : -> @Model .
    op FSMModel : -> @Model .
    eq FSMModel = FiniteStateMachine {
        < 'fsm : FSM@FSM | label@Label@FSM : "(ab)+c" #
                        alphabet@FSM@FSM : Set {"a" ; "b" ; "c" } #
                        states@FSM@FSM : Set {'one ; 'two ; 'three} #
                        transitions@FSM@FSM : Set {'a ; 'b ; 'c} >

        < 'one : State@FSM | label@Label@FSM : "1" #
                        fsm@State@FSM : 'fsm #
                        kind@State@FSM : START@Kind@FSM #
                        in@State@FSM : Set {'b} #
                        out@State@FSM : Set {'a} >

        < 'two : State@FSM | label@Label@FSM : "2" #
                        fsm@State@FSM : 'fsm #
                        kind@State@FSM : NORMAL@Kind@FSM #
                        in@State@FSM : Set {'a} #
                        out@State@FSM : Set {'b ; 'c} >

        < 'three : State@FSM | label@Label@FSM : "3" #
                        fsm@State@FSM : 'fsm #
                        kind@State@FSM : STOP@Kind@FSM #
                        in@State@FSM : Set {'c} #
                        out@State@FSM : Set {mt} >

        < 'a : Transition@FSM | label@Label@FSM : "a" #
                        fsm@Transition@FSM : 'fsm #
                        src@Transition@FSM : 'one #
                        tgt@Transition@FSM : 'two >

        < 'b : Transition@FSM | label@Label@FSM : "b" #
                        fsm@Transition@FSM : 'fsm #
```

```
Ready.
    empty))) ; FSMModel >> .
rewrites: 68 in 5495204793ms cpu (0ms real) (0 rewrites/second)
result Bool: true

Maude> reduce in FSM-MODEL : << not i@getStart@FSM@FSM .=. states@FSM@FSM -> size() or
    states@FSM@FSM -> at(i@getStart@FSM@FSM) . kind@State@FSM .=.
    START@Kind@FSM ; env(putVar(i@getStart@FSM@FSM <- 3, putVar(self <- 'fsm,
    empty))) ; FSMModel >> .
rewrites: 68 in 5489444793ms cpu (1ms real) (0 rewrites/second)
result Bool: false

Maude> reduce in FSM-MODEL : << i@getStart@FSM@FSM .=. states@FSM@FSM -> size() ; env(
    putVar(i@getStart@FSM@FSM <- 1, putVar(self <- 'fsm, empty))) ; FSMModel >>
    .
rewrites: 31 in 5483660793ms cpu (0ms real) (0 rewrites/second)
result Bool: false

Maude> reduce in FSM-MODEL : << states@FSM@FSM -> size() ; env(putVar(
    i@getStart@FSM@FSM <- 1, putVar(self <- 'fsm, empty))) ; FSMModel >> .
rewrites: 27 in 5479628793ms cpu (0ms real) (0 rewrites/second)
result NzNat: 3

Maude>
```

Figure 9.3 – Integration of KMV in Eclipse.

9.5 The KMV Tool

This Section elaborates on the integration of KMV within the Eclipse Integrated Development Environment, and discusses the current limitations of the tool.

9.5.1 Eclipse Integration

Since Kermeta and Maude (through the Maude Development Tool[6]) are both integrated into Eclipse, we started developing KMV also in Eclipse to ensure a better integration.

Eclipse can easily be extended with different features. Two possibilities are currently under study. The first would consist of extending the Kermeta Perspective with new features for translating Kermeta code into Maude: this approach has the advantage to let designers and modellers work in their natural environment; but it requires to evolve along with Kermeta, forcing to constantly update it to follow the subsequent versions of Kermeta. The second one would consist of developing a brand new Eclipse Perspective with dedicated plugins, which presents the opposite advantages and drawbacks than the previous one. Note that for the moment, only translation from Kermeta to Maude works (cf. next Section).

9.5.2 Limitations

The first task required to enable a better experience with KMV is to complete the automatic translator from Kermeta into the Maude syntax, as defined in Section 9.3.2: this translation is currently performed with Kermeta itself, since a metamodel for Kermeta and for Maude (Rivera, Durán, and Vallecillo 2008) already exist in Ecore.

[6]Maude Development Tool (MDT) Website: http://mdt.sourceforge.net/

However, transparently handling such a translation still needs improvements: for now, the translator generates an intermediate file in the same directory than the original Kermeta file, which imposes that the Kermeta specification is included in a single file.

The second point is the specification of properties. Currently, the specification has to be performed directly on the Maude representation, which somehow breaks the principle of "staying at the modelling level": the designer has to understand partially understand how the translation works in order to correctly specify properties. Invariants can easily be expressed in Kermeta under the form of *contracts invariants* based on OCL expressions. Since an evaluator for OCL expressions is embedded into KMV through mOdCL, invariants can easily be translated into Maude. However, some Maude-specific constructions need an equivalent representation in Kermeta to be exploited by designers. Similarly, LTL expressions have to be defined at a higher level to provide designers a way to express their safety properties appropriately. We are currently investigating how to provide a general framework for this purpose.

We are also currently studying how it is possible to programmatically collect Maude traces in order to relay them at the level of the Kermeta code (still under the single-file assumption). This will help the user to better locate specification errors instead of inspecting the Maude traces.

10 Conclusion

This Thesis tries to bring closer two usually separated worlds: modelling and formal verification. Mde should become a full engineering discipline: beyond simply providing model browsers and editors, the Mde community should work on tackling all aspects of the software lifecycle, from simulation and debugging, to software quality evaluation and formal verification.

In this concluding Chapter, we first summarise the contributions of the Thesis, in the perspective of the challenges we enunciated in the Introduction. Then, we explore the limits of our work, as well as possible perspectives and future work.

10.1 Contributions Summary

We first argued that it was necessary, as a community effort, to guide transformation designers in the process of formal verification. This is necessary because knowledge about transformation techniques and tools and about formal analysis in general is usually difficult to acquire, but is becoming nowadays necessary since Mde reaches maturity and tends to apply on more application domains, among which some are critical.

To address this point, we proposed a methodology consisting of a mapping between *transformation intents* and *characteristic properties*. The notion of intent captures what the transformation is achieving, beyond the form it adopts. This semantic shift allows to better identify property classes that characterise all transformations fitting under the same intent. We precisely defined such mappings for five different intents among the most represented and most used in Mde. We concretised some of these intents with a specific transformation borrowed from a large case study, and studied how the property classes operate in practice.

We also argued that it should be interesting to address formal analysis at the level of transformation frameworks, instead of developing analysis methodologies on a *per*-Dsl basis. By doing that, all Dsls written with a transformation engine would directly benefit from the already implemented analysis, allowing to capitalise the efforts of identification and implementation of new analysis applications. Of course, such a shift requires to accurately identify relevant abstractions for each analysis, which can only be enabled if a full, precise formal semantics of the transformation framework is available. In the Thesis, we have chosen Kermeta as a transformation engine: Kermeta is a popular, object-oriented transformation language, fully aligned with Omg standards, that benefits from a large user community and has an industrial relevance.

To demonstrate the feasibility of this argument, we defined a formalisation of the semantics of Kermeta, from which we derived a mapping into a specific verification domain, Maude. The formalisation is tool-independent: it only uses mathematical tools (i.e. set theory and structural operational semantics) that constitute the common background of engineers and computer scientists. Maude was chosen for two main reasons. On the one hand, it provides a natural executable semantic domain, which makes the semantic gap from the formal semantics specification smaller: set theory is simply mapped to multi-sorted algebraic specifications, whereas structural operational semantics is naturally expressed using Maude's rewriting logic. On the other hand, Maude comes equipped with several powerful analysis capabilities (reachability analysis, model-checking and theorem-proving), which allows to prove several classes of properties for a large set of transformation intents.

10.2 Perspectives & Future Work

Several lines of work can interestingly continue the work initiated in this Thesis. We present them in this Section, grouped by topics that follow the development of the Thesis.

10.2.1 Description Framework

The contribution of Chapter 4 admits several limitations that could be addressed in future work. As an immediate continuation, the framework can be completed with the description of other intents on the basis of the case study we presented, or based on other transformation intents that are not represented in the case study. This may reveal the need for additional intents, or properties leading to an extension of the framework.

We already discussed the relevance of our work to research on the “intent-specific” specification, implementation and analysis of model transformations. In this context, it would be interesting to explore the potential connections with recent work on the formal specification, testing, formal verification of model transformations (Büttner, Egea, Cabot, and Gogolla 2012; Guerra, De Lara, et al. 2013; Selim, Cordy, and Dingel 2012a; Selim, Cordy, and Dingel 2012b; Vallecillo and Gogolla 2012): for example, to what extent can existing techniques be used, or extended, to help developers verify properties associated with a transformation’s intent?

We briefly mentioned the use of time and concurrency for model transformations (in particular for the *Simulation* intent). By following the same steps, it should be possible to explore the characteristic properties of concurrent and real-time transformations, and sketch an equivalent framework specific to these domains. An interesting starting point could be the OMG UML-based standard MARTE (Object Management Group 2011b), for which C. André, Mallet, and de Simone (2007) already explored the notions of time that can cope with MARTE.

On a longer term, yet practical side, the true utility of our notion of intent for industrial model-driven software development still remains to determined: inputs from industrial MDE practitioners might be helpful to conduct systematic experiments on the applicability of the Description Framework.

10.2.2 Formal Semantics

The contribution of Part II providing an interesting basis for the formal specification of Kermeta. Several extensions could pave the way towards a fully specified transformation language if the remaining parts of the language receive better attention: *genericity* and *aspects*, but also adding the *Constraint Language* (CL) and the notion of *model type*.

10.2.2.1 Genericity

This is a well-known specification mechanism, currently available in many programming languages. A natural way to handle this feature would consist in selecting an execution/verification domain that natively provide an equivalent mechanism. This is perfectly acceptable, and in favour of a better execution and verification efficiency. However, following our motivation of providing a tool-/syntax- independent formalisation, this point should be mathematically clarified to enable powerful abstractions for properties of interest not directly linked to genericity. This could constitute an interesting result: defining genericity in the context of object-orientation is known to be a difficult task (Castagna and Xu 2011), but Kermeta offers an interesting combination of object-oriented features: multiple inheritance, overriding, but not overloading. It is interesting to study how this particular combination could lead to a full mathematical definition that overcomes the existing difficulty for genericity notions like in Java or Eiffel.

10.2.2.2 Aspects

From a specification viewpoint, aspects are interesting to cleanly separate orthogonal concerns in a model, or during a transformation execution. Kermeta offers a simplified aspect mechanism that is purely syntactic: extensions to existing classes can be specified within different metamodels, that will be merged together (based on name equality) to produce a full, legal Kermeta metamodel before execution, assuming there are no conflicts. These extensions consist of either structural features (attributes or references, but also operations, which is useful to keep the structural and behavioural definitions of a DSL separated) or contracts. Aspects do not directly influence the execution since models are executed after a legal metamodel is obtained from the base and the possible extensions: the main challenge is rather to adequately capture the type-checking rules to properly discard the conflicting cases.

10.2.2.3 Constraint Language (CL)

A CL is useful for at least two purposes: enabling *structural constraints* that restrict the number of models conforming to a given metamodel; and extending the AL with *contracts*, i.e. pre-/ and post-conditions for operations, and class invariants, that prevent executing operations when violated (in Kermeta, this violation leads to a runtime exception that can be caught for appropriate recovery). This feature seems to be easy, mostly because the CL and the AL share some of their constructions. The notable complication comes from *functional* expressions (e.g., `select(...)` or `iterate(...)`): we discarded them because their evaluation requires a proper specification of genericity. However, as explained before, it suffices to define an executable and analysable implementation that takes advantage of native genericity. In Maude, this is even facilitated by the existence of an implementation of the OMG OCL language (cf. mOdCL by Durán, Gogolla, and Roldán 2011; Durán and Roldán 2011), on which Kermeta's CL is aligned.

10.2.2.4 Model Type

Originally introduced by J. R. (Steel (2007), model typing was thought as the natural extension of the notion of type to models (J. Steel and Jézéquel 2007). A type is a classification of values sharing strong characteristics: the operations applicable on the value, the way they are internally stored and represented (i.e. their concrete syntax), and of course their meaning. The usual, so-called "primitive" types (booleans, numerals, characters and strings, dates, and so on) have been extended into more complex datastructures in object-oriented programming through classes: a class basically glues together several of these primitive types, and provides some operations (or methods) applicable on their instances. To reach the next level, the idea would consist of gluing classes together in a stronger way than they are in object-orientation, which is the purpose of (meta-)models: naturally then comes the idea of *(meta-)model operations* applicable to metamodel instances, i.e. models. Unfortunately, the way metamodelling is currently conceived (and so is the formalisation presented in Section 6) prevents to define the appropriate machinery to think about all we know about operations, i.e. subtyping, polymorphism, etc. at the metamodel level.

Recently, Guy (2013) studied this question and proposed to replace the conformance relation by a different relation binding models to one (or several) interface(s) that exposes the model elements and associated model transformations: this interface is the *model type*. Furthermore, model types enjoy subtyping and inheritance relations that, depending on the axioms these relations satisfy, constitute a *typing system* for *model orientation*.

It should be possible to extend and adapt the Structural Language formalised in Chapter 6 for specifying a family of typing systems that a user can freely select depending on the needs of certain (meta-)models. Similarly, the Action Language of Chapter 7 can also be adapted to handle model type transformations: since only the structure of model interfaces are different, but model type transformations work with the same set of statements, it becomes possible to reuse KMV to propose formal analysis of such such metamodels.

10.2.2.5 Towards a Fully Formalised Kermeta

Software systems are required to be certified in critical application domains like avionics, aerospace or even automotive. Since the Kermeta language is ultimately compiled into Java[1], reaching an acceptable certification level requires to look into this compilation scheme to ensure its correctness (more precisely, to establish a refinement relation between the Kermeta specification and the Java Virtual Machine execution). This is a great deal of effort that is worth it only if Kermeta becomes involved in such application domains, but this task is feasible: Börger and Stärk (2003) already provided a formal proof that Java specifications behave correctly regarding their execution on the Virtual Machine; it remains to conduct a similar proof between Kermeta and Java specifications.

10.2.3 KMV

Our technical contribution in Part III resulted in KMV, a prototype tool that demonstrates the feasibility of our proposal. However, this tool is not mature enough at this stage, and several improvements are required to make it easier to use. Furthermore, plain maturity can only be gained when dealing with concrete case studies: a common issue for software, especially in the domain of formal analysis, is the scalability. We discuss possible lines of improvements for KMV and features that we are currently working on.

Eclipse Integration. The automatic translation from Kermeta to Maude is necessary for transformation designers to transparently use our tool. We are currently working on the following aspects:

- A pre-processing phase that identifies "wrong" Kermeta transformations, i.e. transformations using constructions that are not currently handled. This would let the choice for the designer to express the transformation differently (for example, by using loops instead of functionals for traversing the structure of models);
- A transformation from the metamodel of Kermeta into the metamodel of Maude (Rivera, Durán, and Vallecillo 2008), from which a pretty-printer already exists to feed Maude. Although this step would be convenient (and can be handled by Kermeta, since Maude's metamodel is based on ECore), it complicates the backward traceability necessary for helping the designer to identify the locations of its errors.

A full integration into Eclipse would allow transformation designer to use KMV directly from their natural transformation framework. Such an integration is expected in the near future.

Tradeoff between Backward Traceability & Performance. Tracing analysis results back to the original program, or in the context of MDE, model, has always represented a crucial challenge in the computer-aided verification field. Generally, one has to deal with a tradeoff between a translation that maximise the performance of the target verification domain tool, and that minimise the effort to trace results back to the original artefacts. Another solution would let the designers inspect Maude traces themselves (as it is often the case when the semantic gap between the analysed language and the analysing tool is too high, for example for model-checking with SPIN).

Currently, we designed KMV in such a way that backward traceability is facilitated: we use one rule for each statement, and a rule firing is precisely determined by the statement at hand. Replaying the scenarios that

[1] The first versions of Kermeta were compiled into Java. By the time we finished our Thesis, a new version was released (the so-called *Kermeta 2*) that is now compiled into Scala, which provides a better support for genericity and functionals. These features were planned to be integrated into Java 8, whose public release is scheduled for March 2014. Would then be Kermeta reimplemented in Java?

violate a property is therefore easy, although one has to deal with the different operation calls. However, this scheme may be insufficient when manipulating very big models, because the resulting actions can become heavy to handle. We are currently investigating two research paths for improving the model-checking of Kermeta transformations:

1. JavaFan (Farzan et al. 2004) is an analysis tool for multithreaded Java programs that present acceptable performance: we are currently studying the specificities of the Maude specification in this tool, since Java possess a core object-oriented language very close to Kermeta. This will probably require to refund the algebraic specifications for the structural part to obtain efficient data structures for the model-checker.

2. An interesting property of our work is that it should be possible to amalgamate the effect of several statements during a transformation, so that the resulting changes are considered as a black box regarding the properties one is interested to prove. Basically, this defines an *abstract interpretation* for the Kermeta Action Language: the effect of non-relevant statements are abstracted away (and this is what the rewriting equations suggest), and only the ones that have an influence on the property (i.e. either impacting a relevant property or value; or transferring a parameter to an operation that has such an impact) would be retained in the analysis. This opens many theoretical research paths for improving analysis techniques on languages such as Kermeta.

10.2.4 Real-Time Kermeta

Even with the Al subset presented in Part II of this Thesis, Kermeta is a Turing-complete transformation framework: Kermeta can possibly simulate any possible computation. However, it would be more convenient to have at our disposal explicit notions of time, as required by the various time models used for embedded systems (C. André, Mallet, and de Simone 2007). A common solution to overcome this issue consists of giving transformation designers the possibility to build their own notion of time, based on the expressive power of their transformation framework. But this solution comes with several drawbacks: time becomes specific to each Dsl. This solution limits the possibility of exchanging models for which the notion of time differs or worst, hinders their use if they are incompatible. As a consequence, verification techniques, which are already difficult to handle due to their intrinsic complexity, become difficult to handle due to these different time semantics.

Building upon Kmv, it should be interesting to equip Kermeta with Real-Time capabilities in order to provide engineers using Kermeta a built-in explicit capability for handling different notions of time. However, mixing real-time with object-orientation is known to be difficult from a semantic viewpoint, and consequently, from a verification viewpoint (Selic, Gullekson, and Ward 1995). Interestingly, our work can be a good starting point for extending Kermeta and enabling formal verification using Maude, since a real-time extension for Maude already exists (Ölveczky and Meseguer 2007).

This idea is directly inspired from other projects and contributions. In Mde, Multi-Paradigm Modelling (Mosterman and Vangheluwe 2004) (also known as *heterogeneous modelling*, Hardebolle 2008) promotes the modelling of all parts of a system, at the most appropriate level(s) of abstraction, using the most appropriate formalism(s), to reduce the accidental complexity. We were recently aware of the GeMoC project[2], an open initiative to explore how to enable the support of global software engineering. This project focuses on three design and validation issues for software systems: multiple concerns, handled by multiple Dsls; heterogeneous parts and their integration, supported by various models of computations; and software evolution and openness to react to changes. The project already investigated how to combine Kermeta transformations with various models of concurrency (Combemale, Deantoni, et al. 2013; Combemale, Hardebolle, et al. 2012).

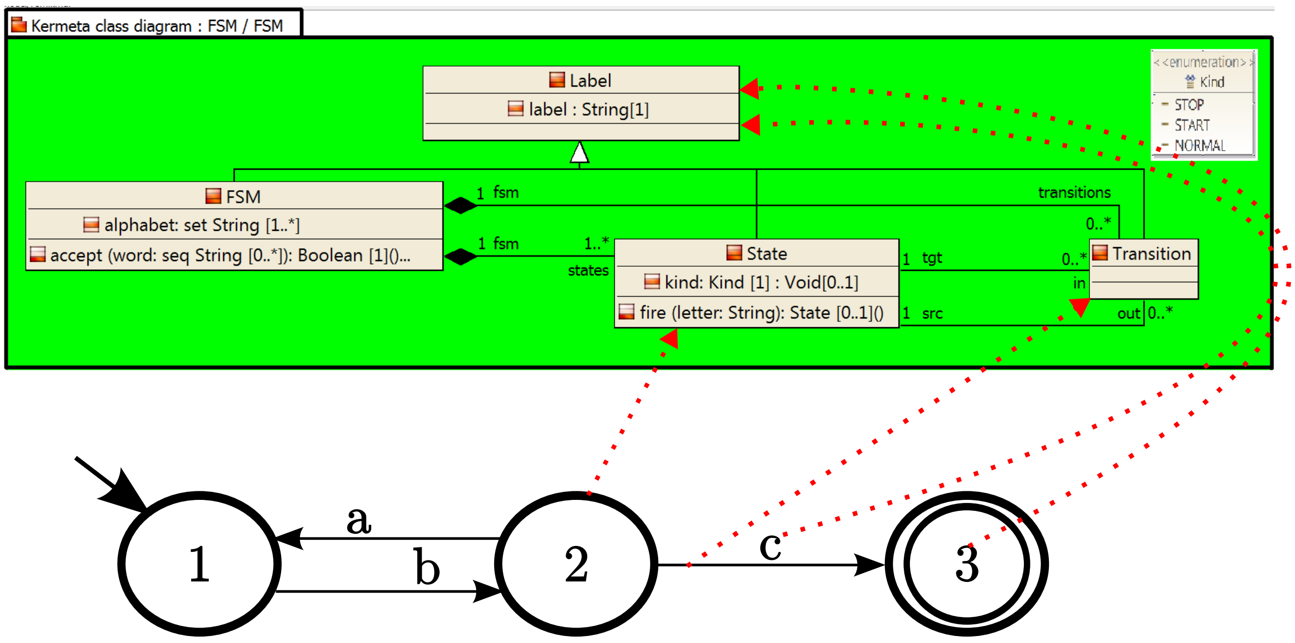
[2]http://gemoc.org

A
The Finite State Machine Example

Throughout this Thesis, we used a toy example, the Finite State Machine (FSM), as a running example. Although relatively simple, this example is however illustrative enough for our purpose: its behavioural semantics is simple, and well-known from computer scientists because FSMs constitute a core formalism for describing computations, but its specification as a Kermeta transformation makes use of the whole core Action Language (cf. Chapter 7), the subset of the Kermeta Action Language we isolated for formalisation. Besides, it is a classical example in Kermeta literature (see e.g. the official Manual of Drey et al. 2009)

The FSM running example showed how to specify Domain-Specific Languages in Kermeta, and illustrated both the formal specification and its Maude implementation. This Appendix aims at summarising these various facets of the FSM: Section A.1 recalls the metamodel and the model we have chosen; Section A.2 presents its full specification using Kermeta's textual representation; Section A.3 summarises and completes its formal representation used in Part II, and finally Section A.4 show the corresponding code in Maude.

A.1 Metamodel and Model



Kermeta class diagram : FSM / FSM
Label
label : String[1]
<<enumeration>>
Kind
STOP
START
NORMAL
FSM
alphabet: set String [1..*]
accept (word: seq String [0..*]): Boolean [1]()...
1 fsm
transitions
0..*
1 fsm
1..*
states
State
kind: Kind [1] : Void[0..1]
fire (letter: String): State [0..1]()
1 tgt
0..*
in
Transition
1 src
out
0..*
1
a
b
2
c
3

A.2 Kermeta Full Textual Definition

```
@mainClass "FSM::FSM"
@mainOperation "accept"

package FSM;
   require kermeta
   using kermeta::standard

   enumeration Kind {NORMAL;START;STOP;}


   class Label{
      attribute label: String
   }

   class FSM inherits Label{
   // FSM assumes there is only one START and one FINAL State
      attribute alphabet: set String [1..*]
      reference states: seq State [1..*] # fsm
      reference transitions: seq Transition [0..*]# fsm

      operation getStart(): State is do
         var i : Integer init 1
         from i := 1
            until i == states.size() or states.at(i).kind == Kind.START
         loop
            i := i+1
         end
         if i == states.size() then
            result := void
         else
            result := self.states.at(i)
         end
      end

      operation getFinal(): State is do
         var i : Integer init 1
         from i := 1
            until i == states.size() or states.at(i).kind == Kind.STOP
         loop
            i := i+1
         end
         if i == states.size() then
            result := void
         else
            result := self.states.at(i)
         end
      end

      operation accept(word: seq String [0..*]) : Boolean is do
         var current: State init self.getStart()
         var final  : State init self.getFinal()
         var toEval : seq String[0..*] init word
         var isNull : Boolean init false

         from var i : Integer init 1
            until i == toEval.size() or isNull
         loop
            current := current.fire(toEval.at(i))
            if(current.isVoid) then
               isNull := true
           end
            i := i+1
         end
         result := (current == final)
      end
   }

   class State inherits Label{
      attribute kind: Kind
```

```
    reference fsm: FSM # states
    reference in:  Transition [0..*] # tgt
    reference out: Transition [0..*] # src

    operation fire(letter: String): State [0..1] is do
       var trans: seq Transition [0..*] init self.out.asSequence()

       if(trans.isVoid) then
          // no output transitions: return what is suitable for no state (card = 0)
          result := void
       else
          var current: Transition init trans.at(0)
          // head only reads the head, do not modify the sequence

          from var i : Integer init 0
             until i == trans.size() or trans.at(i).label == letter//(current.isVoid) or (current.label ==
                 letter)
          loop
             i := i+1
          end
          if(current.isVoid) then
             result:= void
          else
             result:= current.tgt
          end
       end
    end
 }

 class Transition inherits Label{
    reference fsm: FSM # transitions
    reference tgt: State [1..1] # in
    reference src: State [1..1] # out
 }
```

A.3 Mathematical Representation

A.3.1 The FSM Metamodel $\mathsf{MM}_{\mathsf{FSM}}$

Let us call $\mathsf{MM}_{\mathsf{FSM}} = (p, c, e, \mathrm{prop}, o) \in \mathcal{M}$ the mathematical representation of the metamodel depicted in Section A.1. Because the mathematical framework uses partial functions, we have to specify the functions' domains first: $\mathsf{Dom}\,(p) = \mathsf{Dom}\,(c) = \mathsf{Dom}\,(\mathrm{prop}) = \mathsf{Dom}\,(o) = \{\mathsf{FSM}\}$ (only one package); $\mathsf{Dom}\,(c(\mathsf{FSM})) = \mathsf{Dom}\,(\mathrm{prop}(\mathsf{FSM})) = \mathsf{Dom}\,(o(\mathsf{FSM})) = \mathsf{C}_{\mathsf{FSM}}$ with $\mathsf{C}_{\mathsf{FSM}} = \{\mathsf{Label}, \mathsf{FSM}, \mathsf{State}, \mathsf{Transition}\}$, i.e. the classes contained inside the FSM package; and $\mathsf{Dom}\,(e(\mathsf{FSM})) = \mathsf{E}_{\mathsf{FSM}}$ with $\mathsf{E}_{\mathsf{FSM}} = \{\mathsf{Kind}\}$. The domains for prop and o are build by gathering all properties and operations respectively: $\mathsf{Dom}\,(\mathrm{prop}(\mathsf{FSM})(\mathsf{Label})) = \{\mathsf{label}\}$ and $\mathsf{Dom}\,(o(\mathsf{FSM})(\mathsf{Label})) = \varnothing$; $\mathsf{Dom}\,(\mathrm{prop}(\mathsf{FSM})(\mathsf{FSM})) = \{\mathsf{alphabet}, \mathsf{states}, \mathsf{transitions}\}$ and $\mathsf{Dom}\,(o(\mathsf{FSM})(\mathsf{FSM})) = \{\mathsf{getStart}, \mathsf{getFinal}, \mathsf{accept}\}$; $\mathsf{Dom}\,(\mathrm{prop}(\mathsf{FSM})(\mathsf{State})) = \{\mathsf{fsm}, \mathsf{kind}, \mathsf{in}, \mathsf{out}\}$ and $\mathsf{Dom}\,(o(\mathsf{FSM})(\mathsf{State})) = \{\mathsf{fire}\}$; and finally $\mathsf{Dom}\,(\mathrm{prop}(\mathsf{FSM})\ (\mathsf{Transition})) = \{\mathsf{fsm}, \mathsf{src}, \mathsf{tgt}\}$ and $\mathsf{Dom}\,(o(\mathsf{FSM})(\mathsf{Transition})) = \varnothing$.

We then have to define all the component functions of $\mathsf{MM}_{\mathsf{FSM}}$. Let us start with the simplest ones, namely the package, class and enumeration functions.

$$\begin{array}{rcl}
p(\mathsf{FSM}) & = & (\varnothing, \mathsf{C}_{\mathsf{FSM}}, \mathsf{E}_{\mathsf{FSM}}) \\
e(\mathsf{FSM})(\mathsf{Kind}) & = & \langle\!\langle \mathsf{NORMAL}, \mathsf{STOP}, \mathsf{START} \rangle\!\rangle \\
c(\mathsf{FSM})(\mathsf{Label}) & = & (\top, \varnothing) \\
c(\mathsf{FSM})(\mathsf{FSM}) & = & (\bot, \{\mathsf{Label}\}) \\
c(\mathsf{FSM})(\mathsf{State}) & = & (\bot, \{\mathsf{Label}\}) \\
c(\mathsf{FSM})(\mathsf{Transition}) & = & (\bot, \{\mathsf{Label}\})
\end{array}$$

Let us now proceed with the property and operation functions, class by class.

$$\mathrm{prop}(\mathsf{FSM})(\mathsf{Label})(\mathsf{label}) \quad = \quad (\top, \bot, (1, 1, \bot, \mathsf{String}), \bot)$$

$$\begin{aligned}
\mathsf{prop}(\mathsf{FSM})(\mathsf{FSM})(\mathsf{alphabet}) &= (\top, \bot, (1, \star, \mathsf{Set}, \mathsf{String}), \bot)\\
\mathsf{prop}(\mathsf{FSM})(\mathsf{FSM})(\mathsf{states}) &= (\top, \bot, (1, \star, \mathsf{OSet}, \mathsf{State}), \mathsf{fsm})\\
\mathsf{prop}(\mathsf{FSM})(\mathsf{FSM})(\mathsf{transitions}) &= (\top, \bot, (0, \star, \mathsf{OSet}, \mathsf{Transition}), \mathsf{fsm})\\
o(\mathsf{FSM})(\mathsf{FSM})(\mathsf{accept}) &= (\bot, \bot, \langle\!\langle(\mathsf{word}, (0, \star, \mathsf{List}, \mathsf{String}))\rangle\!\rangle, (1, 1, \bot, \mathsf{Boolean}), b_{\mathsf{accept}})\\
o(\mathsf{FSM})(\mathsf{FSM})(\mathsf{getStart}) &= (\bot, \bot, \langle\!\langle\rangle\!\rangle, (1, 1, \bot, \mathsf{State}), b_{\mathsf{getStart}})\\
o(\mathsf{FSM})(\mathsf{FSM})(\mathsf{getFinal}) &= (\bot, \bot, \langle\!\langle\rangle\!\rangle, (1, 1, \bot, \mathsf{State}), b_{\mathsf{getFinal}})
\end{aligned}$$

$$\begin{aligned}
\mathsf{prop}(\mathsf{FSM})(\mathsf{State})(\mathsf{kind}) &= (\top, \bot, (1, 1, \bot, \mathsf{Kind}), \bot)\\
\mathsf{prop}(\mathsf{FSM})(\mathsf{State})(\mathsf{fsm}) &= (\bot, \bot, (1, 1, \bot, \mathsf{FSM}), \mathsf{states})\\
\mathsf{prop}(\mathsf{FSM})(\mathsf{State})(\mathsf{in}) &= (\bot, \bot, (0, \star, \mathsf{List}, \mathsf{Transition}), \mathsf{tgt})\\
\mathsf{prop}(\mathsf{FSM})(\mathsf{State})(\mathsf{out}) &= (\bot, \bot, (0, \star, \mathsf{List}, \mathsf{Transition}), \mathsf{src})\\
o(\mathsf{FSM})(\mathsf{State})(\mathsf{fire}) &= (\bot, \bot, \langle\!\langle(\mathsf{letter}, (1, 1, \bot, \mathsf{String}))\rangle\!\rangle, (0, 1, \bot, \mathsf{State}), b_{\mathsf{fire}})
\end{aligned}$$

$$\begin{aligned}
\mathsf{prop}(\mathsf{FSM})(\mathsf{Transition})(\mathsf{fsm}) &= (\bot, \bot, (1, 1, \bot, \mathsf{FSM}), \mathsf{transitions})\\
\mathsf{prop}(\mathsf{FSM})(\mathsf{Transition})(\mathsf{src}) &= (\bot, \bot, (1, 1, \bot, \mathsf{State}), \mathsf{out})\\
\mathsf{prop}(\mathsf{FSM})(\mathsf{Transition})(\mathsf{tgt}) &= (\bot, \bot, (1, 1, \bot, \mathsf{State}), \mathsf{in})
\end{aligned}$$

A.3.2 The FSM Model $\mathsf{M}_{\mathsf{abc}}$

Let us now call $\mathsf{M}_{\mathsf{abc}} \in \mathbb{M}$ the representation of the model depicted in Section A.1. Using the names as object identifiers, we have $\mathsf{Dom}\,(\mathsf{M}_{\mathsf{abc}}) = \{abc, 1, 2, 3, a, b, c\}$. We obviously have $type(abc) = (\mathsf{FSM}, \mathsf{FSM})$, $type(1) = type(2) = type(3) = (\mathsf{FSM}, \mathsf{State})$ and $type(a) = type(b) = type(c) = (\mathsf{FSM}, \mathsf{Transition})$. We only describe the state of the necessary instances for the conformance proof.

$$\begin{aligned}
\sigma^{abc}(\mathsf{label}) &= \text{"}(\mathsf{ab}) + \mathsf{c}\text{"}\\
\sigma^{abc}(\mathsf{alphabet}) &= \{\text{"a"}, \text{"b"}, \text{"c"}\}\\
\sigma^{abc}(\mathsf{states}) &= \langle\!\langle 1, 2, 3\rangle\!\rangle\\
\sigma^{abc}(\mathsf{transitions}) &= \langle\!\langle a, b, c\rangle\!\rangle
\end{aligned}
\qquad
\begin{aligned}
\sigma^{1}(\mathsf{label}) &= \text{"1"}\\
\sigma^{1}(\mathsf{kind}) &= \mathsf{START}\\
\sigma^{1}(\mathsf{in}) &= \langle\!\langle b\rangle\!\rangle\\
\sigma^{1}(\mathsf{out}) &= \langle\!\langle a\rangle\!\rangle\\
\sigma^{1}(\mathsf{fsm}) &= abc
\end{aligned}$$

$$\begin{aligned}
\sigma^{a}(\mathsf{label}) &= \text{"a"}\\
\sigma^{a}(\mathsf{src}) &= 1\\
\sigma^{a}(\mathsf{tgt}) &= 2\\
\sigma^{a}(\mathsf{fsm}) &= abc
\end{aligned}
\qquad
\begin{aligned}
\sigma^{b}(\mathsf{label}) &= \text{"b"}\\
\sigma^{b}(\mathsf{src}) &= 2\\
\sigma^{b}(\mathsf{tgt}) &= 1\\
\sigma^{b}(\mathsf{fsm}) &= abc
\end{aligned}$$

A.3.3 Does $\mathsf{M}_{\mathsf{abc}}$ conform to $\mathsf{MM}_{\mathsf{FSM}}$?

We now prove the conformance, i.e. $\mathsf{M}_{\mathsf{abc}} \blacktriangleright \mathsf{MM}_{\mathsf{FSM}}$. The conformance predicate holds if certain conditions hold on all objects of the model. Since it is a repetitive task, we only demonstrate for one object of each type, letting the reader infer what is missing for the rest of the objects.

The first condition is easy to check from the definition of $\mathsf{M}_{\mathsf{abc}}$ itself: each object has a type that appears in $\mathsf{MM}_{\mathsf{FSM}}$, and each accessible property of each object possess a value.

It then remains to prove that each accessible property of each object has a valid value regarding the property declared type, and so do opposites. We only select one object per type, i.e. the object already presented when describing $\mathsf{M}_{\mathsf{abc}}$.

FSM An FSM instance has four accessible properties: $\mathsf{Dom}\,(\pi(\mathsf{FSM})(\mathsf{FSM})) = \{\mathsf{label}, \mathsf{alphabet}, \mathsf{states}, \mathsf{transitions}\}$. We go through all of them, since they all have different types:

label From the previous definitions, we have $\sigma^{abc}(\mathsf{label}) = \text{"}(\mathsf{ab}) + \mathsf{c}\text{"}$ and $\mathsf{prop}(\mathsf{FSM})(\mathsf{Label})(\mathsf{label}) = (\top, \bot, (1, 1, \bot, \mathsf{String}), \bot)$.

- Since type("$\mathsf{(ab) + c}$") $= (\bot, \mathsf{String})$, it implies that type("1") $\preccurlyeq (\bot, \mathsf{String})$;
- Since $\mid$ "$\mathsf{(ab) + c}$" $\mid = 1$, it implies that $1 \leqslant \mid$ "$\mathsf{(ab) + c}$" $\mid \leqslant 1$;
- label does not declare an opposite so the last check is irrelevant.

alphabet We have $\sigma^{abc}(\mathsf{alphabet}) = \{\text{"a"}, \text{"b"}, \text{"c"}\}$ with $\mathsf{prop}(\mathsf{FSM})(\mathsf{FSM})(\mathsf{alphabet}) = (\top, \bot, (1, \star, \mathsf{Set}, \mathsf{String}), \bot)$.

- Since type($\{\text{"a"}, \text{"b"}, \text{"c"}\}$) $= (\mathsf{Set}, \mathsf{String})$, it implies that type($\{\text{"a"}, \text{"b"}, \text{"c"}\}$) $\preccurlyeq (\mathsf{Set}, \mathsf{State})$;
- Since $\mid \{\text{"a"}, \text{"b"}, \text{"c"}\} \mid = 3$, it implies that $1 \leqslant \mid \{\text{"a"}, \text{"b"}, \text{"c"}\} \mid \leqslant \star$;
- $\mathsf{alphabet}$ does not declare an opposite so the last check is irrelevant.

states We have $\sigma^{abc}(\mathsf{states}) = \langle\!\langle 1, 2, 3 \rangle\!\rangle$ and $\mathsf{prop}(\mathsf{FSM})(\mathsf{FSM})(\mathsf{states}) = (\bot, \bot, (1, \star, \mathsf{OSet}, \mathsf{State}), \mathsf{fsm})$.

- Since type($\langle\!\langle 1, 2, 3 \rangle\!\rangle$) $= (\mathsf{OSet}, (\mathsf{FSM}, \mathsf{State}))$, it implies that type($\langle\!\langle 1, 2, 3 \rangle\!\rangle$) $\preccurlyeq (\mathsf{OSet}, \mathsf{String})$;
- Since $\mid \langle\!\langle 1, 2, 3 \rangle\!\rangle \mid = 3$, it implies that $1 \leqslant \mid \langle\!\langle 1, 2, 3 \rangle\!\rangle \mid \leqslant \star$;
- We have $\sigma^1(\mathsf{fsm}) = abc$ and $\mathsf{prop}(\mathsf{FSM})(\mathsf{State})(\mathsf{fsm}) = (\bot, \bot, (1, 1, \bot, \mathsf{FSM}), \mathsf{states})$ and effectively, $abc \in \sigma^1(\mathsf{fsm})$ (same holds for 2's and 3's values).

transitions We have $\sigma^{abc}(\mathsf{transitions}) = \langle\!\langle a, b, c \rangle\!\rangle$ and $\mathsf{prop}(\mathsf{FSM})(\mathsf{FSM})(\mathsf{transitions}) = (\bot, \bot, (0, \star, \mathsf{OSet}, \mathsf{Transition}), \mathsf{fsm})$.

- Since type($\langle\!\langle a, b, c \rangle\!\rangle$) $= (\mathsf{OSet}, (\mathsf{FSM}, \mathsf{State}))$, it implies that type($\langle\!\langle a, b, c \rangle\!\rangle$) $\preccurlyeq (\mathsf{OSet}, \mathsf{String})$;
- Since $\mid \langle\!\langle a, b, c \rangle\!\rangle \mid = 3$, it implies that $1 \leqslant \mid \langle\!\langle a, b, c \rangle\!\rangle \mid \leqslant \star$;
- We have $\sigma^a(\mathsf{fsm}) = abc$ and $\mathsf{prop}(\mathsf{FSM})(\mathsf{Transition})(\mathsf{fsm}) = (\bot, \bot, (1, 1, \bot, \mathsf{FSM}), \mathsf{transitions})$ and effectively, $abc \in \sigma^a(\mathsf{fsm})$ (same holds for a's and b's values).

State A State instance has five accessible properties: $\mathsf{Dom}\,(\pi(\mathsf{FSM})(\mathsf{State})) = \{\mathsf{label}, \mathsf{kind}, \mathsf{in}, \mathsf{out}, \mathsf{fsm}\}$. Properties in and out only differ for their opposite, therefore we only consider in.

label We have $\sigma^1(\mathsf{label}) =$ "1" and $\mathsf{prop}(\mathsf{FSM})(\mathsf{Label})(\mathsf{label}) = (\top, \bot, (1, 1, \bot, \mathsf{String}), \bot)$.

- Since type("1") $= (\bot, \mathsf{String})$, it implies that type("1") $\preccurlyeq (\bot, \mathsf{String})$;
- Since $\mid$ "1" $\mid = 1$, it implies that $1 \leqslant \mid$ "1" $\mid \leqslant 1$;
- label does not declare an opposite so the last check is irrelevant.

kind We have $\sigma^1(\mathsf{kind}) = \mathsf{START}$ and $\mathsf{prop}(\mathsf{FSM})(\mathsf{State})(\mathsf{kind}) = (\top, \bot, (1, 1, \bot, \mathsf{Kind}), \bot)$.

- Since type(START) $= (\bot, \mathsf{Kind})$, it implies that type(START) $\preccurlyeq (\bot, \mathsf{Kind})$;
- Since $\mid \mathsf{START} \mid = 1$, it implies that $1 \leqslant \mid \mathsf{START} \mid \leqslant 1$;
- kind does not declare an opposite so the last check is irrelevant.

in We have $\sigma^1(\mathsf{in}) = \langle\!\langle b \rangle\!\rangle$ and $\mathsf{prop}(\mathsf{FSM})(\mathsf{State})(\mathsf{in}) = (\bot, \bot, (0, \star, \mathsf{List}, \mathsf{Transition}), \mathsf{tgt})$.

- Since type($\langle\!\langle b \rangle\!\rangle$) $= (\mathsf{OSet}, \mathsf{Transition})$, it implies that type($\langle\!\langle b \rangle\!\rangle$) $\preccurlyeq (\mathsf{OSet}, \mathsf{Transition})$;
- Since $\mid \langle\!\langle b \rangle\!\rangle \mid = 1$, it implies that $0 \leqslant \mid \langle\!\langle b \rangle\!\rangle \mid \leqslant \star$;
- We have $\sigma^b(\mathsf{tgt}) = 1$ and $\mathsf{prop}(\mathsf{FSM})(\mathsf{Transition})(\mathsf{tgt}) = (\bot, \bot, (1, 1, \bot, \mathsf{State}), \mathsf{in})$ and effectively, $1 \in \sigma^b(\mathsf{tgt})$.

fsm We have $\sigma^1(\mathsf{fsm}) = abc$ and $\mathsf{prop}(\mathsf{FSM})(\mathsf{Transition})(\mathsf{fsm}) = (\bot, \bot, (1, 1, \bot, \mathsf{FSM}), \mathsf{transitions})$.

- Since type(abc) $= (\bot, (\mathsf{FSM}, \mathsf{FSM}))$, it implies that type($abc$) $\preccurlyeq (\bot, (\mathsf{FSM}, \mathsf{FSM}))$;
- Since $\mid abc \mid = 1$, it implies that $\leqslant \mid abc \mid \leqslant 1$;
- We already saw that $\sigma^{abc}(\mathsf{transitions}) = \langle\!\langle a, b, c \rangle\!\rangle$ and $\mathsf{prop}(\mathsf{FSM})(\mathsf{FSM})(\mathsf{transitions}) = (\bot, \bot, (0, \star, \mathsf{OSet}, \mathsf{Transition}), \mathsf{fsm})$ and effectively, $b \in \sigma^{abc}(\mathsf{transitions})$.

Transition A $\mathsf{Transition}$ has four accessible properties: $\mathsf{Dom}\,(\pi(\mathsf{FSM})(\mathsf{Transition})) = \{\mathsf{label}, \mathsf{src}, \mathsf{tgt}, \mathsf{fsm}\}$. Properties src and tgt only differ for their opposite, therefore we only consider tgt (as the opposite of property in for type State).

label We have $\sigma^b(\mathsf{label}) = \text{"b"}$ and $\mathsf{prop}(\mathsf{FSM})(\mathsf{Label})(\mathsf{label}) = (\top, \bot, (1, 1, \bot, \mathsf{String}), \bot)$.

- Since $\mathsf{type}(\text{"b"}) = (\bot, \mathsf{String})$, it implies that $\mathsf{type}(\text{"b"}) \preccurlyeq (\bot, \mathsf{String})$;
- Since $\mid \text{"b"} \mid = 1$, it implies that $1 \leqslant \mid \text{"b"} \mid \leqslant 1$;
- label does not declare an opposite so the last check is irrelevant.

tgt We have $\sigma^b(\mathsf{tgt}) = 1$ and $\mathsf{prop}(\mathsf{FSM})(\mathsf{Transition})(\mathsf{tgt}) = (\bot, \bot, (1, 1, \bot, \mathsf{State}), \mathsf{in})$.

- Since $\mathsf{type}(1) = (\bot, (\mathsf{FSM}, \mathsf{State}))$, it implies that $\mathsf{type}(1) \preccurlyeq (\bot, (\mathsf{FSM}, \mathsf{State}))$;
- Since $\mid 1 \mid = 1$, it implies that $\leqslant \mid 1 \mid \leqslant 1$;
- We already saw that $\sigma^1(\mathsf{in}) = \langle\!\langle b \rangle\!\rangle$ with $\mathsf{prop}(\mathsf{FSM})(\mathsf{State})(\mathsf{in}) = (\bot, \bot, (0, \star, \mathsf{List}, \mathsf{Transition}), \mathsf{tgt})$ and effectively, $b \in \sigma^1(\mathsf{in})$.

A.4 Maude Representation

```
mod FSM-MM is
   protecting MAUDELING .

   *** Metamodel & Package ***

   --- Metamodel declaration
   op FiniteStateMachine : -> @Metamodel .
   eq name(FiniteStateMachine) = "FiniteStateMachine" .
   eq packages (FiniteStateMachine) = FSM .

   --- Package declaration
   op FSM : -> @Package .
   eq name(FSM) = "FSM" .
   eq metamodel(FSM) = FiniteStateMachine .
   eq superPackage(FSM) = null .
   eq subPackages(FSM) = nil .
   eq classes(FSM) = __(FSM@FSM, Label@FSM, State@FSM, Transition@FSM) .

   *** Enumerations ***
   --- Kind

   sort Kind@FSM .
   subsorts Kind@FSM < @EnumerationInstance .
   op Kind@FSM : -> @Enumeration .
   op NORMAL@Kind@FSM : -> Kind@FSM .
   op START@Kind@FSM : -> Kind@FSM .
   op STOP@Kind@FSM : -> Kind@FSM .
   eq metaAux( X:Kind@FSM ) = Kind@FSM .
   eq name( Kind@FSM ) = "Kind" .
   eq package( Kind@FSM ) = FSM .
   eq defaultValue( Kind@FSM ) = NORMAL@Kind@FSM .
   eq literals( Kind@FSM ) = __( NORMAL@Kind@FSM , START@Kind@FSM , STOP@Kind@FSM ) .



   *** Classes ***

   --- Label

   sort Label@FSM .
   subsort Label@FSM < @Class .
   op Label@FSM : -> Label@FSM .
   eq name(Label@FSM) = "Label" .
   eq isAbstract(Label@FSM) = true .
   eq package(Label@FSM) = FSM .
   eq superTypes (Label@FSM) = nil .
   eq references (Label@FSM) = nil .
   eq attributes (Label@FSM) = label@Label@FSM .

   op label@Label@FSM  : -> @Attribute .
   eq name (label@Label@FSM) = "label" .
```

```
  eq type (label@Label@FSM) = @String .
  eq lowerBound (label@Label@FSM) = 1 .
  eq upperBound (label@Label@FSM) = 1 .
  eq containingClass (label@Label@FSM) = Label@FSM .
  eq isOrdered (label@Label@FSM) = true .
  eq isUnique (label@Label@FSM) = true .
  eq isId (label@Label@FSM) = true .

  --- FSM

  sort FSM@FSM .
  subsort FSM@FSM < Label@FSM .
  op FSM@FSM : -> FSM@FSM .
  eq name(FSM@FSM) = "FSM" .
  eq isAbstract(FSM@FSM) = false .
  eq package(FSM@FSM) = FSM .
  eq superTypes (FSM@FSM) = Label@FSM .
  eq references (FSM@FSM) = __(states@FSM@FSM, transitions@FSM@FSM) .
  eq attributes (FSM@FSM) = alphabet@FSM@FSM .

  op alphabet@FSM@FSM  : -> @Attribute .
  eq name (alphabet@FSM@FSM) = "alphabet" .
  eq type (alphabet@FSM@FSM) = @String .
  eq lowerBound (alphabet@FSM@FSM) = 1 .
  eq upperBound (alphabet@FSM@FSM) = * .
  eq containingClass (alphabet@FSM@FSM) = FSM@FSM .
  eq isOrdered (alphabet@FSM@FSM) = false .
  eq isUnique (alphabet@FSM@FSM) = true .
  eq isId (alphabet@FSM@FSM) = false .

  op states@FSM@FSM : -> @Reference .
  eq name (states@FSM@FSM) = "states" .
  eq opposite (states@FSM@FSM) = fsm@State@FSM .
  eq type (states@FSM@FSM) = State@FSM .
  eq lowerBound (states@FSM@FSM) = 1 .
  eq upperBound (states@FSM@FSM) = * .
  eq containingClass (states@FSM@FSM) = FSM@FSM .
  eq isOrdered (states@FSM@FSM) = false .
  eq isUnique (states@FSM@FSM) = true .
  eq isContainment (states@FSM@FSM) = true .

  op transitions@FSM@FSM : -> @Reference .
  eq name (transitions@FSM@FSM) = "transitions" .
  eq opposite (transitions@FSM@FSM) = fsm@Transition@FSM .
  eq type (transitions@FSM@FSM) = Transition@FSM .
  eq lowerBound (transitions@FSM@FSM) = 0 .
  eq upperBound (transitions@FSM@FSM) = * .
  eq containingClass (transitions@FSM@FSM) = FSM@FSM .
  eq isOrdered (transitions@FSM@FSM) = false .
  eq isUnique (transitions@FSM@FSM) = true .
  eq isContainment (transitions@FSM@FSM) = true .

  --- Transition

  sort Transition@FSM .
  subsort Transition@FSM < Label@FSM .
  op Transition@FSM : -> Transition@FSM .
  eq name(Transition@FSM) = "Transition" .
  eq isAbstract(Transition@FSM) = false .
  eq package(Transition@FSM) = FSM .
  eq superTypes (Transition@FSM) = Label@FSM .
  eq references (Transition@FSM) = __(src@Transition@FSM, tgt@Transition@FSM,
                          fsm@Transition@FSM) .
  eq attributes (Transition@FSM) = nil .

  op src@Transition@FSM : -> @Reference .
  eq name (src@Transition@FSM) = "src" .
  eq opposite (src@Transition@FSM) = out@State@FSM .
  eq type (src@Transition@FSM) = State@FSM .
  eq lowerBound (src@Transition@FSM) = 1 .
  eq upperBound (src@Transition@FSM) = 1 .
  eq containingClass (src@Transition@FSM) = Transition@FSM .
  eq isOrdered (src@Transition@FSM) = true .
  eq isUnique (src@Transition@FSM) = true .
  eq isContainment (src@Transition@FSM) = false .

  op tgt@Transition@FSM : -> @Reference .
  eq name (tgt@Transition@FSM) = "tgt" .
  eq opposite (tgt@Transition@FSM) = in@State@FSM .
  eq type (tgt@Transition@FSM) = State@FSM .
  eq lowerBound (tgt@Transition@FSM) = 1 .
  eq upperBound (tgt@Transition@FSM) = 1 .
  eq containingClass (tgt@Transition@FSM) = Transition@FSM .
  eq isOrdered (tgt@Transition@FSM) = true .
  eq isUnique (tgt@Transition@FSM) = true .
  eq isContainment (tgt@Transition@FSM) = false .

  op fsm@Transition@FSM : -> @Reference .
  eq name (fsm@Transition@FSM) = "fsm" .
  eq opposite (fsm@Transition@FSM) = transitions@FSM@FSM .
  eq type (fsm@Transition@FSM) = FSM@FSM .
  eq lowerBound (fsm@Transition@FSM) = 1 .
  eq upperBound (fsm@Transition@FSM) = 1 .
  eq containingClass (fsm@Transition@FSM) = Transition@FSM .
  eq isOrdered (fsm@Transition@FSM) = true .
  eq isUnique (fsm@Transition@FSM) = true .
  eq isContainment (fsm@Transition@FSM) = false .


  --- State

  sort State@FSM .
  subsort State@FSM < Label@FSM .
  op State@FSM : -> State@FSM .
  eq name(State@FSM) = "State" .
  eq isAbstract(State@FSM) = false .
  eq package(State@FSM) = FSM .
  eq superTypes (State@FSM) = Label@FSM .
  eq references (State@FSM) = __(in@State@FSM, out@State@FSM,
                          fsm@State@FSM) .
  eq attributes (State@FSM) = kind@State@FSM .

  op in@State@FSM : -> @Reference .
  eq name (in@State@FSM) = "in" .
  eq opposite (in@State@FSM) = tgt@Transition@FSM .
  eq type (in@State@FSM) = Transition@FSM .
  eq lowerBound (in@State@FSM) = 0 .
  eq upperBound (in@State@FSM) = * .
  eq containingClass (in@State@FSM) = State@FSM .
  eq isOrdered (in@State@FSM) = false .
  eq isUnique (in@State@FSM) = true .
  eq isContainment (in@State@FSM) = false .

  op out@State@FSM : -> @Reference .
  eq name (out@State@FSM) = "out" .
  eq opposite (out@State@FSM) = src@Transition@FSM .
  eq type (out@State@FSM) = Transition@FSM .
  eq lowerBound (out@State@FSM) = 0 .
  eq upperBound (out@State@FSM) = * .
  eq containingClass (out@State@FSM) = State@FSM .
  eq isOrdered (out@State@FSM) = false .
  eq isUnique (out@State@FSM) = true .
  eq isContainment (out@State@FSM) = false .

  op fsm@State@FSM : -> @Reference .
  eq name (fsm@State@FSM) = "fsm" .
  eq opposite (fsm@State@FSM) = states@FSM@FSM .
  eq type (fsm@State@FSM) = FSM@FSM .
  eq lowerBound (fsm@State@FSM) = 1 .
  eq upperBound (fsm@State@FSM) = 1 .
  eq containingClass (fsm@State@FSM) = State@FSM .
  eq isOrdered (fsm@State@FSM) = true .
```

```
   eq isUnique (fsm@State@FSM) = true .
   eq isContainment (fsm@State@FSM) = false .

   op kind@State@FSM : -> @Attribute .
   eq name (kind@State@FSM) = "kind" .
   eq type (kind@State@FSM) = Kind@FSM .
   eq lowerBound (kind@State@FSM) = 1 .
   eq upperBound (kind@State@FSM) = 1 .
   eq containingClass (kind@State@FSM) = State@FSM .
   eq isOrdered (kind@State@FSM) = true .
   eq isUnique (kind@State@FSM) = true .
   eq isId (kind@State@FSM) = false .
endm
```

```
mod FSM-MODEL is
   protecting MAUDELING .
   protecting FSM-MM .

   *** Set{mt} for
   *** Set separator is ;
   *** No references = nil

   op FSMModel : -> @Model .
   eq FSMModel = FiniteStateMachine {
      < 'fsm : FSM@FSM | label@Label@FSM : "(ab)+c" #
                  alphabet@FSM@FSM : Set {"a" ; "b" ; "c" } #
                  states@FSM@FSM : Set {'one ; 'two ; 'three} #
                  transitions@FSM@FSM : Set {'a ; 'b ; 'c} >

      < 'one : State@FSM | label@Label@FSM : "1" #
                  fsm@State@FSM : 'fsm #
                  kind@State@FSM : START@Kind@FSM #
                  in@State@FSM : Set {'b} #
                  out@State@FSM : Set {'a} >

      < 'two : State@FSM | label@Label@FSM : "2" #
                  fsm@State@FSM : 'fsm #
                  kind@State@FSM : NORMAL@Kind@FSM #
                  in@State@FSM : Set {'a} #
                  out@State@FSM : Set {'b ; 'c} >

      < 'three : State@FSM | label@Label@FSM : "3" #
                  fsm@State@FSM : 'fsm #
                  kind@State@FSM : STOP@Kind@FSM #
                  in@State@FSM : Set {'c} #
                  out@State@FSM : Set {mt} >

      < 'a : Transition@FSM | label@Label@FSM : "a" #
                  fsm@Transition@FSM : 'fsm #
                  src@Transition@FSM : 'one #
                  tgt@Transition@FSM : 'two >

      < 'b : Transition@FSM | label@Label@FSM : "b" #
                  fsm@Transition@FSM : 'fsm #
                  src@Transition@FSM : 'two #
                  tgt@Transition@FSM : 'one >

      < 'c : Transition@FSM | label@Label@FSM : "c" #
                  fsm@Transition@FSM : 'fsm #
                  src@Transition@FSM : 'two #
                  tgt@Transition@FSM : 'three >
   } .
endm
```

B

Maude Specification for Kermeta Semantics

```
view @Class from TRIV to METAMODEL-CTORS is
  sort Elt to @Class .
endv

mod KERMETA-STMT-PREP is
   sort @Statement .
endm

mod KERMETA-SL-CTORS is
   pr METAMODEL-PROP .
   pr MGLIST{@NamedElement} * (sort MGList{@NamedElement} to MyList) .

   pr MGMAYBE{@Class} .

   --- Operation & Parameters signature
   sort @Operation @Parameter @Variable @LocalVariable .
   subsort @Variable @Parameter < @LocalVariable < Vid .
   subsort @Operation @Parameter @LocalVariable < @StructuralFeature .
   op self : -> @LocalVariable .
   op KRESULT : -> @LocalVariable .
   op operations : @Class -> MyList . --- of @Operation
   op isAbstract : @Operation -> Bool .
   op from : @StructuralFeature -> Maybe{@Class} .
   op containingOperation : @Parameter -> @Operation .
   op containingOperation : @Variable  -> @Operation .
   op parameters : @Operation -> MyList . --- of @Parameter
   op variables : @Operation -> MyList .  --- of @Variable

   --- Label
   sort @Label @LabelNxt .
   op [_,_,_,_] : @Package @Class @Operation Nat -> @Label [ctor] .
   --- default label
   op [] : -> @Label .
   op <_,_> : @Label @Label -> @LabelNxt [ctor] .
   op nxt : @Label -> @LabelNxt .
   op labels : @Operation -> MyList . --- of @Label

   var LV : @LocalVariable .
   op defaultValue : @LocalVariable -> OCL-Type .
   eq defaultValue(LV) =
      if isMany(LV) then
         if isOrdered(LV) then
            if isUnique(LV) then
               OrderedSet{}
            else
               Sequence{}
            fi
         else
            if isUnique(LV) then
               Set{}
            else
               Bag{}
            fi
         fi
      else
```

```
         if type(LV) :: @DataType then
            defaultValue(type(LV))
         else
            null
         fi
      fi .
endm

view @Label from TRIV to KERMETA-SL-CTORS is
   sort Elt to @Label .
endv

view @Statement from TRIV to KERMETA-STMT-PREP is
   sort Elt to @Statement .
endv

view @Operation from TRIV to KERMETA-SL-CTORS is
   sort Elt to @Operation .
endv

view @LocalVariable from TRIV to KERMETA-SL-CTORS is
   sort Elt to @LocalVariable .
endv

view OCL-Type from TRIV to MGmOdCL is
    sort Elt to OCL-Type .
endv

mod KERMETA-STATEMENTS-MAP is
   pr MAP{@Label, @Statement} .
   pr MAP{@LocalVariable, OCL-Type} .
endm

mod KERMETA-SL is
   pr KERMETA-SL-CTORS .
   pr KERMETA-STATEMENTS-MAP .
   op statements : @Metamodel -> Map{@Label, @Statement} .
endm

--- ---------------------------------------------------------------------------------
--- Action Language Syntax
---
--- Body ::= [Stm]+
--- Stm  ::= lab: Stmt
--- Stmt ::= condStmt
---        |  assignStmt | castStmt
---        |  instanceCreationStmt
---        |  collStmt
---        |  returnStmt | callStmt
--- CondStmt             ::= <if> exp
--- AssignStmt           ::= lhs := exp
--- InstanceCreationStmt ::= var := exp
--- ReturnStmt           ::= <return> | <return> exp
--- CallStmt             ::= call | var := call
--- Call                 ::= target.op(exp*)
--- ---------------------------------------------------------------------------------
mod KERMETA-AL is
   pr KERMETA-STMT-PREP .
   pr KERMETA-SL .
    pr METAMODEL-PROP .
    pr MGMAYBE{@Operation} .

   sort    @CondStmt @AssignStmt @InstanceCreationStmt @Call @CallStmt @ReturnStmt .
   subsort @Call < @CallStmt .
   subsort @CondStmt @AssignStmt @InstanceCreationStmt @CallStmt @ReturnStmt < @Statement .
   sort    @CollItem .

   ops bag set seq oset : -> @CollItem .
   op iff_              : OCL-Exp -> @CondStmt                               [ctor] .
   op _.:=._            : OCL-Exp OCL-Exp -> @AssignStmt                     [ctor] .
   op _.:=. new‘(_‘)    : @Variable @Classifier -> @InstanceCreationStmt     [ctor] .
```

```
  op _.:=. new‘(_,_‘)  : @Variable @CollItem @Classifier -> @InstanceCreationStmt [ctor] .
  op return            : -> @ReturnStmt                                          [ctor] .
  op return_           : OCL-Exp -> @ReturnStmt                                  [ctor] .
  op _._<>             : OCL-Exp String -> @Call                                 [ctor] .
  op _._<_>            : OCL-Exp String List{OCL-Exp} -> @Call                   [ctor] .
  op _.:=._            : OCL-Exp @Call -> @CallStmt                              [ctor] .

  var CLASSIFIER : @Classifier .
  op default(_,_) :  @CollItem @Classifier -> OCL-Exp .
  eq default(bag,  CLASSIFIER) = Bag{} .
  eq default(set,  CLASSIFIER) = Set{} .
  eq default(seq,  CLASSIFIER) = Sequence{} .
  eq default(oset, CLASSIFIER) = OrderedSet{} .

  var ON : String .
  var C  : @Class .
  var O : @Operation .
  var L VALLIST PARAMLIST : MyList .
  var LV VAL PARAM : @LocalVariable .
  vars VPAIRS VPAIRSS VPSET : Set{VarPair} .
  var VPELT : VarPair .
  var LEXP : List{OCL-Exp} .
  var EXP  : OCL-Exp .

  op createLocalEnv : @Operation -> Set{VarPair} .
  eq createLocalEnv(O) = createLocalEnv$(variables(O)) .

  op createLocalEnv : @Operation List{OCL-Exp} -> Set{VarPair} .
  eq createLocalEnv(O, LEXP) = createParamBindings(parameters(O), LEXP) # createLocalEnv$(variables(O)) .

  op createLocalEnv$ : MyList -> Set{VarPair} . --- of @Variable
  eq createLocalEnv$(nil) = empty .
  eq createLocalEnv$(LV L) = putVar(LV <- defaultValue(LV), createLocalEnv$(L)) .

  op createParamBindings : MyList List{OCL-Exp} -> Set{VarPair} .
  eq createParamBindings(nil, mt-ord) = empty .
  eq createParamBindings(LV L, EXP # LEXP) = putVar(LV <- EXP, createParamBindings(L, LEXP)) .


  op lookup : String @Class -> @Operation .
  eq lookup(ON , C) =
     if(not ( getByName(ON , C) == null )) then
        getByName(ON , C)
     else
        findByName(ON , allSuperTypes(C) )
     fi
  .

  op findByName : String MyList -> @Operation . --- of @Class
  eq findByName(ON , C) = getByName(ON , C) .
  eq findByName(ON , C L) =
     if(not (getByName(ON , C) == null)) then
        getByName(ON , C)
     else
        findByName(ON , L)
     fi
  .

  op getByName : String @Class -> Maybe{@Operation} .
  eq getByName(ON , C) = getByName$(ON , operations(C)) .

  op getByName$ : String MyList -> Maybe{@Operation} . --- of @Operation
  eq getByName$(ON , nil) = null .
  eq getByName$(ON , O L) =
     if (ON == name(O)) then
        O
     else
        getByName$(ON , L)
     fi
  .
endm
```

```
mod KERMETA-DOMAIN is
   pr MGmOdCL .
   pr METAMODEL-PROP .
   pr KERMETA-SL-CTORS .
   ---pr MAP{@LocalVariable, OclType} .
   sort Domain .
   op <_##_> : @Model Set{VarPair} -> Domain [ctor] .
endm

mod KERMETA-STACK is
   pr KERMETA-SL .
   pr KERMETA-STATEMENTS-MAP .
   pr MGMAYBE{@LocalVariable} .

   sort StackEntry .
   op `(|_,_,_|`) : @Label Set{VarPair} Maybe{@LocalVariable} -> StackEntry .
endm

fmod STACK{X :: TRIV} is
   protecting BOOL .

   sorts NeStack{X} Stack{X} .
   subsort X$Elt  < NeStack{X} < Stack{X} .

   op NOPE : -> Stack{X} [ctor] .
   op _!!!_ : X$Elt Stack{X} -> NeStack{X} [ctor right id: NOPE] .

   var E : X$Elt .
   var S : Stack{X} .

   op isEmpty_ : Stack{X} -> Bool .
   eq isEmpty(NOPE) = true .
   eq isEmpty(S)    = false [owise] .
endfm

view StackEntry from TRIV to KERMETA-STACK is
   sort Elt to StackEntry .
endv

mod KERMETA-CONFIGURATION is
   pr KERMETA-SL .
   pr STACK{StackEntry} .
   pr KERMETA-DOMAIN .

   sort KConfig .
   op <|_,_,_,_|> : @Label Stack{StackEntry} Domain Nat -> KConfig .

   var LAB : @Label .
   var S : Stack{StackEntry} .
   var N : Nat .
   var D : Domain .

   op isStop : KConfig -> Bool .
   eq isStop(<| [], NOPE, D, N |>) = true .
   eq isStop(<| LAB, S, D, N |>) = false [owise] .
endm

mod KERMETA is
   pr KERMETA-AL .
   pr KERMETA-CONFIGURATION .

   --- Create a "fresh" Oid (i.e. not present in a model's config)
   --- Intended to be called with the "N" param in the config
   op newOid : Nat -> Oid .
   var N : Nat .
   eq newOid(N) = qid("O" + string(N, 10)) .

   --- update(M, O, REF, EXP) updates object O in model M
   --- with value EXP, preserving reference integrity
   op update : @Model Oid @Reference OCL-Exp -> @Model .
```

```
vars MM : @Metamodel .
vars O O' X Y : Oid .
var SFIS SFISO SFISO' SFISX SFISY : Set{@StructuralFeatureInstance} .
var CLASSO CLASSO' CLASSX CLASSY : @Class .
vars P Q : @Reference .
var OBJSET : Set{@Object} .
var VALO' VALX VALY : OCL-Exp .

ceq update(MM{< O : CLASSO | P : X  # SFISO > OBJSET}, O, REF, O') =
           MM{< O : CLASSO | P : O' # SFISO > OBJSET}
   if not(isMany(REF)) /\ null = opposite(REF) .

ceq update(MM{< O : CLASSO | P : X    # SFISO >
              < Y : CLASSY | P : O'   # SFISY >
              < X : CLASSX | Q : VALX # SFISX > OBJSET}, O, REF, O') =
          MM{ < O : CLASSO | P : O'   # SFISO >
              < Y : CLASSY | P : null # SFISY >
              < X : CLASSX | Q : null # SFISX > OBJSET}
   if not(isMany(P)) /\ not(isContainment(P)) /\ Q := opposite(P) .

ceq update(MM{< O  : CLASSO  | P : X     # SFISO  >
              < O' : CLASSO' | Q : VALO' # SFISO' >
              < Y  : CLASSY  | P : O'    # SFISY  >
              < X  : CLASSX  | Q : VALX  # SFISX  > OBJSET}, O, REF, O') =
          MM{ < O  : CLASSO  | P : O'   # SFISO  >
              < O' : CLASSO' | Q : O    # SFISO  >
              < Y  : CLASSY  | P : null # SFISO' >
              < X  : CLASSX  | Q : null # SFISX  > OBJSET}
   if not(isMany(P)) /\ isContainment(P) /\ Q := opposite(P) .

ceq update(MM{< O : CLASSO | P : X    # SFISO >
              < Y : CLASSY | P : O'   # SFISY >
              < X : CLASSX | Q : VALX # SFISX > OBJSET}, O, REF, O') =
           MM{< O : CLASSO | P : O'   # SFISO >
              < Y : CLASSY | P : null # SFISY >
              < X : CLASSX | Q : null # SFISX > OBJSET}
   if not(isMany(P)) /\ isContainer(P) /\ Q := opposite(P) .

vars LAB LABNxt LABELSE LABTHEN LABNULL : @Label .
vars VPSET VPSETT : Set{VarPair} .
vars S : Stack{StackEntry} .
vars M MP : @Model .

vars E RES INST OLD EXP : OCL-Exp .
var OID : Oid .
vars VAR  MYVAR : @Variable .
var MAYBEVAR : Maybe{@LocalVariable} .
var CLASS : @Class .
var CLASSIFIER : @Classifier .
var SE : StackEntry .
var COLLITEM : @CollItem .
var OPNAME : String .
var OP : @Operation .
var LO LEXP : List{OCL-Exp} .
var K : KConfig .
var ATT : @Attribute .
var REF : @Reference .

op run : KConfig -> KConfig .
eq run(K) = if isStop(K) then K else run(exec(K)) fi .

op exec : KConfig -> KConfig [iter] .

--- CondStmt
ceq exec( <| LAB, S, < M ## VPSET >, N |> ) = <| LABTHEN, S, < M ## VPSET >, N |>
-------------------------------------------------------------------------------------
   if  iff( E ) := statements( meta(M) ) [ LAB ] /\
       < LABTHEN , LABELSE > := nxt(LAB)          /\
       true = << E ; env(VPSET) ; M >>
.

```

```
ceq exec(<| LAB, S, < M ## VPSET >, N |>) = <| LABELSE, S, < M ## VPSET >, N |>
--------------------------------------------------------------------------------------
    if  iff( E ) := statements( meta(M) ) [ LAB ] /\
        < LABTHEN , LABELSE > := nxt(LAB)         /\
        false = << E ; env(VPSET) ; M >>
.
--- ReturnStmt
ceq exec(<| LAB,   (| LABNxt , VPSETT, null |) !!! S, < M ## VPSET >,  N |>) =
         <| LABNxt,                                  S, < M ## VPSETT >, N |>
--------------------------------------------------------------------------------------
    if return :=  statements( meta(M) ) [ LAB ]
.
ceq exec(<| LAB, (| LABNxt , VPSETT, MYVAR |) !!! S, < M ## VPSET >,                          N |>) =
         <| LABNxt,                                S, < M ## putVar( MYVAR <- RES , VPSETT) >, N |>
------------------------------------------------------------------------------------------------------
    if  return( E ) :=  statements( meta(M) ) [ LAB ] /\
                RES := << E ; env(VPSET) ; M >>
.
ceq exec(<| LAB, NOPE, < M ## VPSET >, N |>) =
         <| [] , NOPE, < M ## VPSET >, N |>
--------------------------------------------------------------------------------------
    if return :=  statements( meta(M) ) [ LAB ]
.
ceq exec(<| LAB, NOPE, < M ## VPSET >,                            N |>) =
         <| [] , NOPE, < M ## putVar( KRESULT <- RES , VPSET) >, N |>
    if  return( E ) :=  statements( meta(M) ) [ LAB ] /\
                RES := << E ; env(VPSET) ; M >>
.
--- NewInstStmt: depends on the object's type created
ceq exec(<| LAB,    S, < MM { OBJSET } ## VPSET >, N |>) =
         <| LABNxt, S, < MM { complete(< newOid(N) : CLASSIFIER | empty >) OBJSET } ##
                                  (putVar(MYVAR <- newOid(N), VPSET)) >, (N + 1) |>
       if  MYVAR .:=. new ( CLASSIFIER ) := statements( MM ) [ LAB ] /\
       < LABNxt , LABNULL >               := nxt(LAB)                 /\
       CLASSIFIER :: @Class
.
ceq exec(<| LAB,  S, < MM { OBJSET } ## VPSET >,                                          N |>) =
       <| LABNxt, S, < MM { OBJSET } ## (putVar(MYVAR <- defaultValue(CLASSIFIER), VPSET)) >, N |>
       if  MYVAR .:=. new ( CLASSIFIER ) := statements( MM ) [ LAB ] /\
       < LABNxt , LABNULL >               := nxt(LAB)                 /\
       CLASSIFIER :: @DataType
.
ceq exec(<| LAB, S, < MM {OBJSET} ## VPSET >, N |>) =
         <| LABNxt, S, < MM {OBJSET} ## (putVar(MYVAR <- default(COLLITEM , CLASSIFIER), VPSET)) >, N |>
       if  MYVAR .:=. new ( COLLITEM , CLASSIFIER ) := statements( MM ) [ LAB ] /\
       < LABNxt , LABNULL >                          := nxt(LAB)
.
--- CallStmt
ceq exec(<| LAB, S, < MM { < OID : CLASS | SFIS > OBJSET } ## VPSET >, N |>) =
       <| [ package(containingClass(OP)) , containingClass(OP) , OP , 1 ] ,
          (| LABNxt , VPSET, null |) !!! S,
          < MM { < OID : CLASS | SFIS > OBJSET } ## (putVar(self <- OID, createLocalEnv(OP))) >, N |>
    if INST . OPNAME <>  := statements( MM ) [ LAB ] /\
    < LABNxt , LABNULL > := nxt(LAB)                 /\
    OID                  := << INST ; env(VPSET) ; MM { < OID : CLASS | SFIS > OBJSET } >> /\
    OID                  :: Oid /\
    OP                   := lookup(OPNAME, CLASS)
. --- Having O : CLASS in the to-be-rewritten KConfig causes Maude trying to match ALL
--- objects in the model before giving up
ceq exec(<| LAB, S, < MM { < OID : CLASS | SFIS > OBJSET } ## VPSET >, N |>) =
         <| [ package(containingClass(OP)) , containingClass(OP) , OP , 1 ] ,
         (| LABNxt , VPSET, null |) !!! S,
         < MM { < OID : CLASS | SFIS > OBJSET } ## (putVar(self <- OID, createLocalEnv(OP, LEXP))) >, N
             |>
    if INST . OPNAME < LO > := statements( MM ) [ LAB ] /\
    < LABNxt , LABNULL >    := nxt(LAB) /\
    LEXP                    := eval-EL(LO, env(VPSET), none) /\
    OID                     := << INST ; env(VPSET) ; MM { < OID : CLASS | SFIS > OBJSET } >> /\
    OID                     :: Oid /\
    OP                      := lookup(OPNAME, CLASS)
.
```

```
  ceq exec(<| LAB, S, < MM { < OID : CLASS | SFIS > OBJSET } ## VPSET >, N |>) =
           <| [ package(containingClass(OP)) , containingClass(OP) , OP , 1 ] ,
           (| LABNxt , VPSET, MYVAR |) !!! S,
           < MM { < OID : CLASS | SFIS > OBJSET } ## (putVar(self <- OID, createLocalEnv(OP))) >, N |>
     if MYVAR .:=. INST . OPNAME <> := statements( MM ) [ LAB ] /\
     < LABNxt , LABNULL >           := nxt(LAB) /\
     OID                            := << INST ; env(VPSET) ; MM { < OID : CLASS | SFIS > OBJSET } >> /\
     OID                            :: Oid /\
     OP                             := lookup(OPNAME, CLASS)
  . --- Having O : CLASS in the to-be-rewritten KConfig causes Maude trying to match ALL
  --- objects in the model before giving up
  ceq exec(<| LAB, S, < MM { < OID : CLASS | SFIS > OBJSET } ## VPSET >, N |>) =
           <| [ package(containingClass(OP)) , containingClass(OP) , OP , 1 ] ,
           (| LABNxt , VPSET, MYVAR |) !!! S,
         < MM { < OID : CLASS | SFIS > OBJSET } ## (putVar(self <- OID, createLocalEnv(OP, LEXP))) >, N |>
     if MYVAR .:=. INST . OPNAME < LO > := statements( MM ) [ LAB ] /\
     < LABNxt , LABNULL >             := nxt(LAB) /\
     LEXP                             := eval-EL(LO, env(VPSET), none) /\
     OID                              := << INST ; env(VPSET) ; MM { < OID : CLASS | SFIS > OBJSET } >> /\
     OID                              :: Oid /\
     OP                               := lookup(OPNAME, CLASS)
  .
  --- AssignStmt
  ceq exec(<| LAB,    S, < MM { OBJSET } ## ((MYVAR <- OLD) # VPSET) >,      N |>) =
           <| LABNxt, S, < MM { OBJSET } ## (putVar(MYVAR <- RES, VPSET)) >, N |>
        if  MYVAR .:=. E         := statements( MM ) [ LAB ] /\
           < LABNxt , LABNULL > := nxt(LAB) /\
           RES                  := << E ; env((MYVAR <- OLD) # VPSET) ; MM { OBJSET } >>
  .
  ceq exec(<| LAB,    S, < MM { < O : CLASS | ATT : OLD # SFIS > OBJSET } ## VPSET >, N |>) =
           <| LABNxt, S, < MM { < O : CLASS | ATT : RES # SFIS > OBJSET } ## VPSET >, N |>
        if  INST . ATT .:=. E   := statements( MM ) [ LAB ] /\
           < LABNxt , LABNULL > := nxt(LAB) /\
           O                 := << INST ; env(VPSET) ; MM { < O : CLASS | ATT : OLD # SFIS > OBJSET } >> /\
           O                 :: Oid /\
           RES               := << E    ; env(VPSET) ; MM { < O : CLASS | ATT : OLD # SFIS > OBJSET } >>
  .
  ceq exec(<| LAB,    S, < MM { < O : CLASS | REF : OLD # SFIS > OBJSET } ## VPSET >, N |>) =
         <| LABNxt, S, < update(MM { < O : CLASS | REF : RES # SFIS > OBJSET }, O, REF, RES) ## VPSET >, N
            |>
        if  INST . REF .:=. E   := statements( MM ) [ LAB ] /\
           < LABNxt , LABNULL > := nxt(LAB) /\
           O                 := << INST ; env(VPSET) ; MM { < O : CLASS | REF : OLD # SFIS > OBJSET } >> /\
           O                 :: Oid /\ --- not(isMany(REF)) /\ opposite(REF) == null /\
           RES               := << E ; env(VPSET) ; MM { < O : CLASS | REF : OLD # SFIS > OBJSET } >>
  .
endm
```

C Summary of Publications

This Appendix provides a comprehensive list of all our publications, sorted by publication kind. The list highlightings the relationship between the publication entries, and explains how each publication is related to this Manuscript content.

Journal

[J1] This publication provides a survey, articulated around a simple taxonomy of characteristics for evaluating the impact, relevance and usability of existing Model-Driven Security approaches. It also sketches some directions about how an ideal Model-Driven Security would look like. This complements the work presented in [C1].

Levi Lúcio, Zhang Qin, et al. (2014). "Advances in Model-Driven Security". In: *Advances in Computer Science*, (submitted, under review)

[J2] This publication is at the basis of Chapter 4, and significantly extends [W1].

Moussa Amrani et al. (2013). "Model Transformation Intents and Their Properties". In: *Journal of Software And Systems (*SoSyM*)*, (submitted, under review)

[J3] This publication is an extended version of [W3].

Moussa Amrani et al. (2014). "A Survey of Formal Verification Techniques for Model Transformations: A Tridimensional Classification". In: *Journal of Technology (*JoT*)*, (submitted, under review)

Book Chapter

[B1] This double-blind, double rounded reviewed publication contains the formal semantics of Kermeta, from which Chapters 6 and 7 are directly inspired.

Moussa Amrani (2013). "A Formal Semantics of Kermeta". In: *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments.* Ed. by Marjan Mernik. Igi Global. Chap. 10, pp. 270–309

Conference

[C1] This paper presents a policy-based approach named *Security@Runtime* for automating the integration of security mechanisms into Java-based business applications. This work complements the topic of the Ph.D by defining an real-life Domain-Specific Language for specifying security configurations by means of authorisations, obligations and reactions, which are enforced using the classical Pep/Pdp (Policy Enforcement/Decision Points) approach.

Yehia Elrakaiby, Moussa Amrani, and Yves Le Traon (2014). “Security@Runtime: A Flexible MDE approach to Enforce and Manage Fine-grained Advanced Security Policies”. In: *Proceedings of the International Symposium on Engineering Secure Software and Systems (*ESSoS*)*

Workshop

[W1] This paper is a preliminary exploration of the concepts of intent and property mapping developed in Chapter 4, demonstrated with only one intent.

Moussa Amrani et al. (2012a). “Towards a Model Transformation Intent Catalog”. In: *Proceedings of the First Workshop on Analysis of Model Transformations (*AMT*)*

[W2] Levi Lúcio, Eugene Syriani, et al. (2012). “Invariant Preservation In Iterative Modeling”. In: *Workshop on Models and Evolution (*ME*)*

[W3] This paper first started as a state-of-the-art in the domain of formal verification of model transformations, from which Chapter 3 is directly inspired. It will be extended as a Journal paper in [J3].

Moussa Amrani et al. (2012b). “A Tridimensional Approach for Studying the Formal Verification of Model Transformations”. In: *Proceedings of the First Workshop on Verification And Validation of Model Transformations*

Technical Report

[T1] The following Report is an extended version of [W2]: it provides full details of the mathematical proofs, as well as complete specifications of the running example used in [W2].

Moussa Amrani, Levi Lúcio, Eugene Syriani, et al. (2013). *Invariant Preservation In Iterative Modeling. (Extended Version).* Tech. rep. TR-LASSY-12-13. Laboratory of Advanced Software and Systems (LASSY)

[T2] This Report completes [B1] with the missing details of the semantics specification that were omitted for space reasons, as well as an extended state-of-the-art on the formalisation of Kermeta’s languages.

Moussa Amrani (2011). *A Formal Semantics of Kermeta.* Tech. rep. TR-LASSY-12-12. Available at. University of Luxembourg

[T3] This work reports on a preliminary effort to specify Kermeta’s Structural Language, using Z (Spivey 1992) as a validation tool. However, this path was stopped because it was difficult to obtain a fully executable specification needed as a prerequisite for formal verification of dynamic properties.

Moussa Amrani and Nuno Amálio (2011). *A Set-Theoretic Formal Specification of the Semantics of Kermeta.* Tech. rep. TR-LASSY-11-03. Available at `http://bit.ly/ju4NcG`. University of Luxembourg

Bibliography

Adrion, W. Richards, Martha A. Branstad, and John C. Cherniavsky (1982). “Validation, Verification, and Testing of Computer Software”. In: *ACM Comput. Surv.* 14.2, pp. 159–192 (cit. on p. 52).

Agrawal, Aditya, Gabor Karsai, Zsolt Kalmar, Sandeep Neema, Feng Shi, and Attila Vizhanyo (2006). “The Design of a Language for Model Transformations”. In: *Software and Systems Modeling* 5.3, pp. 261–288 (cit. on p. 56).

Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman (1986). *Compilers: Principles, Techniques, and Tools.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0-201-10088-6 (cit. on pp. 117–119).

Akehurst, David H., Stuart Kent, and Octavian Patrascoiu (2003). “A Relational Approach to Defining Transformations in a Metamodel”. In: *Proceedings of the 5th International Conference on The Unified Modeling Language (UML).* Vol. 2. 4, pp. 215–239 (cit. on pp. 40, 42, 78, 79).

Alanen, Marcus and Ivan Porres (2003). *A Relation Between Context-Free Grammars and Meta-Object Facility Metamodels.* Tech. rep. TUCS Turku Center for Computer Science (Finland) (cit. on p. 28).

Amrani, Moussa (2011). *A Formal Semantics of Kermeta.* Tech. rep. TR-LASSY-12-12. Available at. University of Luxembourg (cit. on pp. 89, 198).

— (2013). “A Formal Semantics of Kermeta”. In: *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments.* Ed. by Marjan Mernik. IGI Global. Chap. 10, pp. 270–309 (cit. on pp. 89, 197).

Amrani, Moussa and Nuno Amálio (2011). *A Set-Theoretic Formal Specification of the Semantics of Kermeta.* Tech. rep. TR-LASSY-11-03. Available at `http://bit.ly/ju4NcG`. University of Luxembourg (cit. on pp. 89, 198).

Amrani, Moussa, Jürgen Dingel, Leen Lambers, Levi Lúcio, Rick Salay, Gehan Selim, Eugene Syriani, and Manuel Wimmer (2012a). “Towards a Model Transformation Intent Catalog”. In: *Proceedings of the First Workshop on Analysis of Model Transformations (AMT)* (cit. on pp. 9, 49, 85, 198).

— (2013). “Model Transformation Intents and Their Properties”. In: *Journal of Software And Systems (SoSyM)* (cit. on pp. 9, 60, 197).

Amrani, Moussa, Levi Lúcio, Gehan Selim, Benoît Combemale, Jürgen Dingel, Hans Vangheluwe, Yves Le Traon, and James R. Cordy (2012b). “A Tridimensional Approach for Studying the Formal Verification of Model Transformations”. In: *Proceedings of the First Workshop on Verification And Validation of Model Transformations* (cit. on pp. 9, 17, 51, 52, 86, 198).

— (2014). “A Survey of Formal Verification Techniques for Model Transformations: A Tridimensional Classification”. In: *Journal of Technology (JoT)* (cit. on p. 197).

Amrani, Moussa, Levi Lúcio, Eugene Syriani, Qin Zhang, and Hans Vangheluwe (2013). *Invariant Preservation In Iterative Modeling. (Extended Version)*. Tech. rep. TR-LASSY-12-13. Laboratory of Advanced Software and Systems (Lassy) (cit. on p. 198).

Anastasakis, Kyriakos, Behzad Bordbar, and Jochen M. Küster (2007). "Analysis of Model Transformations *via* Alloy". In: MoDeVVa, pp. 47–56 (cit. on pp. 44, 46).

André, Charles, Frédéric Mallet, and Robert de Simone (2007). "Modeling Time(s)". In: *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (*MoDELS/Uml*)*. Vol. 4735. Lecture Notes on Computer Science. Springer, pp. 559–573 (cit. on pp. 174, 177).

André, Sylvain (2004). "Model-Driven Architecture: Principles and State of the Art". (in French). MA thesis. Conservatoire National d'Arts et Metiers (cit. on p. 1).

Arendt, Thorsten, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer (2010). "Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations". In: *International Conference on Model Driven Engineering Languages and Systems (*MoDELS*)*. Ed. by Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen. Vol. 6394. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 121–135 (cit. on p. 29).

Asztalos, Márk, László Lengyel, and Tihamer Levendovszky (2010). "Towards Automated, Formal Verification of Model Transformations". In: Icst (cit. on pp. 44, 46).

Asztalos, Márk, Eugene Syriani, Manuel Wimmer, and Marouane Kessentini (2011). "Simplifying Model Transformation Chains By Rule Composition". In: *Models in Software Engineering - Workshops and Symposia at MODELS 2010, Reports and Revised Selected Papers*. Vol. 6627. Lncs, pp. 293–307 (cit. on p. 56).

Atkinson, Colin and Thomas Kühne (2002a). "Profiles in a Strict Metamodeling Framework". In: *Science of Computer Programming* 44.1, pp. 5–22 (cit. on p. 12).

— (2002b). "Rearchitecting the UML Infrastructure". In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 12.4, pp. 290–321 (cit. on p. 12).

— (2007). "A Tour of Language Customization Concepts". In: *Advances in Computers* 70, pp. 105–161 (cit. on p. 26).

(Atl), The Atlas Transformation Language. Available at http://www.eclipse.org/m2m/atl/ (cit. on p. 20).

Bae, Jung Ho, KwangMin Lee, and Heung Seok Chae (2008). "Modularization of the UML Metamodel Using Model Slicing". In: *ITNG'08*. IEEE, pp. 1253–1254 (cit. on p. 62).

Baresi, Luciano, Reiko Heckel, Sebastian Thöne, and Dániel Varró (2006). "Style-Based Modeling and Refinement of Service Oriented Architectures". In: *Journal of Software and Systems Modeling* 5 (2), pp. 187–207 (cit. on p. 64).

Barišic, Ankica, Vasco Amaral, Miguel Goulão, and Bruno Barroca (2014). "Evaluating the Usability of Domain-Specific Languages". In: *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments.* Ed. by Marjan Mernik. Igi Global. Chap. 14, pp. 386–407 (cit. on p. 26).

Barroca, Bruno, Levi Lúcio, Vasco Amaral, Roberto Félix, and Vasco Sousa (2010). "DslTrans: A Turing-Incomplete Transformation Language". In: Sle (cit. on pp. 38, 39, 42, 43, 46).

Becker, Basil, Dirk Beyer, Holger Giese, Florian Klein, and Daniela Schilling (2006). "Symbolic Invariant Verification For Systems With Dynamic Structural Adaptation". In: Icse. Shanghai, China. isbn: 1-59593-375-1 (cit. on pp. 41, 42, 44, 46).

Berdine, Josh, Byron Cook, Dino Distefano, and Peter W. O'Hearn (2006). "Automatic Termination Proofs for Programs with Shape-Shifting Heaps". In: *Computer-Aided Verification (*CAV*)*. Vol. 4144. LNCS, pp. 386–400 (cit. on pp. 38, 42).

Bergmann, Gábor, Zoltán Ujhelyi, István Ráth, and Dániel Varró (2011). "A Graph Query Language for EMF Models". In: *Proceedings of the 4th International Conference on Theory and Practice of Model Transformations (*ICMT*)*. Vol. 6707. LNCS. Springer, pp. 167–182 (cit. on p. 55).

Bertot, Yves and Pierre Castéran (2004). *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer (cit. on p. 34).

Bettini, Lorenzo (2013). *Implementing Domain-Specific Languages with Xtext and Xtend*. Learn To Implement a DSL with Xtext and Xtend using easy-to-understand examples and best practices. PACKT Publishing. 342 pp. (cit. on p. 19).

Bézivin, Jean, Fabian Büttner, Martin Gogolla, Frédéric Jouault, Ivan Kurtev, and Arne Lindow (2006). "Model Transformations? Transformation Models!" In: MoDELS (cit. on pp. 18, 39, 45, 66).

Biermann, Enrico (2011). "Local Confluence Analysis of Consistent EMF Transformations". In: ECEASST 38, pp. 68–84 (cit. on pp. 38, 42).

Biermann, Enrico, Karsten Ehrig, Claudia Ermel, and Jonas Hurrelmann (2009). "Generation of Simulation Views for Domain Specific Modeling Languages Based on the Eclipse Modeling Framework". In: *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pp. 625–629 (cit. on p. 71).

Biermann, Enrico, Claudia Ermel, and Gabriele Taentzer (2008). "Precise Semantics of EMF Model Transformations by Graph Transformation". In: *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*. MoDELS '08. Berlin, Heidelberg: Springer-Verlag, pp. 53–67. URL: http://dx.doi.org.proxy.bnl.lu/10.1007/978-3-540-87875-9%5C_4 (cit. on p. 114).

Biggerstaff, Ted James (1998). "A Perspective of Generative Reuse". In: *Annals of Software Engineering* 5, pp. 169–226 (cit. on p. 24).

Bohlen, Matthias, Chad Brandon, Martin West, Carlos Cuenca, Peter Friese, Naresh Bhatia, Steve Jerman, Joel Kozikowski, Bob Fields, Michail Plushnikov, and Vance Karimi. *The AndroMDA Website*. URL: http://www.andromda.org (cit. on p. 19).

Boocock, Paul. *The Jamda Website*. URL: http://jamda.sourceforge.net (cit. on p. 19).

Börger, Egon and Robert Stärk (2003). *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag (cit. on pp. 28, 70, 176).

Boronat, Artur (2007). "MOMENT: A Formal Framework for MOdel manageMENT". PhD thesis. University of Valencia (cit. on pp. 39, 42).

Boronat, Artur and José Meseguer (2008). "An Algebraic Semantics For MOF". In: *FASE'08/ETAPS'08: Proceedings of the Theory and Practice of Software, 11th International Conference on Fundamental Approaches to Software Engineering*. Berlin, Heidelberg: Springer-Verlag, pp. 377–391 (cit. on pp. 114, 147, 148).

Bouhoula, Adel, Jean-Pierre Jouannaud, and José Meseguer (2000). "Specification and Proof in Membership Equational Logic". In: *Theoretical Computer Science* 236.1, pp. 35–132 (cit. on p. 138).

Boulanger, Jean-Louis, ed. (2011). *Static Analysis of Software – The Abstract Interpretation*. Wiley (cit. on p. 2).

Bruggink, H.J. Sander. (2008). "Towards a Systematic Method for Proving Termination of Graph Transformation Systems". In: ENTCS **213**(1) (cit. on pp. 38, 42).

Brunelière, Hugo, Jordi Cabot, Cauê Clasen, Frédéric Jouault, and Jean Bézivin (2010). "Towards Model Driven Tool Interoperability: Bridging Eclipse and Microsoft Modeling Tools". In: *European Conference on Modelling Foundations and Applications (*ECMFA*)*. Vol. 6138. Lecture Notes in Computer Science (LNCS), pp. 32–47 (cit. on p. 66).

Bryant, Barrett R., Jeff Gray, Marjan Mernik, Peter J. Clarke, Robert B. France, and Gabor Karsai (2011). "Challenges and Directions in Formalizing the Semantics of Modeling Languages". In: *Journal of Computer Science and Information Systems* 8.2 (Special Issue on Advances in Formal Languages, Modeling and Applications), pp. 225–253 (cit. on p. 28).

Büttner, Fabian, Marina Egea, and Jordi Cabot (2012). "On Verifying ATL Transformations Using 'off-the-shelf' SMT Solvers". In: *International Conference on Model-Driven Engineering Languages and Systems (MoDELS)*. Springer, pp. 432–448 (cit. on p. 78).

Büttner, Fabian, Marina Egea, Jordi Cabot, and Martin Gogolla (2012). "Verification of ATL Transformations Using Transformation Models and Model Finders". In: *14th International Conference on Formal Engineering Methods (*ICFEM*)*. LNCS 7635. Springer, pp. 198–213 (cit. on pp. 78, 79, 174).

Cabot, Jordi, Robert Clarisó, Esther Guerra, and Juan de Lara (2010). "Verification and Validation of Declarative Model-to-Model Transformations Through Invariants". In: *Journal of Systems and Software* **83**(2), pp. 283–302 (cit. on p. 67).

Calegari, Daniel and Nora Szasz (2013). "Verification of Model Transformations: A Survey of the State-of-the-Art". In: *Electronic Notes in Theoretical Computer Science* 292, pp. 5–25 (cit. on pp. 48, 51, 52, 86).

Cardelli, Luca (2004). "Type Systems". In: *The Computer Science and Engineering Handbook*. CRC Press (cit. on p. 120).

Cariou, Éric, Nicolas Belloir, Franck Barbier, and Nidal Djemam (2009). "OCL Contracts For The Verification Of Model Transformations". In: ECEASST 24 (cit. on p. 78).

Castagna, Giuseppe and Zhiwu Xu (2011). "Set-Theoretic Foundation of Parametric Polymorphism and Subtyping". In: *16th ACM SIGPLAN International Conference on Functional Programming (*ICFP*)* (cit. on pp. 94, 174).

Chechik, Marsha, Shiva Nejati, and Mehrdad Sabetzadeh (2011). "A Relationship-Based Approach to Model Integration". In: ISSE 7 (cit. on pp. 40, 42).

Chen, Kai, Janos Sztipanovits, and Sandeep Neema (2005). "Toward a Semantic Anchoring Infrastructure for Domain-Specific Modeling Languages". In: *Proceedings of the 5th ACM International Conference on Embedded Software*. EMSOFT '05. Jersey City, NJ, USA: ACM, pp. 35–43. ISBN: 1-59593-091-4. DOI: `http://doi.acm.org.proxy.bnl.lu/10.1145/1086228.1086236`. URL: `http://doi.acm.org.proxy.bnl.lu/10.1145/1086228.1086236` (cit. on p. 28).

Cheung, Shing Chi and Jeff Kramer (1999). "Checking Safety Properties Using Compositional Reachability Analysis". In: ACM *Transactions on Software Engineering Methodologies* 8.1, pp. 49–78 (cit. on p. 84).

Cicchetti, Antonio, Davide Di Ruscio, R Eramo, and Alfonso Pierantonio (2008). "Automating co-evolution in model-driven engineering". In: *International IEEE Enterprise Distributed Object Computing Conference*. IEEE Computer Society, pp. 222–231 (cit. on p. 56).

Clark, Tony, Andy Evans, Paul Sammut, and James Willans (2004). *Applied Metamodelling: A Foundation for Language Driven Development*. Ceteva, Sheffield (cit. on p. 71).

Clark, Tony, Paul Sammut, and James Willans (2008). *Superlanguages: Developing Languages and Applications with* XMF. Ceteva (cit. on p. 132).

Clarke, Edmund M., Orna Grumberg, and Doron A. Peled (1999). *Model-Checking*. The MIT Press (cit. on pp. 2, 81).

Clarke, Edmund M. and Jeannette M. Wing (1996). "Formal methods: state of the art and future directions". In: *ACM Comput. Surv.* 28.4, pp. 626–643 (cit. on p. 52).

Clavel, Manuel, Francisco Duran, Steven Eker, Patrick Lincoln, Narciso Marti-Oliet, Jose Meseguer, and Carolyn Talcott (2007). *All About* MAUDE*. A High-Performance Logical Framework*. Vol. 4350. Lecture Notes in Computer Science (LNCS). Springer (cit. on pp. 29, 70, 137–139, 141, 147).

Combemale, Benoit, Xavier Crégut, Pierre-Loïc Garoche, and Xavier Thirioux (2009). "Essay on Semantics Definition in MDE. An Instrumented Approach for Model Verification". In: *Journal of Software* 4.9, pp. 943–958. URL: http://www.academypublisher.com/ojs/index.php/jsw/article/view/0409943958 (cit. on pp. 28, 29, 41, 132).

Combemale, Benoit, Julien Deantoni, Matias Vara Larsen, Fr'ed'eric Mallet, Olivier Barais, Benoit Baudry, and Robert France (2013). "Reifying Concurrency for Executable Metamodeling". In: *International Conference on Software Language Engineering (*SLE*)*. Ed. by Richard F. Paige Martin Erwig and Eric van Wyk. Lecture Notes in Computer Science. Springer-Verlag (cit. on p. 177).

Combemale, Benoit, C'ecile Hardebolle, Christophe Jacquet, Fr'ed'eric Boulanger, and Benoit Baudry (2012). "Bridging the Chasm between Executable Metamodeling and Models of Computation". In: *International Conference on Software Language Engineering (*SLE*)*. Lecture Notes in Computer Science. Springer (cit. on p. 177).

Costagliola, Gennaro, Andrea Delucia, Sergio Orefice, and Giuseppe Polese (2002). "A Classification Framework to Support the Design of Visual Languages". In: *Journal of Visual Languages & Computing* 13.6, pp. 573–600 (cit. on p. 27).

Cousot, Patrick and Radhia Cousot (2010). "Logics and Languages for Reliability and Security". In: ed. by Javier Esparza, Orna Grumberg, and Manfred Broy. NATO Series III: Computer and Systems Sciences. IOS Press. Chap. A Gentle Introduction to Formal Verification of Computer Systems by Abstract Interpretation, pp. 1–29 (cit. on pp. 31, 35).

Czarnecki, Krzysztof and Simon Helsen (2006). "Feature-Based Survey of Model Transformation Approaches". In: IBM *Systems J.* **45**(3), pp. 621–645 (cit. on pp. 19, 21, 37, 49, 51, 54, 85).

Dalal, S. R., A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz (1999). "Model-based testing in practice". In: *International Conference on Software Engineering*. Los Angeles: ACM Press, pp. 285–294 (cit. on p. 56).

Darlington, John and R.M. Burstall (1976). "A System Which Automatically Improves Programs". In: *Acta Informatica* 6.1, pp. 41–60 (cit. on p. 20).

De Lara, Juan, Esther Guerra, Artur Boronat, Reiko Heckel, and Paolo Torrini (2010). "Graph Transformation for Domain-Specific Discrete Event Time Simulation". In: *International Conference on Graph Transformation (*ICGT*)*, pp. 266–281 (cit. on p. 71).

De Lara, Juan and Gabriele Taentzer (2004). “Automated Model Transformation and its Validation Using AToM3 and AGG”. In: *Diagrammatic Representation and Inference (Diagrams*, pp. 182–198 (cit. on p. 67).

De Lara, Juan and Hans Vangheluwe (2002). “Using ATOM^3 as a Meta-CASE Tool”. In: ICEIS, pp. 642–649 (cit. on p. 20).

— (2004). “Defining Visual Notations and their Manipulation Through Meta-Modelling and Graph Transformation”. In: *Journal of Visual Languages & Computing* 15.3 – 4, pp. 309–330 (cit. on p. 70).

— (2010). “Automating the Transformation-Based Analysis of Visual Languages”. In: FAC **22**(3-4), pp. 297–326 (cit. on pp. 28, 44, 46).

De Roever, Willem-Paul, Frank de Boer, Ulrich Hanneman, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers (2001). *Concurrency Verification: Introduction to Compositional and Non-Compositional Methods*. Cambridge University Press (cit. on p. 84).

Deltombe, Gaëtan, Olivier Le Goaer, and Franck Barbier (2012). “Bridging KDM and ASTM for Model-Driven Software Modernization”. In: *International Conference on Software Engineering & Knowledge Engineering (*SEKE*)*, pp. 517–524 (cit. on pp. 16, 66).

Denil, Joachim (2013). “Design, Verification and Deployment of Software-Intensive Systems: A Multi-Paradigm Modelling Approach”. PhD thesis. Antwerp University (cit. on pp. 73, 75).

Denil, Joachim, Antonio Cicchetti, Matthias Biehl, Paul De Meulenaere, Romina Eramo, Serge Demeyer, and Hans Vangheluwe (2012). “Automatic Deployment Space Exploration Using Refinement Transformations”. In: *Recent Advances in Multi-paradigm Modelling*. 50 (cit. on p. 55).

Dorf, Richard C. (2011). *Modern Control Systems*. 12th. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. (cit. on p. 75).

Drey, Zoé, Cyril Faucher, Franck Fleurey, Vincent Mahé, and Didier Vojtisek (2009). *Kermeta Language — Reference Manual*. University of Rennes, Triskell Team (cit. on pp. 93, 95, 97, 109–111, 114–116, 123, 132, 179).

D’Silva, Vijay, Daniel Kroening, and Georg Weissenbacher (2008). “A Survey of Automated Techniques for Formal Software Verification”. In: *IEEE Trans. on CAD of Integrated Circuits and Systems* 27.7, pp. 1165–1178 (cit. on p. 52).

Ducasse, Stéphane and Tudor Gîrba (2006). “Using Smalltalk as a Reflective Executable Meta-language”. In: *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (*MODELS*)*, pp. 604–618 (cit. on p. 71).

Duffy, David A. (1991). *Principles of Automated Theorem-Proving*. Wiley & Sons (cit. on p. 2).

Durán, Francisco (1999). “A Reflective Module Algebra with Application to the Maude Language”. PhD thesis. University of Málaga (Spain) (cit. on p. 141).

Durán, Francisco, Martin Gogolla, and Manuel Roldán (2011). “Tracing Properties of UML and OCL Models With Maude”. In: *Proceeding of the Second International Workshop on Algebraic Methods in Model-based Software Engineering (*AMMSE*)*. Ed. by Francisco Durán and Vlad Rusu. Vol. 56. Electronic Proceedings in Theoretical Computer Science (cit. on pp. 148, 175).

Durán, Francisco and Manuel Roldán (2011). “Dynamic Validation of OCL Constraints with mOdCL”. In: *Proceedings of the International Workshop on* OCL *and Textual Modelling*. Ed. by Jordi Cabot, Robert Clarisó, Martin Gogolla, and Burkhart Wolff (cit. on pp. 148, 175).

Ehrig, Hartmut, Karsten Ehrig, Gabriele Taentzer, Juan de Lara, Dániel Varró, and Szilvia Varró-Gyapai (2005). “Termination Criteria for Model Transformation”. In: Fase (cit. on pp. 38, 42, 43, 46).

Ehrig, Hartmut and Bernd Mahr (1985). *Fundamentals of Algebraic Specifications*. Springer-Verlag (cit. on p. 99).

Elrakaiby, Yehia, Moussa Amrani, and Yves Le Traon (2014). “Security@Runtime: A Flexible MDE approach to Enforce and Manage Fine-grained Advanced Security Policies”. In: *Proceedings of the International Symposium on Engineering Secure Software and Systems (*ESSoS*)* (cit. on p. 198).

Engel, Klaus-D., Richard Paige, and Dimitrios Kolovos (2006). “Using a Model Merging Language for Reconciling Model Versions”. In: *Model Driven Architecture-Foundations and Applications*. Ed. by Arend Rensink and Jos Warmer. Vol. 4066. LNCS. Springer, pp. 143–157 (cit. on p. 57).

Engels, Gregor, Jan Hendrik Hausmann, Reiko Heckel, and Stefan Sauer (2000). “Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML”. In: *International Conference on The Unified Modeling Language — Advancing the Standard*, pp. 323–337 (cit. on pp. 67, 71, 72).

Ermel, Claudia and Hartmut Ehrig (2008). “Behavior-Preserving Simulation-to-Animation Model and Rule Transformations”. In: *Electronic Notes in Theoretical Computer Science* 213.1, pp. 55–74 (cit. on pp. 56, 71).

Fabro, Marcos Didonet Del and Patrick Valduriez (2009). “Towards the efficient development of model transformations using model weaving and matching transformations”. In: *Software and System Modeling* 8.3, pp. 305–324 (cit. on p. 57).

Farzan, Azadeh, Feng Chen, José Meseguer, and Grigore Rosu (2004). “Formal Analysis of Java Programs in JavaFAN”. In: *Proceedings of Computer-aided Verification (*Cav*)*. Ed. by Rajeev Alur and Doron Peled. Lecture Notes in Computer Science 3114, pp. 501–505 (cit. on p. 177).

Fischer, Thorsten, Jörg Niere, Lars Turunski, and Albert Zündorf (2000). “Story diagrams: A new graph rewrite language based on the Unified Modelling Language and Java”. In: *Theory and Application of Graph Transformations*. Ed. by Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg. Vol. 1764. LNCS. Paderborn: Springer-Verlag, pp. 296–309 (cit. on p. 55).

Fleurey, Frank (2006). “Language and Method for Trustable Modeling Engineering”. (in french). PhD thesis. University of Rennes (France) (cit. on pp. 127, 132).

Fowler, Martin (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, p. 464 (cit. on p. 57).

Gabmeyer, Sebastian, Petra Brosch, and Martina Seidl (2013). “A Classification of Model Checking-Based Verification Approaches for Software Models”. In: *Proceedings of the 2nd International Workshop on the Verification of Model Transformation (VOLT)* (cit. on p. 86).

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. isbn: 0-201-63361-2 (cit. on pp. 19, 50).

Gardner, Tracy, Catherine Griffin, Jana Koehler, and Rainer Hauser (2003). “A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final standard”. In: *Proceedings of the MetaModelling for MDA Workshop*, pp. 178–197 (cit. on p. 62).

Gargantini, Angelo, Elvinia Riccobene, and Patrizia Scandurra (2008). “Model-Driven Language Engineering: The ASMETA Case Study”. In: *Proceedings of the Third International Conference on Software Engineering Advances (ICSEA ’08)*. Washington, DC, USA: IEEE Computer Society, pp. 373–378. ISBN: 978-0-7695-3372-8. DOI: http://dx.doi.org/10.1109/ICSEA.2008.62 (cit. on p. 114).

— (2009). “A Semantic Framework for Metamodel-Based Languages”. In: Ase 16.3–4, pp. 415–454 (cit. on pp. 28, 114).

— (2010). “Combining Formal Methods and Mde Techniques for Model-Driven System Design and Analysis”. In: JaS 3.1–2, pp. 1–18 (cit. on pp. 41, 70).

Gessenharter, D (2008). “Mapping the UML2 Semantics of Associations to a Java Code Generation Model”. In: *International Conference on Model Driven Engineering Languages and Systems*. Ed. by K. et al. Czarnecki. Vol. 5301. Lecture Notes in Computer Science. Springer-Verlag, pp. 813–827 (cit. on p. 56).

Ghosh, Sudipto, ed. (2010). *Models in Software Engineering, Workshops and Symposia at MODELS 2009, Denver, CO, USA, October 4-9, 2009, Reports and Revised Selected Papers*. Vol. 6002. Lecture Notes in Computer Science. Springer. ISBN: 978-3-642-12260-6.

Giese, Holger, Sabine Glesner, Johannes Leitner, Wilhelm Schäfer, and Robert Wagner (2006). “Towards Verified Model Transformations”. In: MoDeVVa, pp. 78–93 (cit. on pp. 44, 46).

Giese, Holger, Tihamer Levendovszky, and Hans Vangheluwe (2007). “Summary of the Workshop on Multi-Paradigm Modeling: Concepts and Tools”. In: *Models in Software Engineering: Workshops and Symposia at MoDELS 2006, Reports and Revised Selected Papers*. Vol. 4364. Lncs. Springer (cit. on pp. 55, 65).

Gogolla, Martin and Antonio Vallecillo (2011). “*Tract*able Model Transformation Testing”. In: *Proceedings of the 7th European Conference on Modelling Foundations and Applications (ECMFA)*. Vol. 6698. Lncs. Springer, pp. 221–235 (cit. on p. 78).

Graf, Susanne, Bernhard Steffen, and Gerald Lüttgen (1996). “Compositional Minimisation of Finite-State Systems Using Interface Specifications”. In: *Formal Aspects of Computing* 8.5, pp. 607–616 (cit. on p. 84).

Griswold, William G. (1991). “Program Restructuring as an Aid to Software Maintenance”. PhD thesis. University of Washington (cit. on p. 57).

Grønmo, Roy, Ragnhild Runde, and Birger Møller-Pedersen (2011). “Confluence of Aspects For Sequence Diagrams”. In: SoSyM, pp. 1–36 (cit. on pp. 38, 42).

Guerra, E., J. De Lara, M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger (2013). “Automated Verification of Model Transformations based on Visual Contracts”. In: *Automated Software Engineering* 20.1, pp. 5–46 (cit. on pp. 78, 174).

Guerra, Esther and Juan de Lara (2006). “Model View Management with Triple Graph Transformation Systems”. In: *International Conference on Graph Transformation*. Vol. 4178. LNCS. Springer-Verlag, pp. 351–366 (cit. on p. 57).

— (2007). “Event-Driven Grammars: Relating Abstract and Concrete Levels of Visual Languages”. In: *Journal on Software and Systems Modeling* 6.6, pp. 317–347 (cit. on p. 56).

Guy, Clément (2013). “Facilités de Typage pour l’Ingénierie des Modèles”. PhD thesis. University of Rennes (France) (cit. on p. 175).

Hardebolle, Cécile (2008). “Composition de Modèles pour la Modélisation Multi-Paradigme du Comportement des Systèmes”. PhD thesis. Université Paris-Sud XI — École Supérieure d’Électricité (cit. on p. 177).

Harel, David and Bernhard Rumpe (2000). *Modeling Languages: Syntax, Semantics and All That Stuff, Part I: The Basic Stuff.* Tech. rep. Weizmann Institute Of Sience (cit. on pp. 55, 57).

— (2004). "Meaningful Modeling: What's the Semantics of "Semantics"?" In: *Computer* 37.10, pp. 64–72. ISSN: 0018-9162. DOI: http://doi.ieeecomputersociety.org/10.1109/MC.2004.172 (cit. on p. 27).

Harman, Mark, David Binkley, and Sebastian Danicic (1997). "Amorphous Program Slicing". In: *Software Focus*. IEEE Computer Society Press, pp. 70–79 (cit. on p. 62).

Heckel, Reiko, Jochen M. Küster, and Gabriele Taentzer (2002). "Confluence of Typed Attributed Graph Transformation Systems". In: ICGT (cit. on p. 38).

Iacob, Maria-Eugenia, Maarten W. A. Steen, and Lex Heerink (2008). "Reusable Model Transformation Patterns". In: *Proceedings of EDOCW'08*, pp. 1–10 (cit. on pp. 54, 85).

Izquierdo, Javier L.C and Jesús García Molina (2012). "Extracting Models from Source Code in Software Modernization". In: *Software & Systems (*SoSyM*)*, pp. 1–22 (cit. on p. 66).

Jackson, Daniel (2011). *Software Abstractions* (cit. on pp. 2, 34).

Jia, Yue and Mark Harman (2010). "An Analysis and Survey of the Development of Mutation Testing". In: IEEE *Transactions of Software Engineering* 37 (5), pp. 649–678 (cit. on p. 2).

Jurack, Stefan and Gabriele Taentzer (2010). "A Component Concept for Typed Graphs With Inheritance and Containment Structures". In: ICGT (cit. on pp. 19, 114).

Kabore, Eveline Chantal (2008). "Contribution à l'Automatisation d'un Processus de Construction d'Abstractions de Communication par Transformations Successives de Modèles". PhD thesis. École Nationale Supérieure des Télécommunications de Bretagne / Université de Rennes (cit. on p. 2).

Kastenberg, Harmen and Arend Rensink (2006). "Model Checking Dynamic States in GROOVE". In: *Model Checking Software (*SPIN*)*. Vol. 3925. Lecture Notes on Computer Science, pp. 299–305 (cit. on p. 70).

Katz, Shmuel (2006). "Aspect Categories and Classes of Temporal Properties". In: TAOSD 3880, pp. 106–134 (cit. on pp. 40, 42).

Kelly, Steven and Juha-Pekka Tolvanen (2008). *Domain-Specific Modeling: Enabling Full Code Generation.* Wiley-IEEE Computer Society. URL: http://www.worldcat.org/isbn/0470036664 (cit. on pp. 19, 26).

Kelsen, Pierre, Qin Ma, and Christian Glodt (2011). "Models within Models: Taming Model Complexity Using the Sub-model Lattice". In: *Proceedings of the International Conference on the Foundational Approaches to Software Engineering (FASE)*. Springer, pp. 171–185 (cit. on p. 63).

Kern, Heiko (2009). "The Interchange of (Meta)Models Between MetaEdit+and Eclipse EMF Using M3-Level-Based Bridges". In: *Workshop on Domain-Specific Modeling (*DSM*)* (cit. on p. 66).

Kern, Heiko, Axel Hummel, and Stefan Kühne (2011). "Towards a Comparative Analysis of Meta-Metamodels". In: *Workshop on Domain-Specific Modeling (*DSM*)* (cit. on p. 66).

Kern, Heiko and Stefan Kühne (2007). "Model Interchange between ARIS and Eclipse EMF". In: *Workshop on Domain-Specific Modeling (*DSM*)* (cit. on p. 66).

— (2009). "Integration of Microsoft Visio and Eclipse Modeling Framework Using M3-Level-Based Bridges". In: *ECMDA Workshop on Model-Driven Tool & Process Integration* (cit. on pp. 66, 68).

Kilov, Haim (1990). "From semantic to object-oriented data modeling". In: *First International Conference on System Integration*, pp. 385–393 (cit. on p. 54).

Kleppe, Anneke (2009). *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Upper Saddle River, NJ: Addison-Wesley (cit. on p. 28).

Kleppe, Anneke G., Jos Warmer, and Wim Bast (2003). *MDA Explained: The Model Driven Architecture: Practice and Promise*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 032119442X (cit. on pp. 17, 25, 55, 64, 65).

Knuth, Donald E. (1992). *Literate Programming*. University of Chicago Press (cit. on p. 150).

Koch, Nora, Alexander Knapp, Gefei Zhang, and Hubert Baumeister (2008). "UML-based web engineering". In: *Web Engineering: Modelling and Implementing Web Applications*, pp. 157–191 (cit. on p. 62).

Kolovos, Dimitrios, Louis Rose, Antonio García-Domínguez, and Richard Paige (2012). *The Epsilon Book*. The Eclipse Foundation. URL: http://www.eclipse.org/epsilon (cit. on pp. 19, 71, 132).

König, Barbara and Vitali Kozioura (2008). "Augur 2: A New Version of a Tool for the Analysis of Graph Transformation Systems". In: *Electronic Notes in Theoretical Computer Science (ENTCS)* 211, pp. 201–210 (cit. on p. 67).

Krishnan, P. (2000). "Consistency Checks For UML". In: *Proceedings of the Seventh Asia-Pacific Software Engineering Conference (APSEC)* (cit. on p. 114).

Kuhlmann, Mirco, Lars Hamann, Martin Gogolla, and Fabian Büttner (2013). "A Benchmark for OCL Engine Accuracy, Determinateness and Efficiency". In: *Journal of Software and Systems (SoSyM)* (cit. on p. 148).

Kühne, Thomas (2006). "Matters of (Meta-) Modeling". In: *Software and Systems Modeling (SoSyM)* 5 (4), pp. 369–385. ISSN: 1619-1366. URL: http://dx.doi.org/10.1007/s10270-006-0017-9 (cit. on pp. 12, 13, 25, 84).

Kühne, Thomas, Gergely Mezei, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer (2009). "Systematic Transformation Development". In: ECEASST 21 (cit. on pp. 39, 42, 67).

— (2010). "Explicit Transformation Modeling". In: *MODELS 2009 Workshops*. Ed. by Sudipto Ghosh. Vol. 6002. LNCS. Denver: Springer, pp. 240–255 (cit. on pp. 56, 80).

Küster, Jochen Malte (2006). "Definition and Validation of Model Transformations". In: SoSyM **5**(3), pp. 233–259 (cit. on pp. 38, 42, 43, 46).

Lambers, Leen, Hartmut Ehrig, and Fernando Orejas (2006). "Efficient Detection of Conflicts in Graph-based Model Transformation". In: ENTCS 152 (cit. on pp. 38, 42).

Lano, Kevin and Shekoufeh Kolahdouz Rahimi (2011). "Slicing Techniques for UML Models". In: *Journal of Technology* 10, pp. 1–49 (cit. on p. 62).

Levendovszky, Tihamér, László Lengyel, and Tamás Mészáros (2009). "Supporting Domain-Specific Model Patterns With Metamodeling". In: SoSyM **8**(4) (cit. on pp. 39, 42).

Lions, Jacques-Louis (1999). *Ariane 5 Flight 501 Failure — Inquiry Board Report*. Tech. rep. European Spatial Agency (cit. on p. 2).

Lúcio, Levi, Bruno Barroca, and Vasco Amaral (2010). "A Technique for Automatic Validation of Model Transformations". In: MoDELS (cit. on pp. 40, 42, 44, 46).

Lúcio, Levi, Joachim, Sadaf Mustafiz, and Hans Vangheluwe (2012). *An Overview of Model Transformations for a Simple Automotive Power Window*. Tech. rep. SOCS-TR-2012.1. McGill University (cit. on p. 75).

Lúcio, Levi, Sadaf Mustafiz, Joachim Denil, Hans Vangheluwe, and Maris Jukss (2013). “FTG+PM: An Integrated Framework for Investigating Model Transformation Chains”. In: *System Design Languages (SDL) Forum: Model-Driven Dependability Engineering.* Vol. 7916. LNCS. Springer, pp. 182–202 (cit. on p. 81).

Lúcio, Levi, Zhang Qin, Phu H. Nguyen, Moussa Amrani, Jacques Klein, Hans Vangheluwe, and Yves Le Traon (2014). “Advances in Model-Driven Security”. In: *Advances in Computer Science* (cit. on p. 197).

Lúcio, Levi, Eugene Syriani, Moussa Amrani, Qin Zhang, and Hans Vangheluwe (2012). “Invariant Preservation In Iterative Modeling”. In: *Workshop on Models and Evolution (ME)* (cit. on p. 198).

Lúcio, Levi and Hans Vangheluwe (2013a). “Model Transformations to Verify Model Transformations”. In: *Proceedings of the 2nd Workshop on Verification of Model Transformations (VOLT)* (cit. on pp. 55, 69, 71).

Lúcio, Lévi and Hans Vangheluwe (2013b). *Symbolic Execution for the Verification of Model Transformations.* Tech. rep. SOCS-TR-2013.2. http://msdl.cs.mcgill.ca/people/levi/files/MTSymbExec.pdf. Montréal, Canada: McGill University (cit. on pp. 78, 79).

Mannadiar, Raphael and Hans Vangheluwe (2010). “Modular Synthesis of Mobile Device Applications from Domain-Specific Models”. In: *Model-based Methodologies for Pervasive and Embedded Software workshop* (cit. on pp. 55, 64–66).

Martí-Oliet, Narcisso, Miguel Palomino, and Alberto Verdejo (2005). “A Tutorial on Specifying Data Structures in Maude”. In: *Electronic Notes in Theoretical Computer Science* 137, pp. 105–132 (cit. on p. 163).

Massoni, Tiago, Rohit Gheyi, and Paulo Borba (2005). “Formal Refactoring for UML Class Diagrams”. In: BSSE, pp. 152–167 (cit. on pp. 41–43, 46).

Mayerhofer, Tanja, Philip Langer, and Manuel Wimmer (2012). “Towards xMOF: Executable DSMLs Based on fUML”. In: *Proceedings of the 12th Workshop on Domain-Specific Modeling (DSM’12)* (cit. on p. 71).

Mc Brien, Peter and Alexandra Poulovassi (1999). “Automatic Migration and Wrapping of Database Applications - A Schema Transformation Approach”. In: *Conceptual Modeling ER’99.* Ed. by Jacky Akoka, Mokrane Bouzeghoub, Isabelle Comyn-Wattiau, and Elisabeth M’etais. Vol. 1782. LNCS. London: Springer-Verlag, pp. 99–114 (cit. on p. 56).

Mens, Tom and Pieter Van Gorp (2006). “A Taxonomy Of Model Transformation”. In: *Electronic Notes in Theoretical Computer Science (ENTCS)* 152, pp. 125–142 (cit. on pp. 17, 21, 37, 49, 54, 85).

Mernik, Marjan, ed. (2013). *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments.* IGI Global. 677 pp.

Mokhati, Farid and Mourad Badri (2009). “Generating Maude Specifications from UML Use Case Diagrams”. In: *Journal of Object Technology* 8.2, pp. 119–136 (cit. on pp. 147, 148).

Molderez, Tim, Hans Schippers, Dirk Janssens, Haupt Michael, and Robert Hirschfeld (2010). “A Platform for Experimenting with Language Constructs for Modularizing Crosscutting Concerns”. In: WASDETT (cit. on pp. 40, 42).

Mosterman, Pieter J. and Hans Vangheluwe (2004). “Computer Automated Multi-Paradigm Modeling: An Introduction”. In: *Simulation: Transactions of The Society for Modeling and Simulation International* 80.9, pp. 433–450 (cit. on pp. 75, 177).

Muller, Pierre-Alain, Franck Fleurey, and Jean-Marc Jézéquel (2005). “Weaving Executability into Object-Oriented Meta-Languages”. In: *Proceedings of MODELS/UML’2005.* Vol. 3713. LNCS. Montego Bay, Jamaica: Springer, pp. 264–278 (cit. on pp. 71, 91, 115).

Muller, Pierre-Alain, Frédéric Fondement, Benoît Baudry, and Benoît Combemale (2010). "Modeling modeling modeling". In: *Software and Systems Modeling* 11, pp. 1–13 (cit. on pp. 84, 85).

Muller, Pierre-Alain and Michel Hassenforder (2005). "HUTN as a Bridge between ModelWare and GrammarWare — An Experience Report". In: *Workshop in Software Model Engineering* WISME *(Satellite Event of MoDELS 2005) (co-located with MoDELS)* (cit. on p. 66).

Mustafiz, Sadaf, Joachim Denil, Levi Lúcio, and Hans Vangheluwe (2012). "The FTG+PM Framework for Multi-Paradigm Modelling: An Automotive Case Study". In: *Proceedings of the MPM (Multi-Paradigm Modelling) 2012 Workshop, associated with MoDELS*. ACM Digital Library (cit. on p. 73).

Narayanan, Anantha and Gabor Karsai (2008a). "Towards Verifying Model Transformations". In: *Electronic Notes in Theoretical Computer Science* 211, pp. 191–200. ISSN: 1571-0661 (cit. on pp. 41, 42, 44, 46, 67, 68).

— (2008b). "Verifying Model Transformation By Structural Correspondence". In: ECEASST 10, pp. 15–29 (cit. on pp. 40, 42, 44, 46, 69).

Naumovich, Gleb and Lori Clarke (2000). "Classifying Properties: An Alternative to the Safety-Liveness Classification". In: *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pp. 159–168 (cit. on p. 61).

Newman, Maxwell Herman Alexander (1942). "On Theories With a Combinatorial Definition of "Equivalence"". In: *Annals of Mathematics* **43**(2), pp. 223–243 (cit. on p. 38).

Nipkow, Tobias, Lawrence C. Paulson, and Markus Wenzel (2013). *Isabelle/*HOL*: A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Springer-Verlag (cit. on p. 34).

Object Management Group (2003). *MDA Guide (version 1.0.1)*. Tech. rep. Object Management Group (cit. on pp. 15, 17, 25).

— (2004). *Unified Modeling Language (*UML*) — Infrastructure Specification (Version 2)*. Tech. rep. Object Management Group (cit. on pp. 15, 25).

— (2006). *Meta-Object Facility 2.0 Core Specification (06-01-01)*. Tech. rep. Object Management Group (cit. on pp. 11, 14, 25, 91, 93, 101, 102, 104).

— (2008). MOF QVT*: Query / View / Transformation*. Tech. rep. Object Management Group (cit. on pp. 20, 67).

— (2010). *Object Constraint Language (*OCL*) Specification (Version 2.2, formal/2010-02-01)*. Tech. rep. Object Management Group (cit. on pp. 61, 148).

— (2011a). *OMG / Unified Modeling Language (*UML*). Superstructure / Infrastructure*. Tech. rep. formal-2011-08-06. Object Management Group (cit. on pp. 11, 52, 73, 148).

— (2011b). UML *Profile for Modeling and Analysis of Real-Time Embedded System (*MARTE*). (formal/2011-06-02)*. Tech. rep. Object Management Group (cit. on p. 174).

Ölveczky, Peter Csaba and José Meseguer (2007). "Semantics and Pragmatics of Real-Time Maude". In: *Higher-Order and Symbolic Computation* 20.1–2. Ed. by Springer (cit. on p. 177).

Padberg, Julia (1999). "Categorical Approach to Horizontal Structuring and Refinement of High-Level Replacement Systems". English. In: *Applied Categorical Structures* 7 (4), pp. 371–403. ISSN: 0927-2852. DOI: 10.1023/A:1008695316594 (cit. on pp. 64, 65).

Padberg, Julia, Magdalena Gajewsky, and Claudia Ermel (1997). *Refinement* versus *Verification: Compatibility of Net Invariants and Stepwise Development of High-Level Petri Nets*. Tech. rep. Technische Universität Berlin (cit. on pp. 41–43, 46).

Paige, Richard F., Phillip J. Brooke, and Jonathan S. Ostroff (2007). “Metamodel-Based Model Conformance and Multi-View Consistency Checking”. In: ACM TOSEM 16.3, pp. 1–48 (cit. on pp. 44, 46).

Parnas, David Lorge (1977). “The Use of Precise Specification in the Development of Software”. In: *International Federation for Information Processing World Congress (IFIP)*. Ed. by Bruce Gilchrist, pp. 861–867 (cit. on p. 24).

Partsch, H. and R. Steingbrüggen (1983). “Program Transformation Systems”. In: *Computing Surveys* 15.3, pp. 199–236 (cit. on p. 20).

Peterson, J. (1977). “Petri Nets”. In: *ACM Comput. Surv.* 9.3, pp. 223–252. ISSN: 0360-0300. DOI: 10.1145/356698.356702. URL: http://doi.acm.org/10.1145/356698.356702 (cit. on p. 75).

Plump, Detlef (1998). “Termination of Graph Rewriting is Undecidable”. In: *Fundamenta Informaticæ* 33.2, pp. 201–209 (cit. on p. 38).

— (2005). “Confluence of Graph Transformation Revisited”. In: *Processes, Terms and Cycles: Steps on the Road to Infinity*. Vol. 3838 (cit. on p. 38).

Poernomo, Iman (2006). “The Meta-Object Facility (MOF) Typed”. In: *SAC*, pp. 1845–1849 (cit. on p. 114).

Pollet, Isabelle (2004). “Towards a Generic Framework For the Abstract Interpretation of Java”. PhD thesis. Catholic University of Louvain (Belgium) (cit. on pp. 118, 119).

Pretschner, Alexander (2005). “Model-Based Testing”. In: *International Conference on Software Engineering (*ICSE*)*. Ed. by Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh (cit. on p. 2).

Quesada Moreno, José Francisco (1997). “The SCP Parsing Algorithm Based on Syntactic Constraint Propagation”. PhD thesis. University of Sevilla (cit. on p. 160).

— (1999). *The* SCP *Parsing Algorithm*. Tech. rep. SRI International, Computer Science Laboratory (cit. on p. 160).

Rahim, Lukman Ab. and Jon Whittle (2013). “A survey of approaches for verifying model transformations”. In: *Software and Systems Modeling (SoSym)*, pp. 1–26 (cit. on pp. 48, 86).

Ráth, István, András Ökrös, and Dániel Varró (2010). “Synchronization of Abstract and Concrete Syntax in Domain-Specific Modeling Languages By Mapping Models and Live Transformations”. In: *Journal of Software Systems and Models* 9, pp. 453–471 (cit. on p. 27).

Reggio, Gianna, Maura Cerioli, and Egidio Astesiano (2001). “Towards A Rigorous Semantics Of UML Supporting Its Multiview Approach”. In: *Conference on Fundamental Approaches to Software Engineering (*FASE*)* (cit. on p. 114).

Rensink, Arend (2003). “The GROOVE Simulator: A Tool for State Space Generation”. In: *Applications of Graph Transformations with Industrial Relevance (*AGTIVE*)*. Vol. 3062. Lecture Notes in Computer Science, pp. 479–485 (cit. on p. 70).

Rensink, Arend, Àkos Schmidt, and Dániel Varró (2004). “Model Checking Graph Transformations: A Comparison of Two Approaches”. In: ICGT (cit. on pp. 44, 46).

Rivera, José E., Francisco Durán, and Antonio Vallecillo (2008). *A Metamodel For Maude*. Tech. rep. University of Málaga (Spain) (cit. on pp. 170, 176).

— (2009). “Formal Specification and Analysis of Domain-Specific Models Using Maude”. In: *Simulation: Transactions of the Society for Modeling and Simulation International* 85.11/12, pp. 778–792 (cit. on pp. 41, 70, 147).

Rivera, José Eduardo, Francisco Durán, and Antonio Vallecillo (2010). “On the Behavioral Semantics of Real-Time Domain Specific Visual Languages”. In: *8th International Workshop on Rewriting Logic and its Applications (*WRLA*)*, pp. 174–190 (cit. on pp. 29, 67).

Rivera, José Eduardo, Esther Guerra, Juan de Lara, and Antonio Vallecillo (2009). “Analyzing Rule-Based Behavioral Semantics of Visual Modeling Languages with Maude”. In: *Proceeding of the International Conference on Software Language Engineering (*SLE*)*. Ed. by Dragan Gašević, Ralf Lämmel, and Eric Wyk. Vol. 5452. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 54–73 (cit. on p. 47).

Rivera, José Eduardo and Antonio Vallecillo (2007). “Adding Behavioral Semantics to Models”. In: *11th IEEE International Enterprise Distributed Object Computing Conference (*EDOC*)*, pp. 169–182 (cit. on p. 28).

Rozenberg, Grzegorz, ed. (1997). *Handbook of Graph Grammars and Computing by Graph Transformation.* Vol. I (Foundations). River Edge, NJ, USA: World Scientific Publishing Co., Inc. ISBN: 98-102288-48 (cit. on pp. 19, 20, 114).

Rusu, Vlad (2011). “Embedding Domain-Specific Modelling Languages in Maude Specifications”. In: *ACM SIGSOFT Software Engineering Notes* 36.1, pp. 1–8 (cit. on p. 147).

Sánchez Cuadrado, Jesús, Esther Guerra, and Juan de Lara (2011). “Generic Model Transformations: *Write Once, Reuse Everywhere*”. In: *International Conference on Theory and Practice of Model Transformations (*ICMT*)*. 6707 vols. Lecture Notes in Computer Science. Springer (cit. on p. 4).

Schatz, B., F. Holzl, and T. Lundkvist (2010). “Design-Space Exploration through Constraint-Based Model-Transformation”. In: *International Conference and Workshops on Engineering of Computer Based Systems.* ECBS’10, pp. 173–182 (cit. on p. 56).

Scheidgen, Markus and Joachim Fischer (2007). “Human Comprehensible and Machine Processable Specifications of operational Semantics”. In: *Proceeding of the European Conference on Model-Driven Architecture Foundations and Applications*, pp. 157–171 (cit. on p. 71).

Schmidt, Douglas C. (2006). “Model-Driven Engineering”. In: IEEE *Computer* 39, pp. 25–31 (cit. on p. 1).

Scholz, Peter (1998). “A Refinement Calculus for Statecharts”. In: *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*. Ed. by Egidio Astesiano. Vol. 1382. LNCS. Springer, pp. 285–301 (cit. on p. 64).

Schürr, Andy and Felix Klar (2008). “15 Years of Triple Graph Grammars”. In: ICGT, pp. 411–425 (cit. on pp. 40, 42).

Selic, Bran, Garth Gullekson, and Paul T. Ward (1995). *Real-Time Object-Oriented Modeling.* John Wiley & Sons Canada. 560 pp. (cit. on p. 177).

Selim, Gehan M.K., James R. Cordy, and Juergen Dingel (2012a). *Analysis of Model Transformations.* Tech. rep. 2012-592. Queen’s University, pp. 1–58 (cit. on p. 174).

Selim, Gehan M.K., James R. Cordy, and Jürgen Dingel (2012b). “Model Transformation Testing: The State of the Art”. In: *Workshop on Analysis of Model Transformations (*AMT*)* (cit. on pp. 2, 174).

Sen, Sagar, Naouel Moha, Benoit Baudry, and Jean-Marc Jézéquel (2009). “Metamodel Pruning”. In: *Models in Software Engineering, Workshops and Symposia at MODELS 2009, Denver, CO, USA, October 4-9, 2009, Reports and Revised Selected Papers*. Ed. by Sudipto Ghosh. Vol. 6002. Lecture Notes in Computer Science. Springer, pp. 32–46. ISBN: 978-3-642-12260-6 (cit. on p. 62).

Sendall, Shane and Wojtek Kozaczynski (2003). “Model Transformation: The Heart And Soul Of Model-Driven Software Development”. In: IEEE *Software* 20.5, pp. 42–45 (cit. on p. 49).

Shannon, Robert and James D. Johannes (1976). “Systems Simulation: The Art and Science”. In: *IEEE Transactions on Systems, Man and Cybernetics* SMC-6.10, pp. 723–724 (cit. on p. 71).

Simos, Mark A. (1995). “Organization Domain Modeling (ODM): Formalizing the Core Domain Modeling Life Cycle”. In: *SIGSOFT Software Engineering Notes* 20 (SI), pp. 196–205. DOI: http://proxy.bnl.lu:2259/10.1145/223427.211845 (cit. on p. 23).

Society, IEEE Computer (2004). *Software Engineering Body of Knowledge*. Ed. by Alain Abran, James W. Moore, Pierre Bourque, and Robert Dupuis (cit. on p. 2).

Song, Dan, Keqing He, Peng Liang, and Wudong Liu (2005). “A Formal Language For Model Transformation Specification”. In: *In Proceedings of the Seventh International Conference on Enterprise Information Systems (*ICEIS*)* (cit. on p. 114).

Soon-Kyeong, Kim and David Carrington (1999). “Formalizing the UML Class Diagram Using Object-Z”. In: *Second International Conference on* UML *’99* (cit. on p. 114).

Spivey, J. Michael (1992). *The Z Notation - A Reference Manual (2nd. Ed.)* Prentice Hall (cit. on pp. 99, 198).

Spoto, Fausto, Patricia M. Hill, and Étienne Payet (2006). “Path-Length Analysis of Object-Oriented Programs”. In: EAAI (cit. on pp. 38, 42).

Stachowiak, Herbert (1973). *Allgemeine Modelltheorie [General Model Theory]*. Springer (cit. on p. 11).

Stahl, Thomas, Markus Voelter, and Krzysztof Czarnecki (2006). *Model-Driven Software Development – Technology, Engineering, Management*. John Wiley & Sons (cit. on p. 55).

Stark, Robert F., E. Borger, and Joachim Schmid (2001). *Java and the Java Virtual Machine: Definition, Verification, Validation*. Secaucus, NJ, USA: Springer-Verlag New York, Inc. ISBN: 3540420886 (cit. on pp. 118, 119).

Steel, James R.H (Jim) (2007). “Model Typing”. PhD thesis. Université de Rennes I. URL: ftp://ftp.irisa.fr/techreports/theses/2007/steel.pdf (cit. on p. 175).

Steel, Jim and Jean-Marc Jézéquel (2007). “On Model Typing”. In: SoSyM **6**(4) (cit. on pp. 39, 175).

Steinberg, David, Frank Budinsky, Marcelo Paternostro, and Ed Merks (2009). *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional. ISBN: 0321331885 (cit. on pp. 106, 149).

Stenzel, Kurt, Nina Moebius, and Wolfgang Reif (2011). “Formal Verification of QVT Transformations for Code Generation”. In: MoDELS. Wellington, New Zealand (cit. on p. 46).

Stephenson Arthur, G., Daniel R. Mulville, Frank H. Bauer, Greg A. Dukeman, Peter Norvig, Lia S. LaPiana, Peter J. Rutledge, David Folta, and Robert Sackheim (1999). *Mars Climate Orbiter – Mishap Investigation Board. Phase I Report*. Tech. rep. National Aeronautics and Space Administration (cit. on p. 2).

Syriani, Eugene (2011). “A Multi-Paradigm Foundation for Model Transformation Language Engineering”. PhD thesis. McGill University (cit. on pp. 17, 18, 20, 54).

Syriani, Eugene and Hüseyin Ergin (2012). "Operational Semantics of UML Activity Diagram: An Application in Project Management". In: *RE 2012 Workshops*. Chicago: IEEE (cit. on p. 55).

Syriani, Eugene and Hans Vangheluwe (2008). "Programmed Graph Rewriting with Time for Simulation-Based Design". In: *Proceedings of the First International Conference on Theory and Practice of Model Transformations (ICMT)*. Vol. 5063. Lncs. Springer, pp. 91–106 (cit. on p. 67).

— (2010a). "De-/Re-constructing Model Transformation Languages". In: *Electronic Communication of the European Association of Software Science and Technology (*EcEasst*)* 29, pp. 1–14 (cit. on p. 4).

— (2010b). "DEVS as a Semantic Domain for Programmed Graph Transformation". In: *Discrete-Event Modeling and Simulation: Theory and Applications*. Boca Raton: CRC Press. Chap. 1, pp. 3–28. isbn: 9781420072334 (cit. on p. 67).

— (2011). "A Modular Timed Model Transformation Language". In: *Journal on Software and Systems Modeling* 11, pp. 1–28 (cit. on p. 80).

Taentzer, Gabriele (2000). "Agg: A Tool Environment for Algebraic Graph Transformation". In: Agtive. Vol. 1779, pp. 333–341 (cit. on pp. 19, 40).

Thirioux, Xavier, Benoit Combemale, Xavier Crégut, and Pierre-Loïc Garoche (2007). "A Framework to Formalise the MDE Foundations". anglais. In: *International Workshop on Towers of Models (TOWERS)*. Ed. by Richard Paige and Jean Bézivin. Zurich, pp. 14–30 (cit. on p. 15).

Tisi, Massimo, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin (2009). "On the Use of Higher-Order Model Transformations". In: *Model Driven Architecture - Foundations and Applications (*Mda-Fa*)*. Vol. 5562. Lecture Notes in Computer Science. Springer (cit. on pp. 3, 19, 54, 85).

Torlak, Emina and Daniel Jackson (2007). "Kodkod: A Relational Model Finder". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Orna Grumberg and Michael Huth. Vol. 4424. LNCS. Springer, pp. 632–647 (cit. on p. 56).

Tratt, Laurence (2005). "Model Transformation And Tool Integration". In: SoSyM **4**(2), pp. 112–122 (cit. on p. 17).

Tri, D.Q. and Q.T. Tho (2012). "Systematic Diagram Refinement for Code Generation in SEAM". In: *Knowledge and Systems Engineering (KSE), 2012 Fourth International Conference on*. IEEE, pp. 203–210 (cit. on pp. 55, 64–66).

Troya, Javier, José E. Rivera, and Antonio Vallecillo (2009). "On the Specification of Non-functional Properties of Systems by Observation". In: *MoDELS Workshops*. Ed. by Sudipto Ghosh. Vol. 6002. Lecture Notes in Computer Science. Springer, pp. 296–309. isbn: 978-3-642-12260-6 (cit. on p. 67).

Troya, Javier, Antonio Vallecillo, Francisco Durán, and Steffen Zschaler (2013). "Model-Driven Performance Analysis of Rule-Based Domain Specific Visual Models". In: *Information & Software Technology* 55.1, pp. 88–110 (cit. on p. 67).

Vallecillo, Antonio and Martin Gogolla (2012). "Typing Model Transformations Using Tracts". In: *5th International Conference on the Theory and Practice of Model Transformations (ICMT'12)*. Vol. LNCS 7307. Springer, pp. 56–71 (cit. on pp. 78, 174).

Van der Straeten, Ragnhild, Viviane Jonckers, and Tom Mens (2007). "A Formal Approach to Model Refactoring and Model Refinement". In: *Journal of Software and Systems Modeling* 6 (2), pp. 139–162. issn: 1619-1366. doi: `10.1007/s10270-006-0025-9` (cit. on p. 64).

Varró, Dániel and András Balogh (2007). "The Model Transformation Language of the VIATRA2 Framework". In: *Science of Computer Programming* 68.3, pp. 214–234 (cit. on p. 20).

Varró, Dániel and András Pataricza (2003). "Automated Formal Verification of Model Transformations". In: CSDUml, pp. 63–78 (cit. on pp. 41, 42, 44, 46, 61).

Varró, Dániel, Szilvia Varró-Gyapai, Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer (2006). "Termination Analysis of Model Transformations by Petri Nets". In: Icgt. Vol. 4178, pp. 260–274 (cit. on pp. 38, 42, 43, 46, 67).

Viehstaedt, Gerhard and Mark Minas (1995). "DiaGen: A Generator for Diagram Editors Based on a Hypergraph Model". In: *International Workshop on Next Generation Information Technologies and Systems*. Ed. by Amihai Motro and Moshe Tennenholtz. Naharia, pp. 155–162 (cit. on p. 56).

Visser, Eelco (2005). "A Survey of Strategies in Rule-Based Program Transformation Systems". In: *J. Symbolic Computation* **40**(1), pp. 831–873. issn: 0747-7171. doi: 10.1016/j.jsc.2004.12.011 (cit. on pp. 19, 20, 85).

Website, AUTOSAR. http://www.autosar.org (cit. on p. 83).

Weiss, David M. and Chi Tau Robert Lai (1999). *Software Product Line Engineering: A Family-Based Software Development Process*. Addison-Wesley Longman Publishing (cit. on p. 31).

Wimmer, Manuel, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, and Wieland Schwinger (2009). "Right or Wrong? Verification of Model Transformations using Colored Petri Nets". In: *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM'09)*. Helsinki Business School (cit. on p. 81).

Wimmer, Manuel and Gerhard Kramler (2005). "Bridging Grammarware and Modelware". In: *Proceedings of* MoDELS *Satellite Events*, pp. 159–168 (cit. on pp. 16, 66).

Winkelmann, Jessica, Gabriele Taentzer, Karsten Ehrig, and Jochen M. Küster (2008). "Translation of Restricted OCL Constraints into Graph Constraints for Generating Meta Model Instances by Graph Grammars". In: *Electronic Notes in Theoretical Computer Science* 211, pp. 159–170 (cit. on p. 56).

Winskel, Glynn (1993). *The Formal Semantics of Programming Languages: An Introduction (Foundations of Computing)*. Cambridge, MA (USA): MIT Press, p. 384 (cit. on pp. 115, 124, 141).

Wirth, Niklaus (1971). "The Programming Language Pascal". In: *Acta Informatica* 1, pp. 35–63 (cit. on p. 144).

Withall, Stephen (2007). *Software Requirement Patterns*. Microsoft Press (cit. on p. 85).

Yang, M., G.J. Michaelson, and R.J. Pooley (2008). "Formal Action Semantics for a UML Action Language". In: *Journal of Universal Computer Science* 14.21, pp. 3608–3624 (cit. on p. 132).

Yu, Eric S. and John Mylopoulos (1994). "Understanding "Why" in Software Process Modelling, Analysis, and Design". In: *International Conference on Software Engineering*, pp. 159–168 (cit. on p. 84).

Zhang, Jing, Yuehua Lin, and Jeff Gray (2005). "Generic and domain-specific model refactoring using a model transformation engine". In: *Volume II of Research and Practice in Software Engineering*. Springer, pp. 199–218 (cit. on p. 57).