A MULTILEVEL MODELLING INFRASTRUCTURE FOR THE DEFINITION, EXECUTION AND COMPOSITION OF DOMAIN-SPECIFIC MODELLING LANGUAGES

Doctoral Dissertation by
Alejandro Rodríguez Tena

Thesis submitted for
the degree of Philosophiae Doctor (PhD)
in
Computer Science:
Software Engineering, Sensor Networks and Engineering Computing

Department of Computer Science,
Electrical Engineering and Mathematical Sciences

Faculty of Engineering and Science

Western Norway University of Applied Sciences

August 16, 2021

Series of dissertation submitted to
the Faculty of Engineering and Science,
Western Norway University of Applied Sciences.

ISSN: 2535-8146
ISBN: 978-82-93677-57-4

Author: Alejandro Rodríguez Tena
Title: A Multilevel Modelling Infrastructure for the Definition, Execution and Composition of Domain-Specific Modelling Languages

Printed production:
Molvik Grafisk / Western Norway University of Applied Sciences

Bergen, Norway, 2021

TO THEM,

because I cannot express how grateful
I am for their unconditional
and endless love

PREFACE

The author of this thesis has been employed as a Ph.D. research fellow in the software engineering research group at the Department of Computer Science, Electrical Engineering and Mathematical Science at Western Norway University of Applied Sciences. The author has been enrolled into the PhD programme in Computer Science: Software Engineering, Sensor Networks and Engineering Computing.

The research presented in this thesis has been accomplished in cooperation with the Western Norway University of Applied Sciences and the University of Málaga, Spain.

This thesis is organized in two parts. Part I is an overview article organised into chapters that motivates the need of this thesis and discuss the state of the art and related work. It also introduces the foundational aspects and elements of this work, and serves as introduction to what is detailed in the collection of articles. Part II consists of a collection of published and peer-reviewed research articles and submitted papers.

Paper A A. Rodríguez, L. M. Kristensen and A. Rutle. Formal Modelling and Incremental Verification of the MQTT IoT Protocol. In Transactions on Petri Nets and Other Models of Concurrency XIV, volume 11790 of Lecture Notes in Computer Science, pages 126-145, Springer International Publishing, 2019.

Paper B A. Rodríguez, L. M. Kristensen and A. Rutle. Verification of the MQTT IoT Protocol Using Property-Specific CTL Sweep-Line Algorithms. In Transactions on Petri Nets and Other Models of Concurrency XV, volume 12530 of Lecture Notes in Computer Science, pages 165-183, Springer International Publishing, 2021.

Paper C A. Rodríguez, F. Durán, A. Rutle and L. M. Kristensen. Executing Multilevel Domain-Specific Models in Maude. In Journal of Object Technology, Volume 18, no. 2, pages 4:1-21. 2019.

Paper D A. Rodríguez, F. Macías, F. Durán, A. Rutle and U. Wolter. Composition of Multilevel Domain-Specific Modelling Languages. Submitted to the Journal of Logical and Algebraic Methods in Programming, Elsevier Ltd, 2020.

Paper E A. Rodríguez, F. Durán and L. M. Kristensen. Execution and Analysis of MultEcore Multilevel Modelling Languages using Maude. Submitted to the International Journal on Software and Systems Modeling, Springer International Publishing, 2021.

Paper F A. Rodríguez and F. Macías. Multilevel Modelling with MultEcore: A contribution to the Multi-Level Process Challenge. Submitted to Enterprise Modelling and Information Systems Architectures, 2021.

ACKNOWLEDGMENTS

I have to start by thanking my supervisors, Adrian, Lars and Paco. Without their absolute support, commitment and help, this thesis would not be possible. I still find it difficult to imagine the amount of time they have dedicated to me and to my work, even when they had several other duties but still managed to be there when I needed it. To Adrian, who was always there to advise, guide and support from the very beginning, especially during the application time before starting the PhD and until the end of it. He went beyond the professional matter and treated me as part of his family, inviting me to his place innumerable times. To Lars, who always had his door open and was willing to help me no matter his countless obligations. He had the ability to reduce, even the most stressful tasks, to the minimum, no matter what the problem was. To Paco for his unconditional dedication. Even though he joined later, he was fully committed from the first moment and he always found the time to talk, meet and discuss. He was behind my research stay in Málaga, where I felt as being at home for their hospitality and kindness. I cannot thank enough the amount of time he has dedicated to me, in spite of the tremendous amount of responsibilities he always had to deal with. To the three of you, again, thank you so much.

I want to dedicate this thesis to the Spilab team, especially to Juanma, Javi and Jose. They welcomed me from the first time I wanted to explore beyond the regular bachelor and master duties in Cáceres. To Juanma, who is one of the kindest persons I know. He was always smiling, willing to accept challenges, and treated me as one more of the department family. Furthermore, he helped me to achieve one of my personal goals at that time, which was to leave Spain and work in a different country. To Javi for being such a nice and smart person. He was always spreading a nice atmosphere and making sure that I was fine and comfortable with them. And to Jose, who was not only an amazing person but also a perfect mentor in my first job at Viable. We worked together, travelled together and he made me feel serene even in the tensest moments. I also want to thank Jaime Parodi, who gave me the opportunity to work at Viable and from whom I learnt a lot, in different aspects. They all made it possible for me to start my adventure in Norway.

To my PhD colleagues who created such a nice working environment and helped me both in the personal and academic aspects. Special mention to Rui, Patrick, Håkon, Suresh, Simon, Lucas, Faustin, Anton and Salah who were always willing to go to drink some beers, go to the cinema or have dinner. I want to hugely thank Frikk, for being such a special person. He is responsible for making our working room a fun and enjoyable place. His spontaneity, insanity and hospitality make him a unique person and a true friend. To the members of the Software Engineering, Sensor Networks and Engineering Computing department. They gave me this opportunity and I will always be thankful for everything this PhD journey has given to me. Thank you Volker and Violet for being so kind and even welcoming me to your place. Also, thank you, Kristin, Pål and Håvard, for being always reachable and help me in numerous situations.

To Fernando. I do not have enough words to express what an incredible person you are. It is pointless to enumerate how many times you have helped me, in every aspect. I admire you and consider you one of the best persons I have ever met. I truly appreciate this journey for giving me the opportunity to meet you and I am looking forward to seeing what you achieve in the future.

To my friends in Spain, who were always supporting and encouraging me to continue this adventure. To Papy and Noemí, for being so supportive and even coming to visit us in Norway. I cannot wait to gather with you again and share our moments and experiences. To my friends in Villanueva—The GD Crew—for making me feel so good every time I came back from Norway, even if it was a short period of time. I would like to greatly thank Conchita for being there all these years (and because she would be upset if I say nothing here about her!). No, but seriously, you are a gift and I am truly happy of keeping our friendship after everything we have lived together. You deserve the best and I am sure we will always have each other to rely on.

I want to especially dedicate this to the True Staff. To my cousin Alfonso for taking everything so easy and gluing the group together. I simply cannot wait to see what the future is holding for us, but it will surely be together. To Manu for your ironic intelligence and the fruitful discussions that only a few people can give nowadays. To Edu for the dynamism and freshness you give to our little family. I wish we could see each other more often. To Sergio, because you are always there to talk and be supportive about personal matters. And to Pedrito. You are my cousin but I consider you my little brother. I am so happy of seeing you grow and accomplish your goals. I am sure that you will always achieve what you set your mind to and I hope to be always there to see it. Just go ahead and fight for it.

To my family, because you are my fundamental pillar and you have always believed in me. Especially to Adela, who is one of the strongest persons I know, this thesis is dedicated to you. To Angelito, for all those moments we share talking about music, movies, etc. It is so pleasant to be able to spend an entire day talking and still enjoying so much each little detail.

To my parents. Mom, dad, this thesis would not be possible without your endless love and support. You have taught me everything I know and you have been the best guides anyone could wish for. Thank you for encouraging me to initiate this Norwegian adventure. I am so proud of being your son and share every aspect of my life with you. I am truly thankful for everything you have done for me and still do.

And finally, to you, Ángela. I admire you for achieving everything you propose, you are a treasure and an extraordinary person. While I could spend pages describing how wonderful you are, I cannot find the words to express how much you mean to me. You have always been there, especially in the difficult times, staying positive and transmitting that feeling to me. We started this journey together and we are about to finish it together. I am completely sure that I could not have gotten so far without your support, encouragement and love. We have lived a lot together, but the best is yet to come; let's just do it.

ABSTRACT

Nowadays software has scaled to a point where it is present in every aspect of our lives. It is paramount to explore techniques that speed up the construction of software and desirably increase the reliability and quality of the final products. In particular, the adoption of software systems to support almost all kinds of tasks for the society has enforced the needs to explore the development of Domain-Specific Modelling Languages (DSMLs) to bridge the gap between clients, domain experts and software engineers. Moreover, the pervasive and concurrent nature of today's software has increased the interest in the behavioural aspects of modelling languages, facilitating the simulation, execution and verification of models prior to their actual implementation to reveal errors and potential misbehaviours.

One of the most prominent fields of Model-Driven Software Engineering (MDSE) is the design and implementation of DSMLs. The creation of DSMLs involves the specification of different abstraction levels to distribute the concepts of the domain appropriately. Limitations in MDSE, especially regarding a strict number of available abstraction levels, have motivated the research and application of Multilevel Modelling (MLM). While the original motivations for MLM were mainly focused on eliminating such a strictness, in recent years the proliferation of MLM approaches has allowed to expand its practical applications in many areas, for instance, in complex and distributed systems. The complexity of these systems has promoted the specification of different modelling artefacts and sublanguages to handle different parts of the systems. This need for reusing system parts and reasoning about global properties has led to the invention of a multitude of approaches for language composition. In order to utilise MLM in such complex systems, one of the novel research paths for MLM has focused on applying language composition techniques in the MLM setting.

In this thesis, we further develop MultEcore which was originally developed as an approach and a tool for the specification of multilevel modelling languages. We have also extended the underlying formalisation and the practical implementation for the execution and analysis of MLM hierarchies and the composition thereof. The execution is described using Multilevel Coupled Model Transformations (MCMTs), while the analysis is performed by utilising Maude. For the composition, we have employed an amalgamation technique for MLM hierarchies and MCMTs which has its foundation from graph transformation and category theory. In particular, MLM hierarchies are composed using multi-typing and the concept of supplementary hierarchies, while MCMTs from the component hierarchies are merged together—taking into account various merge-strategies—to multi-typed rules which are applicable on the composed MLM hierarchy. The results of the thesis are evaluated with a case study in the Coloured Petri nets (CPNs) domain.

SAMMENDRAG

Programvare spiller en viktig rolle i alle aspekter av våre liv. Det er derfor viktig å forske på teknikker som effektiviserer konstruksjonen av programvare og samtidig øker påliteligheten og kvaliteten på sluttproduktene. Samtidig har bruk av programvaresystemer som støtter nesten alle slags samfunnsoppgaver økt behovet for å forskning på utviklingen av domenespesifikke modelleringsspråk (DSML—Domain-Specific Modelling Languages) for å bygge bro mellom klienter, domeneeksperter og programvareingeniører. Videre har interessen for å studere oppførselsaspektene ved modelleringsspråk økt i takt med utbredelsen av komplekse og samtidise programvare, noe som åpner for avdekking av feil og mangler ved hjelp av simulering og verifikasjon av modeller før de faktisk blir implementert.

Et av de mest fremtredende feltene innen modelldrevet programvareutvikling (MDSE—Model-Driven Software Engineering) er utforming og implementasjon av DSML-er. Konstruksjon av DSML-er innebærer spesifikasjon av forskjellige abstraksjonsnivåer for å representere domenekonsepter der de hører til. Begrensninger i MDSE, spesielt når det gjelder restriksjon i antall abstraksjonsnivåer som er tilgjengelige for å definere DSML-er, har gitt opphav til forskningsfeltet multinivå modellering (MLM—Multilevel Modelling). Den opprinnelige motivasjonen for MLM var hovedsakelig å redusere disse begrensingene, men nå har utbredelsen av MLM-tilnærminger de siste årene ført til en utvidelse av dens praktiske anvendelser.

For å håndtere kompleksiteten av dagens programvaresystemer brukes det ulike modelleringsspråk og -artefakter for spesifikasjonen av systemenes mangfoldige aspekter. Dette har også økt behovet for tilnærminger for gjenbruk og sammensetting av systemer samt for resonnering over globale egenskaper for de sammensatte systemene.

I denne avhandlingen videreutvikler vi MultEcore—et rammeverk som opprinnelig ble utviklet som en tilnærming og et verktøy for spesifikasjon av multinivå modelleringsspråk. Vi har også utvidet den underliggende formaliseringen og den praktiske implementeringen for eksekvering og analyse av MLM-hierarkier og sammensetting av disse. Eksekveringen er beskrevet ved hjelp av multinivå og sammensatte modelltransformasjoer (MCMT—Multilevel Coupled Model Transformations), mens analysen utføres ved å benytte Maude. For sammensetting av DSML-er har vi utviklet en teknikk for MLM-hierarkier og MCMT-er, som har sitt fundament fra graftransformasjon og kategoriteori. MLM-hierarkier blir sammensatt ved bruk av fler-typede modellelementer og supplerende hierarkier, mens MCMT fra komponenthierarkiene blir slått sammen—med anvendelse av ulike sammenslåingsstrategier— til fler-typede regler som kan anvendes på sammensatte MLM-hierarkier. Resultatene av avhandlingen er evaluert via studier i domenet Coloured Petri nets.

Contents

Part I

OVERVIEW

Not all those who wander are lost.

—J.R.R. Tolkien

CHAPTER 1

INTRODUCTION

Software has been an important driving factor for innovations in our society in the past 25 years. It has changed almost every aspect of our daily life as well as redefined many industrial sectors and created new ways of transportation, production and communication. Society is more and more dependent on software and data, which in turn is playing a decisive role in most engineering areas.

The increasing complexity of software requirements to satisfy the needs of our society has enforced the usage of higher levels of abstraction, making the solutions closer to lay person's understanding, and further away from the binary instructions that computers process. One of the paths towards handling complexity and providing more accessible solutions and implementations—which is an obvious ingredient in all other engineering disciplines—is *software modelling*.

1.1 Modelling

Although there exist several definitions of a *model*, we can define a model as "a simplified representation of certain reality that focuses on one particular aspect of a system" [34]. Throughout history, the human mind has constantly interpreted reality by applying cognitive processes that adjust its subjective perception. Modelling is a process deducted from the idea of *abstraction* which consists of the capability of finding the commonality in many different observations and thus generating a mental representation of reality. Informally, a model is a simplified representation of some real aspect and therefore it can never describe reality in its entirety [45].

Models have especially become crucial in technical and engineering fields such as mechanical and civil engineering, and ultimately in computer science and software engineering. In production processes, modelling allows investigating, verifying, documenting and discussing the properties of products before they are actually produced [101].

Originally, modelling was adopted with the purpose of sketching, designing and abstracting a system that would later be implemented following well-defined steps in the software engineering life-cycle. Within software engineering, models play an essential role and are the fundamental elements in the Model-Driven Software Engineering.

1.2 Model-Driven Software Engineering

Model-Driven Software Engineering (MDSE) [45, 219] emerged from the goal of tackling the continually increasing complexity of software. In other words, using models not only as a documentation artefacts but also to generate code as well as analyse, verify and test the system prior to its actual implementation. In recent years, the research community has argued that MDSE is a successful approach in terms of quality and effectiveness gains [168, 242] as well as increasing the productivity and flexibility in developing software [1, 213].

As models play a pervasive role in MDSE, models can also be represented as "instances" of some more abstract models. Hence, exactly in the same way we define a model as an abstraction of phenomena in the real world, we can define a metamodel as yet another abstraction, highlighting the properties of the model itself. In a practical sense, metamodels constitute the definition of a modelling language, since they provide a way of describing the whole class of models that can be represented by that language [45]. Most traditional MDSE approaches, such as the Eclipse Modelling Framework (EMF) [221] and the Unified Modelling Language (UML) [235], are based on the Meta-Object Facility (MOF) [166] standard of the Object Management Group (OMG). While this standard defines a 4-level architecture, in practice, designers find only two abstraction levels available, i. e., (meta)models that describe the language, and their instances representing concretisations of it. We provide a detailed description of this architecture in Chapter 2.

The increasing complexity of software systems demands substantial knowledge in the domain for which the system is developed. To accomplish an optimal, functional and correct product, the domain experts and the software system engineers have to find a common ground of understanding [93]. Furthermore, the knowledge gap between the clients (experts in their problem domain) and the software engineers (experts in one or more solution domains) might lead to misunderstandings and misspecified requirements which ultimately would result in project overruns w.r.t time, budget, project failure, or even software malfunctioning which may threaten health and life. In MDSE, the construction of Domain-Specific Modelling Languages (DSMLs) is utilised to overcome these challenges [168]. DSMLs are modelling languages that are tailored to a concrete application area [127], bridging the knowledge gap between software engineers and domain experts. At the same time, they contribute to model quality, since the concepts that are provided by a DSML are usually the result of a thorough development process.

Traditional MDSE approaches present a limitation in the number of abstraction levels that one can use to specify a modelling language. A two-level architecture, such as EMF, restricts the description of a domain within one metalevel using the natively available metamodelling facilities like type definition, inheritance and data types. However, these resources are not available at the model level, which forces the modeller to explicitly model them at the metamodel level, resulting in accidental complexity [145]. This, and other limitations such as model convolution or the fact that the modeller has to mix concepts that belong to different domains, have been widely discussed in the literature (see e. g., [18, 22, 23, 71, 145]).

1.3 Multilevel modelling

Multilevel modelling (MLM) is a research area that can bring solutions to several challenges regarding model convolution, accidental complexity and mixing of concepts belonging to different domains that are present in traditional two-level approaches [15, 17, 20, 22, 23, 66, 71, 103, 145]. MLM represents a significant extension to the traditional two-level object-oriented paradigm with the potential to effectively improve upon the utility, reliability, and maintainability of models. MLM is fundamentally based on the idea of expanding the type-instance relation between the level of metamodels and the level of their instances to more than these two levels.

The MLM community has shown that MLM is a favourable approach in several domains, including process modelling, software architecture [22, 24] and gamification mechanisms [48]. Currently, there exist a plethora of MLM approaches [12, 65, 122, 140, 141, 174, 225, 228, 239], but they all have in common the idea of not limiting the number of abstraction levels. Thus, MLM techniques are excellent for the creation of DSMLs which intuitively require at least three abstraction levels to be available. This is because domain-specific variants are usually built on top of a more generic metamodel and, moreover, the domain-specific metamodels are further instantiated to represent concrete configurations of such domains [66, 155]. We explore MLM, its state of the art and our approach to MLM in Chapter 2.

1.4 Composition

Modularisation, extendibility and reusability have been widely explored by the modelling and language engineering communities [160]. The complexity of software systems has promoted the construction of modelling languages by building different modular artefacts that describe different aspects of software systems. However, it is common to achieve global integrity in large systems to reason about global properties that check the correctness of the entire system. Indeed, some DSMLs might provide similar language constructs that could be composed and reused across further DSMLs [251]. To provide modularisation and composition of modular artefacts, various composition techniques have been explored in the literature [75, 81, 124, 129, 157, 160, 171]. When it comes to MLM, these modularisation and composition techniques are not completely exploited. We explore the existing composition techniques and present our approach in Chapter 3.

1.5 Execution and verification

One of the key advantages of MDSE is the ability to detect errors prior to the implementation of the system. While the definition of the structural aspect of the modelling language can reveal syntactic errors, the most important aspect is to define the semantics of such a language. Providing behavioural descriptions, often employing in-place model transformation rules [161], allows executing a system (a model instance) by repeatedly applying such rules that lead from one system state to the next one. Further steps, for instance, calculating all the possible executions (generating a state space), opens the door for verification and model checking [28] that allow modellers to check whether certain behavioural properties are satisfied. Even though various approaches have been proposed for the definition and simulation of behavioural models based on reusable model transformations, these rely on traditional two-level modelling [43, 126, 223]. Only a few approaches within the MLM community face modelling the behaviour of multilevel systems through model transformations, e. g., [14, 65], and apply them for execution/simulation. However, to the best of our knowledge, none of them handle any form of verification. We further explore various execution techniques and detail our infrastructure in Chapter 4.

1.6 Coloured Petri nets

To illustrate the abovementioned limitations and evaluate our contributions to the MLM domain, we consider one of the industrial domains in which MDSE has successfully been applied, namely, distributed systems. Coloured Petri Nets (CPNs) [116, 118] is a modelling framework in the distributed systems domain that facilitates, among others, the specification of communication protocols [77], data networks [38] and distributed algorithms [188]. CPNs belong to the family of high-level Petri nets (PNs) [119], which are characterised by combining classical Petri nets [187] with a programming language [234].

CPN Tools [136] is a software tool that supports the construction, execution, state space analysis, and performance analysis of CPN models. Even though (C)PNs is a general-purpose language, several recent applications of CPNs have shown that it would be beneficial to facilitate the development of domain-specific CPN-variants [217]. While a major advantage of CPNs is that they contain few but powerful modelling constructs, CPN Tools lacks mechanisms to define domain-specific concepts. Furthermore, it lacks support for basic modelling concepts such as modularisation and separation of data type declaration from behaviour definition, thus hampering its extensibility and adaptability to new domains. We describe and illustrate these CPN-related challenges in Chapter 5.

1.7 Research questions

Our starting premise for this thesis was that some fundamental ideas such as composition and execution were barely explored by the existing MLM approaches. This motivated us to further develop the MultEcore MLM framework to support the definition, composition, execution, and verification of multilevel DSMLs. As one of the successful modelling frameworks in MDSE, we chose CPNs and its de facto tool (CPN Tools) to evaluate parts of our contributions to the multilevel DSMLs domain. While studying CPN Tools, we also identified several challenges related to its modularity, reusability, and ability to define domain-specific concepts which our MLM approach could handle.

Based on this we have defined therefore four research questions to be answered in this thesis:

RQ1: *Research question 1 — How can MLM be used to alleviate the shortcomings of Coloured Petri nets and the CPN Tools?*

This question concerns the study of the CPN language and the CPN Tools, the understanding of the missing features, and how they may be improved using MLM techniques. It is also to study the MLM state of the art to understand different approaches and their features with the goal to alleviate current CPN challenges.

RQ2: *Research question 2 — How can reuse across related multilevel DSMLs be facilitated?*

This question concerns the study of reusing mechanisms and how they can be successfully implemented in the MLM domain. This requires investigations into the definition of both MLM hierarchies and their corresponding MCMTs, especially in cases where several related DSMLs belong to a language family. This question is closely related to the CPNs case in which a family of PNs-based languages, and domain-specific variants of them, could share a common structure and behaviour.

RQ3: *Research question 3 — How can the underlying theory be adapted to achieve the composition of MLM hierarchies and amalgamation of their respective MCMT rules?*

This question concerns the study of the underlying theory behind MLM hierarchies and their corresponding MCMTs. As the MultEcore approach is founded upon graph transformations and category theory, we have to investigate how MLM hierarchies can be composed and how MCMTs are combined and applied to the composed MLM hierarchies.

RQ4: *Research question 4 — How can a term-rewriting engine like Maude be used as an execution engine for MCMTs with the goal to simulate and verify multilevel DSMLs?*

The last question concerns the incorporation of execution and verification into our approach. It involves the development and evaluation of an infrastructure for the execution of multilevel models which are defined in multilevel DSMLs by applying MCMTs and the verification of certain behavioural properties. Answering the question requires investigating the Maude system [54], the seamless integration of Maude and MultEcore, and means to interpret and transfer execution and verification results from Maude to MultEcore. Since the CPN Tools has a proven strength in state space analysis and model checking, the results of this question would showcase that applying an MLM approach to the CPN case would not lose this strength.

1.8 Research method

The research method followed along the course of this thesis is *Constructive Research* [179]. The constructive research method can be phenomenon-driven, theory-driven or the combination of the two. In our case, we employ a combination of both. This methodology begins with a strong grounding in identifying a practical problem complemented by related literature. The identified research problems are used to propose research questions that address the problem. The questions are solved by developing or constructing a solution (e. g., a framework or a tool) that has to be implemented to determine its feasibility, practicality and usability (often tested through case studies and feasibility studies). In constructive research, the goal is to define and solve problems, as well as to improve an existing system or its performance, with the overall implication of adding to the existing body of knowledge. The construction proceeds through design thinking that makes projection into the future envisaged solution (theory that is required and artefacts that will be constructed) and fills conceptual and other knowledge gaps by purposefully building tailored blocks to support the whole construction [78].

A common starting point in many research activities is the specification of research questions that scope the goals of the research activities. The research questions outlined in Section 1.7 were formulated based on the study of the shortcomings present in traditional MDSE approaches and in the CPNs case—which is used as a witness of a successful MDSE framework. These questions, especially RQ1, possess an *exploratory* nature and are classified as *knowledge questions* [82], where we attempt to understand the phenomena and identify useful distinctions that clarify our understanding. In our case, this concerns how CPNs and other modelling languages work and what flaws they present. Essentially, in the early phases of our work, we were on an exploratory and theoretical phase, where we aimed at understanding the current state of the art on MLM and CPNs. The remaining research questions, specially RQ4, which is the most focused on practical contributions, can be classified as *design questions* within *non-empirical* research where the goal is to design better procedures and tools for carrying out some activity [82]. These kinds of questions assume that the associated knowledge questions have already been addressed (which was our case, as the infrastructure was constructed upon the developed theory) so that we have enough information about the nature of the design problem to be solved.

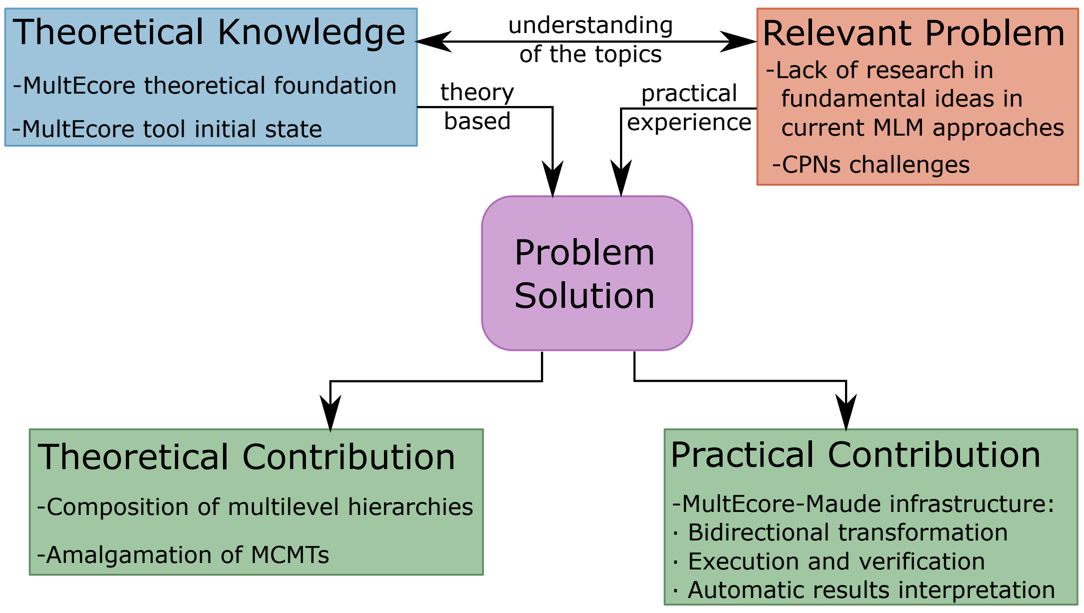

Theoretical Knowledge
-MultEcore theoretical foundation
-MultEcore tool initial state
understanding of the topics
Relevant Problem
-Lack of research in fundamental ideas in current MLM approaches
-CPNs challenges
theory based
practical experience
Problem Solution
Theoretical Contribution
-Composition of multilevel hierarchies
-Amalgamation of MCMTs
Practical Contribution
-MultEcore-Maude infrastructure:
· Bidirectional transformation
· Execution and verification
· Automatic results interpretation

Fig. 1.1: Constructive research method

Figure 1.1 represents the constructive research method. **Theoretical Knowledge** can be associated with the current state of the art on MLM that served as a starting point of this thesis and is detailed in Chapter 2. We took the MultEcore approach as a starting point due to its extendibility potential and the already developed theory and tooling. The **Relevant problem** is about the already discussed scarcities that MLM, CPNs and the CPN Tools currently have. These mostly concern a lack of research in composition, execution and verification techniques applied to MLM. Composition is discussed in Chapter 3 and execution and verification in Chapter 4. We explore the modelling of CPNs and our solution applying the developed MLM techniques using MultEcore in Chapter 5.

The study of the theoretical knowledge and the relevant problem enhances the understanding of the problem and the real needs as well as helps to connect the existing theory to potential solutions. The **Problem Solution** is our proposal that aims at building an infrastructure where one can define, compose, execute and verify multilevel DSMLs. This solution is proposed based on the studied underlying theory and the practical experience of the problems. Then, the outcome is twofold. First, providing a **Theoretical contribution** on top of the theoretical foundation of MultEcore, to handle the composition of structure and behaviour of MLM hierarchies. Our composition approach is detailed in Chapter 3. Second, providing a **Practical contribution** that corresponds to the developed infrastructure to handle the specification, execution and verification of MLM hierarchies as presented in Chapter 4. The constructed infrastructure that connects MultEcore with the Maude system allows the modeller to interact with Maude tools that facilitate execution strategies and verification mechanisms as well as the automatic interpretation of the produced results back into MultEcore graphical models. The MultEcore-Maude infrastructure is explored in Chapter 4 and its application to the CPNs case in Chapter 5. To further evaluate our infrastructure, we submitted a solution to the MULTI Process Modelling Challenge [5] (see Paper F for the details).

1.9 Summary of papers

This section lists the six papers that form the basis of this thesis and that are included in Part II. Then it summarises the content of each paper.

[196] A. Rodríguez, L. M. Kristensen and A. Rutle. Formal Modelling and Incremental Verification of the MQTT IoT Protocol. In Transactions on Petri Nets and Other Models of Concurrency XIV, volume 11790 of Lecture Notes in Computer Science, pages 126-145, Springer International Publishing, 2019.

[198] A. Rodríguez, L. M. Kristensen and A. Rutle. Verification of the MQTT IoT Protocol Using Property-Specific CTL Sweep-Line Algorithms. In Transactions on Petri Nets and Other Models of Concurrency XV, volume 12530 of Lecture Notes in Computer Science, pages 165-183, Springer International Publishing, 2021.

[193] A. Rodríguez, F. Durán, A. Rutle and L. M. Kristensen. Executing Multilevel Domain-Specific Models in Maude. In Journal of Object Technology, Volume 18, no. 2, pages 4:1-21. 2019.

[201] A. Rodríguez, F. Macías, F. Durán, A. Rutle and U. Wolter. Composition of Multilevel Domain-Specific Modelling Languages. Submitted to the Journal of Logical and Algebraic Methods in Programming, Elsevier Ltd, 2020.

[194] A. Rodríguez, F. Durán and L. M. Kristensen. Execution and Analysis of MultEcore Multilevel Modelling Languages using Maude. Submitted to the International Journal on Software and Systems Modeling, Springer International Publishing, 2021.

[200] A. Rodríguez and F. Macías. Multilevel Modelling with MultEcore: A contribution to the Multi-Level Process Challenge. Submitted to Enterprise Modelling and Information Systems Architectures, 2021.

1.9.1 *Paper A: Formal Modelling and Incremental Verification of the MQTT IoT Protocol*

This paper [196] was accepted for the Transactions on Petri Nets and Other Models of Concurrency in 2019. In this paper, we present a formal specification of the Message Queuing Telemetry Transport Protocol (MQTT) [30] in the form of a CPN model. The model covers the three quality of service levels defined by the MQTT specification. This model can be simulated to inspect the exchange of messages between clients (who can subscribe to topics) and the broker. We also present in this paper an incremental model checking approach for the verification of properties that can be used to reduce the effect of the state explosion problem. This incremental approach is based on the fact that the MQTT protocol operates in phases comprised of connect, subscribe, publish, unsubscribe and disconnect. To verify the model we conduct model checking of several behavioural properties.

1.9.2 *Paper B: Verification of the MQTT IoT Protocol Using Property-Specific CTL Sweep-Line Algorithms*

This paper [198] was accepted for the Transactions on Petri Nets and Other Models of Concurrency in 2021 and is based upon the work presented in Paper A. In this paper, we investigate how to alleviate the effect of the state space explosion problem. Specifically, we implement the sweep-line method in Standard ML (SML) and integrate it in the CPN Tools. The sweep-line method allows us to carry out model checking while deleting states from memory during state space exploration. The behavioural properties are formulated using property-specific Computation Tree Logic (CTL) model checking algorithms that we implement to make verification of certain properties compatible with the sweep-line method. As a result, there is a substantial reduction in memory usage at the expense of a modest increase in execution time.

1.9.3 *Paper C: Executing Multilevel Domain-Specific Models in Maude*

This paper [193] was accepted for the Journal of Object Technology in 2019. In this paper, we propose an approach to define multilevel hierarchies and specify its behaviour in a flexible way. We make use of an improved version of the Multilevel Coupled

Model Transformations (MCMTs) to enhance horizontal and vertical flexibility which make them reusable across different domains. Furthermore, we present a preliminary version of an infrastructure that connects MultEcore with Maude which allows us to execute models. We use a Product Line system as case study to demonstrate that the specified MCMT rules can be used in models belonging to different branches of the hierarchy and that these models are executed to observe their evolution.

1.9.4 Paper D: Composition of Multilevel Domain-Specific Modelling Languages

This paper [201] is currently under review at the Journal of Logical and Algebraic Methods in Programming. In this paper, we extend our MLM approach to support composition. We propose a composition mechanism for structure and behaviour of multilevel modelling hierarchies based on the concept of supplementary hierarchy. Our approach facilitates the inclusion of additional typing chains (allowing elements to have more than one type) while keeping a clear separation of concerns which enhances modularity. We also show how MCMT rules can be amalgamated to create composed transformation rules. Furthermore, we apply our proposal to a case study and illustrate how we have developed a semi-automatic amalgamation mechanism to produce amalgamated MCMT rules.

1.9.5 Paper E: Execution and Analysis of MultEcore Multilevel Modelling Languages using Maude

This paper [194] is currently under a second round of review for a theme issue on multilevel modelling in the International Journal on Software and Systems Modelling. In this paper, we built upon the preliminary infrastructure presented in Paper C. First, we improve the expressivity and applicability of MCMTs by incorporating support for attributes, for conditional rules and for nested boxes to handle submodel patterns. Second, we give a rewrite logic semantics to MLM, on which we have based our automated transformation from MultEcore to Maude. Furthermore, we highly improve our infrastructure including support for the Object Constraint Language (OCL) [51], with a user-friendly interface that hides Maude as a background process in MultEcore, and that provides tools for the execution, reachability analysis and model checking of models. Different forms of execution are made available. Even though the analysis performed is limited, it already allows the analysis of infinite state spaces through non-trivial forms of abstraction.

1.9.6 Paper F: Multilevel Modelling with MultEcore: A contribution to the Multi-Level Process Challenge

This paper [200] is currently under review for the Enterprise Modelling and Information Systems Architectures Journal. This paper is a contribution to the MULTI Challenge series [5] that is intended to encourage the MLM research community to submit solutions to the same, well-described problem. In this work, we present our solution in the context of process management where we discuss how we handle each requirement and explain how MultEcore supports the construction of the proposed case study. We not only focus on the structural dimension of the mandatory and optional requirements, but also explore the specification of constraints and behaviour using MCMTs. We make use of the infrastructure presented in Paper E to execute the instance models that belong to the constructed multilevel hierarchy.

1.10 Contributions

Our approach for MLM, formally specified in [149] and Paper D, and reflected in the MultEcore tool [150, 199], rests on the premise that one must be able to specify multilevel models which are both generic and precise [155]. Since a multilevel hierarchy captures the structural aspect of a modelling language, we use our own multilevel transformation language that allows coping with the specification of the semantics, so-called Multilevel Coupled Model Transformations [149, 155]. MCMTs are flexible with respect to the multilevel hierarchy within their two dimensions. They are *vertically* flexible since new models can be introduced in the constructed multilevel hierarchy without affecting the application of the existing MCMT rules. They are also *horizontally* flexible since new branches might be integrated into the hierarchy and yet the existing MCMT rules would be automatically applicable to such new branches. Note that in cases where some branch defines its own specific MCMT rules, these are prioritised to override the more generic rules.

To cope with modularisation and composition in an MLM context, we have extended our underlying theory based on graph transformations and category theory, and developed an alternative composition technique. Specifically, multilevel hierarchies and MCMT rules are composed using multi-typing and the concept of supplementary hierarchies. The evaluation of the results related to composition (RQ3, Section 1.7) was done in the context of a case that was proposed by the MULTI Process Modelling Challenge [5]. Hence, these results are not validated in the CPN case.

To manage execution and analysis, we have developed an infrastructure that connects our MultEcore MLM tool with the Maude system [54], which implements a rewriting logic engine. In a nutshell, this infrastructure supports (i) the specification of multilevel DSMLs as multilevel hierarchies; (ii) the specification of the model transformation rules that describe their behaviour (via the MCMTs); (iii) the translation of this setting into Maude which we use for execution, analysis and verification; (iv) and finally the interpretation of the results from Maude into MultEcore. Note that every Maude-related aspect is hidden to the user, who directly interacts with the MultEcore interface.

Some of the aforementioned contributions have been evaluated by the CPNs case study and is documented in Chapter 5. The ideas integrated into MultEcore facilitate the definition of language families, e. g., PNs, CPNs as a refinement of PNs, domain-specific CPNs, and potentially further concretisations, providing the modeller with capabilities to treat each level as an instantiatable modelling language. This way, the specification of multilevel hierarchies enhances modularisation and facilitates extendibility [193, 204]. We have also evaluated our framework by providing a solution to the MULTI Challenge [5], where we model a multilevel hierarchy for process management, making use of the supplementary dimension and perform execution using the MultEcore-Maude infrastructure. A more detailed description of the contributions of this thesis can be found in Chapter 6. MultEcore and the MultEcore-Maude infrastructure are documented in [151] and [192], respectively.

1.11 Outline

This thesis is organised into two main parts. Part I starts with the present Chapter 1 giving an introduction to the essential aspects that motivates the research work underlying this thesis. Then, it details the state of the art and our solution to the following fields: (i) MLM, (ii) composition, (iii) execution and (iv) verification of multilevel DSMLs and CPNs. Finally, it discusses the contributions we have made and the results obtained. The reminder of Part I is comprised of:

Chapter 2: Multilevel Modelling.
This chapter gives first a roadmap to the evolution of MLM, emphasising its origins and the original aspects that motivated the research into MLM. Then, it revisits the key aspects that characterise the current MLM solutions and explores different MLM approaches and tools. Finally, it examines our approach and positions it with respect to the state of the art in MLM.

Chapter 3: Language Composition.
This chapter elaborates on motivational aspects that gave rise to the research into the composition of languages. Then, it explores existing techniques that handle composition and discusses relevant approaches and tools that currently support some of these composition techniques. Furthermore, it details our approach to handle the composition of multilevel modelling hierarchies and amalgamation of behavioural rules, and positions it with respect to the surveyed techniques.

Chapter 4: Execution and Verification.
In this chapter, we introduce execution and verification in the context of MDSE. We first classify methodologies that handle the execution of behaviour and analyse several existing tools that implement execution engines based on some of these methodologies. We also explore verification, especially model checking, and relevant approaches that incorporate mechanisms to verify the modelled systems. Finally, we describe our developed infrastructure and detail how we achieve execution and verification.

Chapter 5: The Coloured Petri nets Case Study.
In this chapter we provide the CPNs background that is necessary to understand the contributions made to the CPNs field and which has been used as a case study of our infrastructure. Also, we detail how and what MLM techniques have been successfully applied to build the multilevel hierarchy for executable CPNs.

Chapter 6: Related work, conclusions and future Work.
In this chapter, we discuss related work, revisit the research questions, summarise our contributions and outline directions for future work.

Part II consists of a collection of six journal articles, where three of them have been peer-reviewed and published [193, 196, 198] and the other three are currently under review [194, 200, 201]. We have summarised these six papers in Section 1.9. The three under-review papers are extensions of already published articles in workshops and conferences (see Section 1.12).

1.12 Supplementary material

In addition to the articles included in Part II (listed in Section 1.11), seven workshop articles have been published presenting initial research results obtained during the work for this thesis:

[227] A. Rodríguez, F. Macias, L. M. Kristensen and A. Rutle. Towards Domain-Specific CPN Modelling Languages. In Proceedings of the 29th Nordic Workshop on Programming Theory (NWPT'17).

[195] A. Rodríguez, L. M. Kristensen and A. Rutle. On Modelling and Validation of the MQTT IoT Protocol for M2M Communication. In the International Workshop on Petri Nets and Software Engineering (PNSE'18) (pages 99-118). CEUR Workshop Proceedings 2138.

[202] A. Rodríguez, A. Rutle, F. Durán, L. M. Kristensen and F. Macias. Multilevel Modelling of Coloured Petri Nets. In the 5th International Workshop on Multi-Level Modelling (MULTI'18) co-located with MoDELS conference (pages 663-672). CEUR Workshop Proceedings 2245.

[197] A. Rodríguez, L. M. Kristensen and A. Rutle. On CTL Model Checking of the MQTT IoT Protocol using the Sweep-Line Method. In the International Workshop on Petri Nets and Software Engineering (PNSE'19) (pages 57-72). CEUR Workshop Proceedings 2424.

[204] A. Rodríguez, A. Rutle, L. M. Kristensen and F. Durán. A Foundation for the Composition of Multilevel Domain-Specific Languages. 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), Munich, Germany, 2019, pp. 88-97, doi: 10.1109/MODELS-C.2019.00018.

[199] A. Rodríguez and F. Macías. Multilevel Modelling with MultEcore: A Contribution to the MULTI Process Challenge. 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), Munich, Germany, 2019, pp. 152-163, doi: 10.1109/MODELS-C.2019.00026.

[203] A. Rodríguez, A. Rutle, F. Durán, L.M. Kristensen, F. Macías and U. Wolter. Composition of Multilevel Modelling Hierarchies. In Proceedings of the 31st Nordic Workshop on Programming Theory NWPT 2019 (NWPT'19).

The work presented in the above listed workshop articles ([195, 197, 199, 202–204, 227]) have been extended and improved within the six journal articles included in Part II. Therefore, none of the listed workshop articles are formally included in this thesis.

CHAPTER 2

MULTILEVEL MODELLING

In this chapter, we introduce the original motivations that promoted the formulation of Multilevel Modelling (MLM) solutions and discuss the key concepts that define the core ideas of MLM and that have influenced the MLM community until today. Then, we explore existing MLM approaches and finally position our MLM solution with respect to the state of the art.

2.1 The MOF Architecture

The traditional two-level approaches such as the Unified Modelling Language (UML) [41] and the Eclipse Modelling Framework (EMF) [221], which are based on the OMG's 4-level Meta-Object Facility (MOF) [166] (see Figure 2.1), present some shortcomings that MLM aims to solve. In the MOF architecture, the topmost level **MetaMetaModel** (M_3) is reserved for MOF and is the core from which the descriptions of specific modelling languages (i. e., specific language metamodels) are created. The dashed arrows in the figure between levels indicate *instance-of* relationships. Below, **Metamodel** (M_2) defines one of the OMG metamodels, e. g., the UML class diagram and the UML object diagram. It is in the **Models** level (M_1) where the modeller can define user models. Here, in M_1, one uses the concrete language constructs to define specific model instances which represent concrete systems. Finally, the **User Data** level (M_0) associates elements of a UML object diagram or a class diagram to real-world

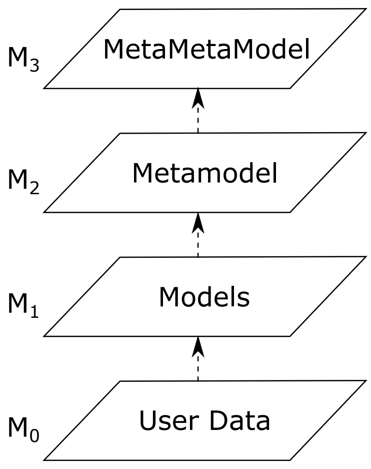


Fig. 2.1: OMG's 4-level architecture

objects. While it is described as a 4-level architecture, it is two-level in practical terms, as the modeller can only use M_1 and M_0 to specify the software systems.

In the MDSE community, there exist different schools of thought regarding the classification of the **User Data** level. While some consider that M_0 should be included in the 4-layer architecture, others argue for the separation of M_0 as it might just represent the modelled system [20, 21]. Although in the OMG hierarchy the relationship between M_0 and M_1 in Figure 2.1 is characterised as an instance-of relationship, those that do not consider M_0 as part of the modelling stack call it a *representation* relationship and consider it as a representation of real-world objects. This separation of the bottommost level w.r.t. the rest of the architecture is called "The 3 + 1 MDA organisation" (see [34] for discussion on this topic).

EMF is based on a variant of the architecture shown in Figure 2.1 that consists of only three levels. It has Ecore at the top which is EMF's metamodel. Below it, we find user models defined by the modellers to describe the software systems. At the bottom, we have instance models which contain individuals and concrete configurations. These can be understood as formal object diagrams which are class diagram instantiations. In practical terms, the architecture in Figure 2.1 and EMF's architecture allow customising only the two bottommost levels to specify software systems. In the case of creating DSMLs, both the OMG and EMF modelling architectures allow users to only modify two levels: M_1, for defining the modelling constructs, and M_0 for the definition of concrete instances of these languages.

2.2 The beginning of Multilevel Modelling

MLM is a prominent research area where models and their specifications can be organised into several levels of abstraction [17, 20, 103]. MLM recognises that some model elements may have a bifold type/instance nature, and hence it makes some metamodelling facilities available at every metalevel. In some situations, this may result in simpler models as the engineer does not need to explicitly model and give semantics to those metamodelling facilities or resort to artificial workarounds [22].

More than two decades ago, Atkinson and Kühne were already exploring, studying and analysing the scarcities of traditional MDSE approaches, especially UML, and how the identified flaws of these approaches could be addressed [11, 15–19]. While UML has been improved over the years, it still lacks some mechanisms to provide solutions to the construction of DSMLs in a natural way, as will be discussed throughout this section. In the late 1990s and the early 2000s, several frameworks, such as UML [41] and the Open Modeling Language (OML) [218] used to mix the instances and the types from which they are created at the same level (e. g., class and object, and association and link) [15]. One of the initial fundamental shortcomings was that even though the UML was based on a 4-level architecture, there was no precise definition of the "instance-of" relationship. In other words, the dashed lines in Figure 2.1 may have different flavours depending on which levels (M_3-M_0) they connect. This was called *loose metamodelling* [11] where a model placed at level M_n is an instance-of a model residing in level M_{n+1}.

An early attempt to introduce some rigour into the use and organisation of the level hierarchy was the formulation of the *strict metamodelling* principle. Strict metamodelling is based on the premise that if a model placed at level M_n is an instance of another model in level M_{n+1}, then every element of the model at M_n must be an instance of some element in the model at M_{n+1} [16]. Basically, it interprets the instance-of relationship not only in general between models, but at the granularity of individual model elements. Strict metamodelling mandates that levels have strict boundaries and are formed purely by instance-of relationships and not by any other unstated criteria. An important consequence of the strict metamodelling approach is that every model element has an instance facet and a type facet, both of which are equally valid [15]. This gave rise to one of the key concepts in the multilevel modelling community, the so-called *Clabject*, which states that every class appearing at any level within a metamodelling framework is not merely a class, but actually a class and an object. Note that this could not apply to the bottommost level as it might not act as metalevel in levels below.

Among others, a main goal of MLM is to reduce the accidental complexity, i. e., complexity added by introducing model elements only to express their multilevel nature [22, 145]. An example is the so-called Type-Object pattern [158] (also known as the Item Descriptor [58] pattern) to describe multiple domain levels in object-oriented languages. In such a pattern, objects play the role of classes, and as such, type-related information can be encoded in them. This means that classification is replaced by association, which—along with objects representing class information—leads to accidental complexity [22, 145].

Another shortcoming regarding the lack of an unlimited number of levels is the need to specify a family of languages where we usually have a topmost metamodel capturing the generic and common parts of the language family, which might then branch into different domains and potential subdomains (see [199, 247] and Paper F for some examples). These hierarchical constructions require well-defined infrastructures to handle the separation of abstraction levels. Therefore, the limitation in the number of levels of traditional MDSE approaches can, even more, affect the specification of reusable DSMLs. Having to rely, for instance, on generalisation inside one metamodel would lead to a single big metamodel where elements that belong to different domains have to be put together, making more difficult the tasks of maintaining, extending or updating the language.

There exist some alternative techniques within the two-level approaches to alleviate the problems described above to encode more information within two levels. The use of the Type-Object pattern itself as well as the Metamodel extension [145], Ad-hoc promotion transformations [145], tagged values, and representing domain metatypes as stereotypes [20, 22, 26] or powertypes [22, 178] are some examples. Still, they remain more as "workarounds" or "tricks". As pointed out by Atkinson and Kühne [19]: "The application of the Type-Object pattern is a symptom of the lack of logical metaclasses" (see [19] for an example of this problem). Hence, the aforementioned workarounds tend to increase the accidental complexity of domain models by polluting them with (inherently unnecessary for the domain itself) extra elements, making it a less accurate representation of the domain [22].

Although traditional instantiation is sufficient when dealing with only two levels (e. g., classes and objects), it should ideally be enhanced for a multilevel instantiation hierarchy. However, the problem with traditional instantiation (e. g., in UML) is that a type may only specify properties of its direct instances but has no bearing on, e. g., the instances of its instances. This is called *shallow instantiation* and introduces two problems: *multiple* or *ambiguous classification* and *replication of concepts*. In a nutshell, ambiguous classification refers to the ambiguity that arises when one has to reason, from a certain level (e. g., M_2), about how elements are defined and connected at the instance level (M_0), but it has to go through the intermediate level (M_1). In other words, the ambiguous classification problem appears because the traditional semantics of instantiation prohibits a model element from influencing anything other than its immediate instances. This forces the modeller to bypass the information through M_1 giving rise to a potential ambiguity where elements at M_0 are (or have to be) potentially classified by M_1 and M_2 which is vague and inconclusive. Replication of concepts is a consequence of the ambiguous classification: The strictness promoted by shallow instantiation that fails to carry information across more than one instantiation link makes it necessary to duplicate information at multiple levels just to carry the necessary information. We refer the reader to [18] for more details on these problems and how they are solved by the notions of *deep instantiation* or *deep characterisation* which are described by so-called *deep modelling*. Deep characterisation relaxes the strictness present in shallow instantiation. It is an instantiation mechanism where the features of an element's class can be acquired automatically by the instantiation step rather than always having to be defined explicitly. Deep instantiation and the key aspects that characterise it are explored in Section 2.3.

Instance-of relationships present another shortcoming regarding the impossibility of distinguishing different interpretations of them. This vagueness causes it not to scale up well for all requirements of what a model-driven development supporting infrastructure should fulfil (see [20, Section 2]). Álvarez et al. [6] presented one of the first works that introduced the Meta-Modeling Language (MML) which distinguished different forms of instance-of relations by abandoning the idea of linear framework organising the levels in a nested way. In that sense, an object at M_0 would have a type coming from its domain dimension (*logical classification*) and another type coming from its physical dimension (*physical classification*). However, there are other implications in the MML proposition such as that logical instantiation might be viewed as the same mechanism as physical instantiation, clearly blurring the differences between them. To overcome that, Atkinson and Kühne [19, 20] proposed to identify two separate orthogonal dimensions of metamodelling, giving rise to two distinct forms of instantiation:

- **Linguistic (physical) Dimension.** It is the dominant classification dimension from the viewpoint of tool builders. This essentially adopts the "UML as language" philosophy described above, and views the **Metamodel** (M_2) as defining the physical classifiers (abstract syntax) from which models are constructed (e. g., Class and Object).

- **Ontological (logical) Dimension.** It is the dominant classification dimension from the viewpoint of modellers. It focuses on classification within a concrete domain, and it does not involve the representation of concepts (i. e., the physical classification). It is in this dimension where there exists a clear need for an unrestricted number of classification levels (rather than relying, for instance, on generalisation in the same model) to express the needs of the domain. In this regard, Odell [178] demonstrates the practicality of associating the information with types so that they can be viewed as being instances themselves. Mili and Pachet [167] also provide some examples, for instance, showing how to express commonalities among models, which cannot be well captured with generalisation.

This organisational architecture is called Orthogonal Classification Architecture (OCA) [21] which is primordially based on level-compaction [20]. The OCA has been influencing the MLM community until today and it is currently the most widely used architecture for the specification of multilevel frameworks. It provides a genuine MLM platform, typically in the context of a single linguistic format definition, while enhancing the use of an unlimited number of ontological levels.

The shortcomings that MLM aims to alleviate have been widely discussed by the MLM community over the past two decades. Especially, de Lara and Guerra [25, 66–71, 106, 145] have studied the nature of these problems in detail and identified not only those patterns that would benefit from MLM solutions, but also cases where MLM might not be a suitable approach. Furthermore, they have explored other alternative solutions and carried on thorough research activities to classify and identify which solution suits best to handle each problem. Also, they have been involved in research works that compare MLM approaches and classify them [13, 46, 100, 113] (see also Section 2.4.3).

2.3 Aspects of Multilevel Modelling

In recent years, there has been a proliferation of MLM solutions. In the following, we introduce key concepts whose interpretation promotes a variety of different MLM approaches and tools.

2.3.1 *Levels*

Atkinson and Kühne introduced the terminology *Level-agnostic* [13] to approaches that do not make the treatment of an element dependent on its level in the ontological classification hierarchy. The idea of level-agnosticism for a modelling language proposes to treat all elements using a uniform notation, independently of which level they belong to, e. g., whether they are objects or classes [13]. In this context, a language can achieve level-agnosticism either by only recognising a single level (*level-blind*) or by explicitly using levels to structure the modelling elements (*level-adjuvant*).

Level-adjuvant approaches [13, 138]. Most of the current MLM approaches follow this doctrine. A level-adjuvant language is a level-agnostic language that recognises the utility of levels. In level-adjuvant approaches, a level is a numeric attribute defined for all elements in a multilevel model, which enables layering elements based on their level value. Most MLM approaches use the instantiation relation (with classification semantics) to relate levels. This has the advantage of enabling the use of mechanisms for deep characterisation that work across instantiation relations, like *potency* (we further discuss potency in Section 2.3.2).

In [138], Kühne explores a more fundamental research line to discern what "level" means, how to systematically organise elements into levels in a standard way, and other fundamental aspects regarding this concept. Frameworks such as the DPF Workbench [141], OMLM [112], Metadepth [65], Melanee [12], MultEcore [149], Dual Deep Modelling (DDM) [176] and FMMLx [50, 94] can be classified as level-adjuvant approaches.

Level-blind approaches [109]. A level-blind approach is one where a single level may contain arbitrary structures of ontological classification relations. Even though *level* is not explicitly modelled, it is often possible to intuitively derive them by analysing the solution implementation. Most of the level-blind approaches are based on the powertype pattern [49, 178] and form sequences of powertype relations between certain classes to implicitly establish levels. A level-blind approach has the advantage of allowing all elements to be treated in the same way since membership to ontological levels is abstracted away. Another advantage of having everything comprised in a single level is that cross-level relationships can be formed, a feature that most of the level-adjuvant approaches do not support as they only allow the "instance-of" relation between levels. Approaches such as DeepTelos [123], ML2 [64, 90] (which is built upon MLT* [3]), DMLA [228] and the programming languages DeepRuby [175] and DeepJava [140] are level-blind approaches.

2.3.2 *Instance characterisation*

Most of the MLM approaches can characterise instances at lower metalevels and not only the immediate one as in standard two-level modelling that follows shallow classification. As mentioned earlier, this is called deep characterisation [18]. In level-adjuvant approaches, the mechanism used to have some control on how deep to instantiate is called *Potency* [18, 19]. The first concept of potency was introduced more than 20 years ago and, since then, the proliferation of MLM approaches has also propitiated several alternative potency interpretations and proposals. In the following, we classify each of the existing potency variants.

- **Classic potency.** Its intuition is that one can assign potency values to model elements and their fields, indicating how many times they can be instantiated. Originally, the potency of a model element was an integer that defines the depth to which a model element can be instantiated [18]. Instantiating a model element amounts to reducing the level of the instances by one and also reducing its potency and the potencies of all its fields by one. Once potency reaches 0, it cannot be further instantiated. The classic potency notion demands the instantiation of every intermediate model element between level n and 0. In other words, the classic potency is aligned with strict metamodelling, where each element is an instance of exactly one element in the upper metalevel. Note that this could be too restrictive as one might sometimes be interested only in instances with potency 0. This would force to create clabjects at each intermediate metalevel — with a so-called identity instantiation [70] that does not introduce new information — but is required to instantiate clabjects at potency 0. In [137], Kühne first contextualises and identifies the limitations of classic potency. Then, he provides four different categories of potency interpretations as result of analysing existing MLM approaches based on how they align to the concepts of *order* and *level* [137, Section 5]. Finally, he proposes the notion of *characterization-potency* as an evolution of classic potency that rests on a new foundation that distinguishes between characterisation and classification.

- **Leap potency.** To alleviate the strictness of the classic potency and the need of using identity instantiation, de Lara et al. [70] introduced the so-called *leap potency* where instantiation is not mediated. Mediation occurs when an element must strictly be instantiated from the level right above it. Elements with a leap potency of n can be instantiated exactly n metalevels below, but not at intermediate levels, which remains as a way to "skip" the instantiation at the intermediate levels.

- **Multi-potency.** This was proposed by Rossini et al. [206] and is similar to classic potency where a node or a relation with a multi-potency p must be instantiated p times, on every level below. Rossini et al. also define a different kind of potency, so-called *single-potency*. A single-potency p on an element at metalevel i denotes that such an element can be instantiated (i. e., can be assigned a value) at metalevel i+p only. In this regard, single-potency works similar to leap potency.

- **Star-potency.** It aims at introducing uncertainty and increasing the flexibility of the language [99]. If a value of "*" is given to an element, an unlimited number of instances can be created and a feature can be passed over an unlimited number of instantiation levels. The star-potency is a requirement for unbounded models as the number of ontological levels is left open.

- **Range potency.** This was proposed by Macías et al. [154] and originally employed an interval that allows for a higher degree of expressiveness, using the notation min–max. The notation specifies the range of levels below, relative to the current level, where the element can be directly instantiated. Note that the approach allows the second value to be "*" which permits to leave unbounded the maximum level at where the element can be instantiated. While this range is sufficient for attributes, a lack of expressiveness for nodes and relations detected in several scenarios forced them to add a third value [149, 150, 155] to express the *depth*. Thus, the current version of the range potency that is also used in the work presented in this thesis contains three values for nodes and relations (min–max–depth), and the first two for attributes (min–max). The depth value (which can be also specified with "*") is used to control the maximum number of times that the element can be transitively instantiated, or re-instantiated, regardless of the levels where this occurs. This three-valued potency proposal provides control on how much flexibility or strictness one wants to incorporate in a language and allow elements to be controlled in a sensible manner, being able to simulate the different aforementioned realisations of potency.

- **Join potency.** This was proposed by Theisz et al. [229]. While the previous potency specifications provide support to specify a MLM hierarchy within its vertical nature, there are not such mechanisms for horizontal integrity. The authors reason that horizontal integrity may be needed in cases where different models describe different facets of a system that one might want to compose or relate or when there exist domains that could share commonalities. In this context, *join potency* operates in an inter-domain way combining multilevel models of separate technical domains. Join potency essentially defines the specification of the context in which the clabjects from separated multilevel models are combined. Join potency is built upon the basic ideas from both leap and star potencies.

- **Dual deep potency.** This was introduced as part of the Dual Deep Modelling (DDM) approach [176]. It is based on the definition of parallel hierarchies, with different depths, where relations from one to the other can be established. Thus, *dual deep potency* differentiates between source potency and target potency of a property or association. It allows connecting clabjects from different abstraction levels simulating cross-level relationships.

Non-potency approaches only support defining features for instances exactly one level below, i. e., they support shallow characterisation. For example, multilevel approaches purely based on powertypes [49, 178] do not support deep characterisation.

2.4 Multilevel Modelling approaches

Over the past years, there has been a proliferation of MLM approaches. In this section, we survey the most prominent ones and classify them according to the MLM features introduced in previous sections.

2.4.1 *Multilevel Modelling languages*

There exist a plethora of MLM languages with different scopes such as modelling, knowledge representation, and programming. In the following, we explore the most prominent ones.

DeepJava [140] DeepJava is an extension (i. e., superset) of Java that incorporates the concept of potency which can be added to attributes, methods and classes. The fact that it is embedded in Java constrains each element to have exactly one type. It provides methods with potency, but has to use special keywords to navigate up the type hierarchy to find attribute values. Internally, a compiler transforms the DeepJava code into plain Java. Thus, each DeepJava class is translated into a set of Java classes, one for each element with a clabject facet. The compiler also generates code for clabject instantiation at runtime, which is realised using Java's reflective functions.

Nivel [10] Nivel is a metamodelling language capable of defining models spanning an arbitrary number of levels. A formal semantics is given for Nivel by translation to the weight constraint rule language (WCRL) [216] which is a general-purpose knowledge representation language used to create models and constraints. It implements the concept of potency and it adheres to the OCA. It is based on a core set of conceptual modelling concepts: class, generalisation, instantiation, attribute, value and association. Nivel adheres to a form of strict metamodelling and supports deep instantiation of classes, associations and attributes. Its potency definition can be aligned with the classic potency [18].

Gmodel [33] Gmodel is a metalanguage designed to enable the specification and instantiation of modelling languages. The primary goal of Gmodel is to address modularity and extensibility for the specification of DSMLs. It is based on a small number of language elements that have their origin in model theory and denotational semantics. The extensibility allows the approach to be used in a multilevel-based nature and it offers support for an unlimited number of instantiation levels. Moreover, it has interoperability with the EMF.

Dual Deep Modelling (DDM) [176] It is a general-purpose approach to MLM that is oriented especially to conceptual data modelling. It allows specifying modelling hierarchies with a different number of instantiation levels. This approach achieves deep characterisation through dual deep potency. The structure and semantics of DDM constructs are formalised in the Flora-2 variant of F-Logic [248], a mature modelling language with concise syntax as well as metamodelling capabilities. Dual potencies extend deep instantiation with the possibility to indicate the depth of characterisation separately for the source and the target of a property. DDM substantially extends on Dual Deep Instantiation (DDI) [174] which was formalised in F-logic and implemented on top of ConceptBase [114, 115] which is a metamodelling system based on Datalog and the Telos data model [173].

DeepTelos [120, 121, 123] It is a level-blind approach (as it does not explicitly express the notion of level) that extends Telos and allows defining hierarchies of objects via so-called *most-general instances* (MGI). Even though it is level-blind, DeepTelos incorporates a mechanism to simulate levels. DeepTelos is defined by just 6 formulas (5 rules and 1 constraint) and is enabled by creating the DeepTelos objects with additional rules/constraints in ConceptBase [114]. Its customisable power through MGI allows implementing the OCA albeit it is not strictly an OCA-based approach. In fact, there does not exist a dependency on a linguistic metamodel, although DeepTelos does require it to manage potencies, which need a custom set of elements specifically specified to represent the allowed instantiations.

DeepRuby [175] It is an implementation of DDI achieved through an extension of the Ruby language. DeepRuby makes use of Ruby's dynamic programming and metaprogramming facilities [184]. For instance, they take advantage of the so-called *eigenclass* which can be understood as a singleton class. The DeepRuby implementation does not support the notion of level and it can therefore be classified as a level-blind approach.

Flexible Meta-Modeling and Execution Language (FMML^x) [94] FMML^x is a modelling language that extends XCore [50]. A model created with this language may not only include objects in different levels but also intrinsic operations, associations and attributes. FMML^x is enriched by a metamodelling environment that extends Xmodeler [52] which is a metamodelling framework that supplements XMF. XMF is a programming language for language-oriented programming, and that is implemented within the EMF. FMML^x is composed of two main components. First, a generic modelling editor gives access to the creation of metamodels based on a generic notation similar to UML. Second, a concrete syntax editor supports the design of symbols and widgets which are connected through correspondence to the respective elements in the metamodel.

The approach also supports the definition of executable constraints that are specified in XOCL [185], a variant of OCL. FMML^x does not stick to the OCA but it is based on the *Golden Braid* architecture (a concept originally raised by Hofstadter [111] as a recursion metaphor) where a topmost level defines itself, which can be transitively instantiated as many times as required to create a multilevel hierarchy. For potency, they allow marking features as *intrinsic*, which requires the specification of the level where that feature can be instantiated.

UFO-MLT [3, 64] The Multi-Level conceptual modeling Theory (MLT) is founded on a basic instantiation relation and characterises the concepts of individuals and types, with types organised in levels related by instantiation. The basic entities in the theory and the proposed relations between entities are formally defined through axiomatisation in first-order logic. MLT is conceived to focus on conceptual modelling and the authors account only for the ontological nature (and not for the linguistic one). They define their cross-level structural relations as *power types of* relations. In [3] they discuss the two powertype definitions given by Odell [178] and Cardelli [49] (based on the *power set* concept) and clarify that they adopt the latter for their interpretation of powertype.

To overcome some limitations of the MLT approach in [3], such as MLT not being capable of dealing with several general notions underlying conceptual models (including the notions used in its own definition), they extended MLT with MLT* [64]. To improve MLT's generality, MLT* combines a strictly stratified theory of levels with the flexibility required to model abstract notions that defy stratification into levels such as a universal "Type" or, even more, abstract notions such as "Entity" and "Thing".

ML2 [90, 91] ML2 is a textual modelling language for multilevel conceptual models, i.e., those in which classes can also be subject to categorisation, extending beyond the two-level division between classes and their instances. ML2 is developed based on the MLT* theoretical foundation. It incorporates the definitions from MLT* in its constructs, allowing the specification of MLT* based models and includes a UML profile for visualisation. Since ML2 is based on MLT*, it combines the approaches based on powertypes and clabjects. It does not support deep characterisation of instances and positions itself between the level-blind and the level-adjuvant approaches as it does not contain an explicit notion of level, even though it can be inferred through *type orders*.

MLT-Telos [122] MLT-Telos is an implementation of MLT* that allows identifying modelling errors via integrity constraint violations. MLT-Telos constitutes a collaborative work between the two MLT* and DeepTelos approaches, where the authors aim for a combination that leverages the best of both worlds in the MLT-Telos implementation: a rich set of multilevel mechanisms and run-time deduction. They first identify the differences between MLT* and DeepTelos and how they are related, to later unify the two approaches to reduce unnecessary diversity in this research domain.

2.4.2 *Multilevel Modelling tools*

In recent years, various MLM tools have been developed to allow experts across several disciplines to apply the theoretical concepts, work with MLM techniques and provide solutions. In the following, we explore the most prominent MLM tools and classify them according to the aspects discussed in Section 2.3.

DPF Workbench [141] This tool applies MLM employing graph theory and category theory. The DPF Workbench is based on the Diagram Predicate Framework (DPF) [209]. It supports an arbitrary number of levels, but it does not implement an explicit potency. An interesting aspect is that constraints are not specified in a textual language, such as OCL [51], but using graphical annotations into the actual model with formal semantics.

Modelverse [239] It is a MLM framework based on AtoMPM [225] (a framework highly focused on offering cloud and web tools). The tool is developed following the OCA principles and, therefore, it offers MLM functionalities by implementing the concept of clabject and building a linguistic metamodel that includes a synthetic typing relation. Its potency implementation aligns with the classic potency by Atkinson and Kühne [18].

MetaDepth [65, 71] MetaDepth is one of the most established tools for MLM that supports an arbitrary number of ontological metalevels. The framework is integrated with the Epsilon languages [87] for model manipulation. This integration permits using the Epsilon Object Language (EOL) [131] as an action language to specify the behaviour for metamodels, and the Epsilon Validation Language (EVL) for expressing constraints. Both EOL and EVL are extensions of OCL. The authors implement the interface of the connectivity layer in a way to make EOL aware of the multiple ontological levels. Through the integration of the Epsilon languages into MetaDepth, one can take advantage of model transformation techniques, code generation, and multilevel refactoring [68].

Guerra and de Lara also discuss the need to investigate ways to make modelling more flexible in [106]. In this work, they propose techniques to lift the inconsistency tolerance (to be more flexible when, for instance, several iterations over a (meta)model might introduce further changes), information extension, a configurable classification relation and some other aspects (see [106] for the complete list). They implement these ideas in a prototype tool called Kite as an Eclipse plugin based on EMF and Xtext. Also, they discuss how MLM might benefit from some ideas of flexible modelling and they envisage a framework that takes the best of MLM and flexible modelling.

Cross-Layer Modeler (XLM) [76] XLM is a modelling tool that supports MLM by reassembling multiple UML notations and views to create models at different levels and describe their classification relationships. It focuses on the automatic co-evolution of constraints and provides instant feedback about model consistency. Thus, XLM allows the user to modify metamodels and models at the same time, and provides consistency checking that automatically updates constraints to keep them valid after metamodel changes. However, the tool does not support advanced instantiation mechanisms such as potency or inheritance. The semantics of model elements or connections in XLM must be given via constraints, otherwise, they do not have meanings.

Melanee [12, 143] Melanee is a well-known OCA-based tool for deep modelling which supports modelling through deep, multi-format, multi-notation user-defined languages. The tool supports a variant of OCL with deep semantics and has been integrated with the Atlas Transformation Language (ATL) for model transformations. Using this OCL variant (so-called deepOCL [142]), users can check constraints spanning multiple classification levels which can be defined and executed. Although Melanee itself is not supporting natively tools for simulation/execution through the specification of the behavioural semantics, e. g., through in-place model transformation rules [161], there are some works on top of it that aim to achieve this. The work presented in [23] is an example of this, where the model execution mechanism is done through a service API and a plug-in mechanism. The communication between the modelling and the execution environments can be realised using socket-based communication.

Lange and Atkinson also proposed in [144] the notion of *deep substitutability* which expresses that during specialisation of elements (through inheritance) not only attributes and methods should be taken into account for substitutability (as in traditional two-level approaches) but also potency and endurance properties (the latter is called *vitality* in [144]). Deep substitutability is described in the Level-agnostic Modeling Language (LML) [144] supported by Melanee. In that work, Lange and Atkinson discuss the well-formedness rules that need to be applied to the vitality properties of clabjects to ensure deep substitutability in inheritance relationships.

Dynamic Multi-Level Algebra (DMLA) [228, 236] DMLA is a self-validating metamodelling formalism relying on gradual model constraining through its interpretation of the classical instantiation relation. DMLA is self-described, and it also provides the so-called *fluid metamodelling*, which means that it is not required to instantiate every entity of a model at once. Models in DMLA are stored in tuples, referencing each other, and thus, forming an entity graph. As mentioned in Section 2.2, DMLA is not an OCA-based approach but implements a self-describing so-called *Bootstrap*. At the root of the instantiation chain, they have an element called *Base* and all entities are direct or indirect instances of it.

MLM Rearchitecter [150, 153] This tool facilitates the migration of standard metamodels into a multilevel setting. To give automatic support for the rearchitecture process, Macias et al. [150] establish a workflow where first, original metamodels are annotated to highlight occurrences of multilevel modelling *smells*. Secondly, the metamodel is automatically transformed into a multilevel neutral representation that can be later accommodated into some of the MLM approaches [150].

In [153], Macias et al. build upon the work presented in [150], where they improve the tool-agnostic metamodel to represent neutral multilevel hierarchies and enhances the bidirectional transformation between such representations and some of the well-known MLM frameworks. The three tools considered in [153] are MultEcore, Melanee and MetaDepth. With such a bidirectional transformation, comparison and interoperability between the involved tools are possible and experimental evaluations can be carried out.

Open Metamodeling Environment (OMME) [241] The OMME is an OCA-based approach that implements dual ontological/linguistic typing and consists of a set of plugins for the EMF. It focuses on the specification of DSMLs and allows for creating models with an arbitrary number of meta layers. It interprets concepts such as (extended) powertypes, deep instantiation, materialisation and clabjects. These concepts are defined in the so-called Linguistic Meta Model (LMM) that aims at reducing redundancies and increase the expressiveness of models and modelling languages.

Open integrated framework for MultiLevel Modelling (OMLM) [112] OMLM is an OCA-based MLM framework that focuses on the field of knowledge representation and on addressing multilevel modelling semantics, implementing and verifying them in Flora-2. It implements single- and multi-potency. An interesting aspect is the incorporation of a new dimension called the *realization dimension* to capture the mapping of the multilevel metamodel to an existing programming language. This provides multiple advantages such as decoupling the multilevel framework from the implementation language, comparing implementations in different programming languages, and automating code generation. OMLM also contains a verification framework called MULti-LEvel Reasoner (MULLER) that aims to verify the correctness of MLM properties using verification rules. Another feature it integrates is the possibility to transform two-level models into multilevel models.

TOTEM [113] This tool aims to bridge the gap between the two-level and the MLM areas. It is an Eclipse-based implementation that employs a multilevel metamodel that is capable of annotating a two-level metamodel with multilevel constructs. The two main components of the tool are the multilevel annotation tool and the MLM tool. The tool implements two mechanisms to create a full multilevel hierarchy. In the first one, an existing EMF metamodel is annotated with multilevel annotations and then instantiated using the MLM tool. This mechanism enhances compatibility with EMF and facilitates migration into a multilevel architecture. The second mechanism consists of a language designer. It does not start from an existing EMF metamodel but directly uses the MLM tool to create the top-level model of the language.

2.4.3 *Classifying Multilevel Modelling approaches*

In the past years, there have been published several research works that aim at comparing MLM approaches. This comparison is not a trivial task, since there exist a lot of different aspects that are particular to each approach, and finding a common framework for comparison is a complex task, especially for quantitative measures or advanced qualitative metrics, such as reusability or flexibility. Indeed, Atkinson and Kühne stress in [13] that first, a key matter is to define what features an MLM approach needs to possess in order to be considered as a multilevel approach. In their article, they also describe some fundamental aspects that should serve to classify the different MLM approaches. Gerbit et al. carried out a comparison work in [100] between Melanee and Metadepth which were the most MLM mature tools. In their work, they made the comparison based on features, i. e., based on how each approach handles e. g., potency, attributes, and classification semantics. The work presented by Jácome and de Lara [113] compares and classifies several MLM approaches using a feature model to present the different alternatives. They use several classification criteria (some of them are also described in [13]) and they seek to identify gaps and opportunities for improving the current state of the art. Macias [149] also made a comparison of different approaches based on their support of some of the potency variants described in 2.3.2.

Still, there is a lack of quantitative and qualitative comparison works in the MLM community. The fact that most of the approaches are quite different between them makes it difficult to find a common ground to evaluate qualitative metrics, such as performance, efficiency, reusability or flexibility, and even more difficult to carry out quantitative comparisons. However, the MLM community is undoubtedly moving forward, and events such as the MULTI Challenge that have taken place for several years [4, 5, 60] are encouraging the different MLM proposals to work towards the same goals. This is helping to compare approaches where the proposed case study is common while analysing the solutions becomes easier. Furthermore, other works have emerged with the aim to establish different measures for comparison. For example, Kühne and Lange [139] present so-called *deep metrics* as an approach to quantitatively measure high-level model concerns of multilevel models, such as understandability and maintainability. While their proposal is still theoretical, it is clear that future work in this direction is going to be key to objectively compare MLM approaches.

2.5 Multilevel Modelling in MultEcore

MultEcore is a set of Eclipse plugins which aims at combining the best from the traditional two-level modelling and MLM: the mature tool ecosystem and familiarity of the former, and the expressiveness and flexibility of the latter. In addition to conceptual modelling, MultEcore also facilitates the definition of multilevel DSMLs. That is, each level is interpreted as the metamodel of a DSML, and in this way providing support for reusability and modularity and the capability of creating related languages on demand. In the following sections, we explore MultEcore by describing how it handles the structure and semantics of multilevel DSMLs. We refer the reader to [149, 152] for additional details on MultEcore.

2.5.1 *Structure*

The structure of a MLM language in MultEcore is described by a *multilevel modelling hierarchy*. By a MLM hierarchy, we understand a tree-shaped hierarchy of models with a single root one typically depicted at the top of the hierarchy tree. This root model is fixed and self-defined, as MultEcore is not an OCA-based approach but uses the golden braid architecture instead. For implementation reasons, we use Ecore [221] as root model at level 0 in all example hierarchies, since Ecore is based on the concept of graph which makes it powerful enough to represent the structure of software models. Thus, hierarchies enclose a set of models which are connected via typing relations. A hierarchy has $n + 1$ abstraction levels, where n is the maximal path length in the hierarchy tree. Levels are indexed with increasing natural numbers starting from the uppermost one, having index 0. Each model in the hierarchy is placed at some level i, where i is the length of the path from that model to the topmost one.

Figure 2.2 shows a simple multilevel hierarchy containing four levels of abstraction with Ecore at the top of it (Figure 2.2 (a)). At **Level 1**, we branch into two paths. The models *generic-model-1* and *generic-model-2* (Figures 2.2 (b) and 2.2 (c), respectively) contain three nodes and one relation each. As shown in the figure, the type of a node is indicated in an ellipse at its top left side, e. g., **EClass** is the type of **A**, **B**, and **C** in model *generic-model-1*, as well as of **D**, **E**, and **F** in model *generic-model-2*. The type of an arrow is written near the arrow in an italic font, e. g., **EReference** under **G** in model *generic-model-1*, and under **H** in model *generic-model-2*. The dashed arrows in the figure between levels indicate *typing* relationships and are formalised as the concept *typing*

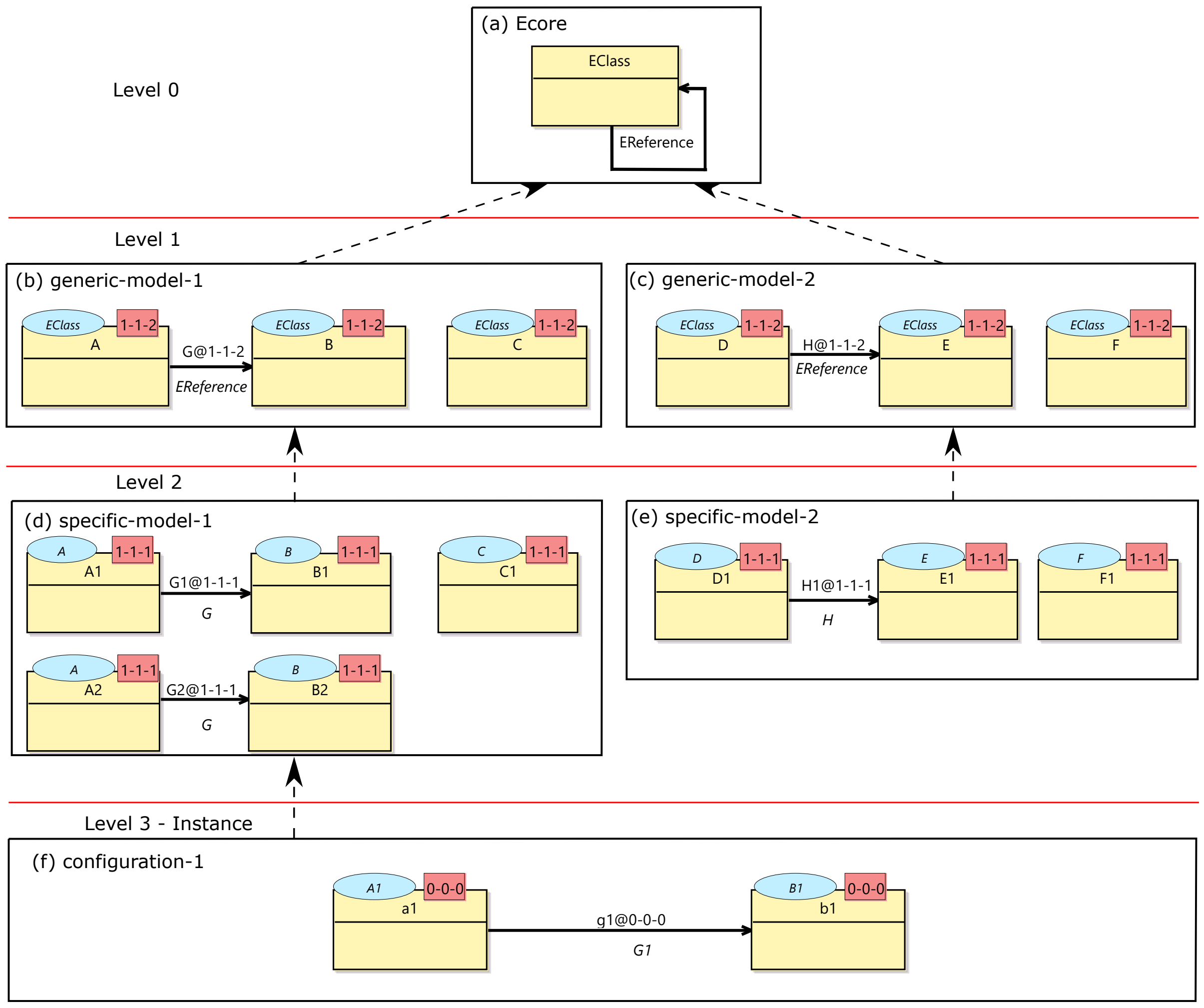


Fig. 2.2: Multilevel hierarchy for a conceptual example

chain in the MLM hierarchy (see [149, 155, 245] for the formal treatment of typing chains).

The potency implementation of MultEcore combines range and start potencies which are specified via three numbers for nodes and relations and two numbers for attributes. In Figure 2.2, potency is displayed in a red box at the top right of every node and concatenated to the name after "@" for every reference. This potency specification allows to precisely define the degree of flexibility and restrictiveness of the elements within the multilevel hierarchy. The first two values, *start* and *end*, specify the range of levels below, relative to the current one, where the element can be directly instantiated. The third value, *depth*, is used to control the maximum number of times that the element can be transitively instantiated, regardless of the levels where this happens. That is, the number of times an instance of that element can be re-instantiated. In the hierarchy example in Figure 2.2, these two values are always 1 (except the model at level 3), meaning that the element can only be instantiated in the level right below. A potency value of, for instance, 2 – 4 – 2, would mean that an element can be directly instantiated two, three and four levels below the one where the element is defined and it can be re-instantiated, at most, two times. Since attributes can be instantiated only once, it does not make sense to create an instance of such instance. The depth on attributes is therefore always 1 and it is not modifiable. In practical terms, only the first two values (start and end) of the potency are available to the user.

We use levels as an organisational tool, where the main rationale for locating elements in a particular model is based on how they could potentially define an independent modular artefact. In this regard, we encourage the *level cohesion* principle [138], that is, we recommend organising elements that are semantically close (employing potency and level organisation). On the contrary, we do not promote the *level segregation* principle [138], which establishes that level organisational semantics should be unique, i. e., aligned to one particular organisational scheme, such as *classification* or *generalisation*. Still, we generally use typing relations with classification semantics, and the typing relation implies that a node defines which attributes its instances can instantiate and which relations it can have to other nodes. Furthermore, the MultEcore tool checks correct potency and typing safeness. Typing safeness is checked via internal constraints that forbid typing relations to be circular, reversed or inconsistent either vertically (i. e., within the same hierarchy) or horizontally (i. e., if we consider more than one hierarchy). Further details on how we achieve these two kinds of flexibility are given in Paper C.

The rest of the models in Figure 2.2 are straightforward, but notice for the elements at the bottommost model (level 3), **a1**, **g1** and **b1** in *configuration-1* (Figure 2.2(f)) which is an instance model of *specific-model-1* (Figure 2.2 (d)), the potency is set as 0 – 0 – 0. This is used to enforce that elements at the bottom level are used purely as instances, which cannot be refined further at lower levels.

Supplementary Hierarchies The use of more than one dimension to describe a system or language either because the system is complex and we need several facets to fully describe it or because it might benefit from additional orthogonal aspects has been widely explored in the literature from different perspectives, e. g., [35, 72, 81, 92, 249] (we further analyse this in Chapter 3). Motivated by these ideas, we have extended the notion of multilevel hierarchy in such a way that multilevel hierarchies can be supplemented by other multilevel hierarchies. Frequently, we denote a multilevel hierarchy as the *main* or *default* one and call it the *application hierarchy*, since it represents the main language being designed. An application hierarchy can include an arbitrary number of *supplementary hierarchies* which add new aspects to the application hierarchy. Indeed, *supplementary* and *application* are simply names, i. e., a hierarchy can act as application in a context and be supplementary of another one in a different context.

Supplementary hierarchies can actually represent proper languages rather than additional features that are incorporated into the main language. Adding or removing supplementary hierarchies is made possible by the incorporation or extraction of additional typing chains (see [245] and [201] for the formal details of typing chains). For instance, we might have different hierarchies (physically separated, e. g., different projects in the MultEcore tool) that we want to compose. This can be achieved by assigning the role of application hierarchy to one of them and adding the rest as supplementary ones.

In our approach, it is not only convenient to use supplementary hierarchies to achieve composition but it is also practical and useful. Note that the notion of multi-typing is well-known, for instance, in OCA-based approaches where elements are often given one linguistic type and one ontological type. In MultEcore, supplementary hierarchies can be used to multiple-type elements with external Data Types [202] and to boost the characteristics of certain elements that require additional types [204]. We show in Paper F how we can provide every element that belongs to the main hierarchy with supplementary information in a flexible and reusable way. We further explore composition in Chapter 3.

2.5.2 Semantics

The semantics of a modelling language assigns a precise meaning to each of its language constructs. As Méndez-Acuña et al. [160] analyse, the *static semantics* reason for the correctness of the structure of a modelling language (typically employing constraints), while the *dynamic semantics* specify the behavioural descriptions. Among the existing methods to specify the behavioural descriptions (which are detailed in Chapter 4), we are interested in *operational semantics*, whose executions change the instance model states.

Transformation rules can be used to represent actions that may happen in the system. There exist many applications of model transformations (e. g., to transform models from one modelling language to another via exogenous model transformations [83], to automatically generate code (model-to-text), and some others [63]). To express behaviour, a well-known mechanism is the use of (endogenous) in-place model transformations (MTs) [161]. These are rule-based modifications of a source model specified in the left-hand side of the rule resulting in a new state of such a model determined by the right-hand side of the rule.

Multilevel Coupled Model Transformations (MCMTs) is a multilevel transformation language that exploits the multilevel capabilities of MultEcore. They overcome the issues of both traditional two-level transformation rules and the multilevel model transformations. While the former lacks the ability to capture generalities, the latter is too loose to be precise enough (see [149] for details). MCMTs can be used either to express the static semantics of a hierarchy as constraints (see Paper F for an example of an MCMT rule that checks structural correctness) and the dynamic semantics to describe the behaviour (see Paper C and Paper E).

In MultEcore, an MCMT rule has the form of $LHS \Rightarrow RHS$ *if* C, where *LHS* and *RHS* are multilevel modelling patterns and C is a an application condition which constraints the cases the rule may be applied. Given a model M that represents a state of the system, if there is a match of the rule's *LHS* in the model M, and its condition C is satisfied, then this match is replaced by the rule's *RHS*. All the formal details on MCMTs can be found in Paper D and [245].

Figure 2.3 shows a simple MCMT rule (called *Add and Connect*) that models the creation of a new node and a relation between the existing node and the new one. We discuss at the end of this section other advanced features that MCMTs support. The **FROM** and **TO** blocks describe the left pattern and the right pattern of the rule, respectively. The **META** block depicts a multilevel pattern allowing us to locate types at any level that can be used as individual types for the items in the **FROM** and **TO** blocks, respectively. Notice that the **META** block facilitates the definition of an entire multilevel pattern, and therefore, we can specify several **META** levels within such a block. This leads to a more natural way of defining that a type is defined at some level above, without explicitly stating in which level. In fact, this also promotes flexibility in case of future modifications to the number of branches (horizontal dimension) and the depth (vertical dimension) of hierarchies. Details on how we achieve flexibility in both dimensions are given in Paper C and Paper F.

At the top level of Figure 2.3, we mirror parts of *generic-model-1* (Figure 2.2 (b)), defining elements like **A**, **B** and **G** as constants. We differentiate constants as their names are underlined and their types are not specified via the ellipse above (for nodes) or the italic text (for references). A constant node in an MCMT rule can only match a node in the hierarchy with the same name in the matched level. For a constant edge to match an edge in the model, its name and the names of its source and target nodes must match the corresponding names in the model. The use of constants constrains the

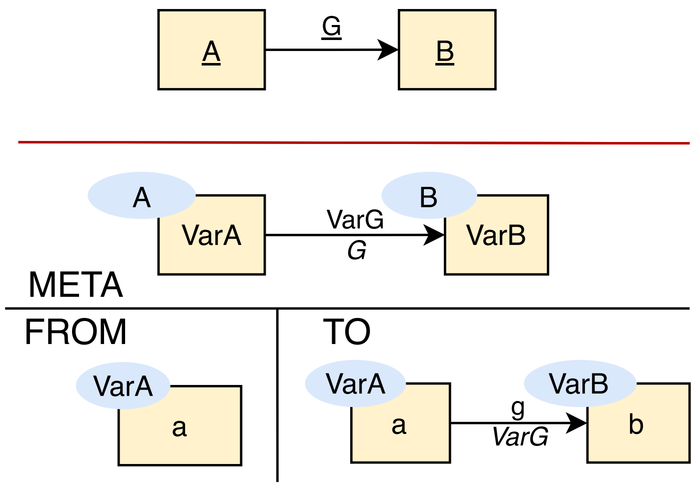


Fig. 2.3: Rule *Add and Connect*: The execution of this rule gives a new state on the model where a new node is created and connected to the existing one

matching process, significantly reducing the number of matches. When the element is a variable, the match is based only on finding the right structure in the model. This opens for the definition of generic rules which are applicable to various structurally similar hierarchies in which the elements have different names. The rule depicted in Figure 2.3 can be applied to models (instances) typed by the left-hand branch Figure 2.2 (i. e., *specific-model-1*, *generic-model-1*, *Ecore*). Note, that the horizontal lines do not enforce consecutiveness between the levels specified in the rule with respect to the hierarchy. For instance, **VarA** (placed at the second level of the **META**), and typed by the variable **A**, would match any node which indirectly has **A** as type, or ultimately will match to **A** if no indirect one is found. A correct match of the rule comes when an element, coupled together with its type, fits an instance of **VarA** (e. g., **a**, located in the **FROM** part). Given the current state of the hierarchy in Figure 2.2, any instances of elements matching the pattern **VarA** would be candidates to perform the transformation. This in turn makes it possible to apply the rule to either instance of **A1** or to instances of **A2** (these elements are defined in the model *specific-model-1* in Figure 2.2 (d)).

The MCMT rule shown in Figure 2.3 is rather basic and do not exhibit many of the available features. The current version of the MCMTs allows the specification and manipulation of attributes, application conditions, submodel patterns and the possibility to add language functionalities such as OCL and SML. Submodel patterns are handled with a nested parametric boxing mechanism. Specifying boxes allows us to define model patterns where its expansion would result in a collection of elements. There exist similar approaches to this, such as the collection operator [104] and the star operator [148]. Some applications of the features listed above can be found in Paper C, Paper E and Paper F. Also, an example of how boxes are used can be found in Chapter 5. More details on the available functionalities are given in Chapter 6. For the detailed documentation of MultEcore, we refer the reader to [151, 192].

CHAPTER 3

LANGUAGE COMPOSITION

In this chapter, we discuss several composition techniques that are described in the literature. First, we analyse the structural dimension, then we describe how different approaches handle the composition of behaviour, and finally, we present our approach.

3.1 Composition in MDSE

MDSE promotes the use of separate models to tackle the different concerns in the development of complex software systems [213]. Building large systems involves constructing several modules that describe each of the systems' parts. However, these modules often have to be composed in order to reason about the global properties of the entire system [129]. *Composition* has been widely studied in the literature and there exist several definitions of composition or *composability*. As stated by Mussbacher et al. [171], for some, composition is an operation that is performed on larger modelling units but not at finer levels of granularity. Others may view composition as establishing relationships between modelling elements. Some researchers define composition as the act of creating new first-class entities of the modelling approach from existing ones, e. g., by putting together several units of encapsulation [171]. A composition specification can be unfolded either as a *composition rule* or a *composition operator*, but any of them are applied to some input model elements and describe some output. A composition rule gives the specification of the composition but does not perform any composition procedure on the actual models. A composition operator, on the other hand, results in a composed model, e. g., a merge operator actually merges the two model elements into one [171].

Another fundamental aspect that has encouraged the development of composition mechanisms is the fact that, although each existing DSML is unique and has been developed for a specific purpose, not all of these are completely orthogonal. Recent research efforts have shown the existence of DSMLs providing similar language constructs [251]. In this direction, the research community in software language engineering has proposed the notion of *Language Product Lines Engineering* (LPLE) to construct software product lines where the products are languages [160]. The key aspect of these approaches is the definition of *language features* that can be interpreted as modular pieces that encapsulate a set of language constructs representing certain DSML functionalities. Usually, one can detect that some DSMLs share certain commonalities coming from similar modelling patterns and that these can be abstracted and reused across several other languages. The concept of feature is, therefore, directly related to the idea of modularisation. Indeed, the constructed frameworks that aim to enhance reusability and extendibility should naturally handle modularisation and composition techniques. A framework with such characteristics would allow to easily separate concerns by organising the description of the system in a modular way (e. g., features), which could be further composed.

Modularisation is a property that is naturally favoured in Aspect-oriented programming (AOP) techniques [128]. Indeed, many standard modelling notations have been extended with aspect-oriented mechanisms to support an advanced separation of concerns (e. g., Kompose [92], HiLA [249] and AoUCM [172]). This is called Aspect-oriented modelling (AOM) where the structure of the modelled system is described through a primary model and one or more aspect models that complement the primary one with additional features [92].

3.2 Structure composition

In general, model composition unfolds along two dimensions, structure and behaviour. We describe in this section different techniques and approaches that handle structure composition and discuss the behavioural aspect in Section 3.3. Méndez-Acuña et al. [160] distinguish between two modularisation techniques to support structural modular language designs:

- **Endogenous modularity.** The bindings between language modules are defined internally in the modules themselves, i. e., the information regarding how the modules are related is encapsulated in one of them. While this internal way of describing the modularisation components enhances maintainability, there is no need to keep up additional software pieces that describe the composition details. Hence, it may pollute the model with redundant information.

- **Exogenous modularity.** The bindings between the different modelling modules are defined externally. In this case, the language modules do not contain any information regarding composition with other components but an external artefact instead holds the information. As there are no direct references between language modules and the corresponding model is not polluted with referencing information, the bindings can be changed in the external artefact without modifying the modules themselves. While this promotes modularity and separation of concerns, the modeller has to maintain such an additional artefact together with the rest of the involved modules.

In the next subsections, we explore the most relevant composition operators and classify them according to the two modularisation techniques discussed above.

3.2.1 *Merge operator*

One of the most used techniques for structure composition is through the implementation of *merging* operations. Merging is a symmetric operation, i. e., the result of the merge does not depend on the order of the input models. Due to the capability of merging to integrate independent artefacts, it is generally used by approaches based on exogenous modularisation. Intuitively, merging refers to the operation in which "the common elements are included only once, while the rest are preserved". Formally, a merge combination operator takes two metamodels, *Metamodel 1* and *Metamodel 2* as inputs, as well as a set of correspondence tuples $C = \{\langle e_x, e_y \rangle, \ldots\}$ with $e_x \in$ *Metamodel 1* and $e_y \in$ *Metamodel 2* (being e_x and e_y elements, i. e., nodes or relations). The merge combination operator produces a new output *Merged Metamodel* that contains, for each tuple $\langle e_x, e_y \rangle \in C$, a single metamodel element. All metamodel elements in *Metamodel 1* and *Metamodel 2* that are not given a correspondence in C are simply copied into the *Merged Metamodel*. Note that composition can be defined at different levels of granularity. The definition given above considers a low level of granularity that takes into account each model element. In other approaches, it is at a higher level, as in [129], where the scope level is "at the composition of structural and behavioural models that represent broader concerns of interest to stakeholders". Some approaches that use the merge operator are Melange [75] and Gromp [159].

3.2.2 *Weaving operator*

Another alternative for composition is the use of *weaving* operations. Weaving is an asymmetric procedure as it involves two different actors: an *aspect* and a *base* model [157]. The aspect is made of two parts, a *pointcut*, which is the pattern to match in the base model, and an *advice*, which represents the modification made to the base model during the weaving. The parts of the base model that match the pointcut are called *joinpoints*. During the weaving, each joinpoint is replaced by the advice. The definition of weaving allows it to be applicable in both endogenous and exogenous modularisation, as the binding between the aspect and the base module can be defined either in the aspect itself or in an external artefact.

Many approaches that formally specify their composition implementation make use of category theory and graph transformations (which can be applied both to merging and weaving). An example of the weaving technique is formally described and implemented in [81]. Other approaches that implement a weaving operator are GeKo [135], MATA [243] and the Atlas Model Weaver (AMW) [35].

In the field of Multimodelling, Stünkel et al. [224] uses exogenous weaving and linguistic extensions (see Section 3.2.4). Multimodelling is addressed by a construction that yields a comprehensive model which contains correspondences between the involved models. To define the relations between the different models in the multimodel, a linguistic metamodel is used.

3.2.3 *Inheritance operator*

Inheritance is a mechanism coming from object-oriented programming to enhance reusability. Approaches that use inheritance as a composition operator are based on endogenous modularity. This is because the nature of the inheritance relationship intrinsically relates elements defined in or accessible from the same model project. Thus, through inheritance one can reuse the specification provided in concrete implementation artefacts for which *direct linking* results useful. Note that in direct linking, all the content of the referenced artefact is included as part of the referencing one which allows seeing the complete resulting language as a unique specification. LISA [162, 163] and MontiCore [133, 134] are some approaches that integrate the inheritance operator.

3.2.4 *Linguistic extension*

In the context of flexible modelling, linguistic (dynamic) extension [72] is influenced by role-based modelling [27, 220] where objects can acquire and drop roles dynamically. It however comes with some shortcomings implying that role-based approaches have to introduce additional mechanisms to describe how roles are assigned or related and that everything must be done a priori using further constructs that pollute the model.

Linguistic extension or facet-oriented modelling [72] is a similar approach to AOM that allows slots, constraints and types to be added or removed from objects dynamically using *facets*. Facets that are defined in an existing metamodel can be added or removed from already existing objects. Facet-oriented modelling is supported and implemented on top of Metadepth [65]. We further discuss this approach and compare it with ours in Section 3.4 as there exist several similarities.

3.3 Behaviour composition

In this thesis, we only consider behaviour definition using model transformations (MTs) (see Chapter 4 for details on MTs). We describe some techniques and approaches that focus on the composition of MT rules in Section 3.3.1. Then, in Section 3.3.2, we explore other techniques that have been used to achieve composition acting at the behavioural model level.

3.3.1 *Acting on model transformations*

One of the common ways to achieve the composition of MT rules is the construction of amalgamated versions of the rules. Usually, each model system (or each multilevel hierarchy, in an MLM context) holds certain MT rules that specify its behaviour. Then, composing the MT rules means to create *amalgamated* versions of the MT rules that capture each individual intention. Informally, an amalgamated rule contains the common action and, additionally, all actions from the elementary rules that do not overlap [74]. Amalgamation can act between symmetric systems (i. e., different functional modelling systems in which all components are treated as first-class) or between asymmetric systems in which "aspects" are woven into "components" that implement a base model analogously as in the weaving operator for structure composition (see Section 3.2.2). Some approaches that support the amalgamation of MTs, which are founded on graph transformation are: GROOVE [190, 191], AToM^3 [73, 74], the GReAT tool [29], the Amalgamation Theorem [39], Taentzer's work on parallel graph transformations [226], the Multi-Amalgamation approach [37] and the DPF Workbench [141].

Another alternative to achieve the composition of MT rules is via composite MTs using, for instance, distributed graph transformations [125]. Jurack and Taentzer [125] consider composite transformations as partial mappings of composite models (i. e., a set of component models which are interconnected) that can describe the major effects of model transformations. Such effects can be the creation, deletion or update of model elements and their references. There exist different composite MT classes, such as component MTs, synchronised MTs and model reconfiguration (see [125] for details). Composite MTs are characterised by so-called "synchronisation points", i. e., time points where all component transformations have finished. Starting at some composite model, several component transformations may take place in parallel [125]. Some approaches that support composite MTs by using distributed graph transformations are [102, 125, 130, 186].

3.3.2 *Acting on behavioural models*

One can also achieve behavioural composition by acting directly on the behavioural input models. One way to achieve this is by creating a new behavioural model that specifies a particular interleaving of the input models' behaviours, as described by Kienzle et al. [129]. They describe their behavioural weaver approach to achieve the composition of behavioural models through the *asymmetric event scheduling* operator. In concrete, they apply their event schedule operator to compose sequence diagrams, state diagrams, and Aspect-Oriented Use Case Maps (AoUCM) [172].

Coordination has also been widely used in the literature to synchronise behavioural models or components [7]. One way to achieve coordination is by establishing some form of communication mechanism, such as shared memory or message passing. Coordination can be conducted endogenously, or exogenously [181]. In endogenous coordination models, the primitives that cause and affect the coordination of an entity with others can reside only inside such an entity (e. g., Linda [98]). For instance, this is the case for models based on object-oriented message passing paradigms. In exogenous coordination models, the primitives that cause and affect the coordination of an entity with other entities are encapsulated externally (e. g., Reo [8] and MANIFOLD [40]). Exogenous coordination models allow third parties to orchestrate the interactions. An underlying exogenous coordination model is essential a component model in which components are building blocks that are (dynamically) composed together by other entities. Other approaches that support exogenous coordination are CoorMaude [211] and BCOoL [146].

3.4 Composition in MultEcore

In MultEcore, we also unfold composition along structure (via multilevel hierarchies) and behaviour (via MCMT rules). We explore both aspects in the next sections.

3.4.1 *Composition of Multilevel Hierarchies*

To compose multilevel hierarchies, we use the *supplementary hierarchies* technique (introduced in Section 2.5.1) that allows model elements to acquire multiple types. The supplementary hierarchy mechanism can be classified as endogenous modularity (see Section 3.2) since the information is encapsulated within the *application hierarchy*. However, MultEcore facilitates the incorporation and deletion of supplementary hierarchies enhancing modularisation, reusability and maintainability. This provides the advantages which are usually associated with exogenous modularity. The process is automatic, and the modeller does not need to worry about the internal information that is kept in the application hierarchy, as it is hidden from the user. Supplementary hierarchies represent an alternative approach to those exposed in Sections 3.2.1- 3.2.4 and share similarities to the merge, weaving and linguistic extension operators.

A crucial shortcoming present, for instance, in the merge composition approach is the loss of the original elements that have been merged (see also [224] for further shortcomings related to constraint checking). This capability might be useful in several situations, specially when the elements that are being merged are not identical, but powering up each other. For example, we might have two nodes, *Worker* and *Human* with some attributes specifying characteristics of each domain, such as *profit* and *stamina*, respectively. Merging *Worker* and *Human* would create a new single node, with the cost of losing the two separate nodes and the possibility to use them in isolation in other parts of the model.

With the supplementary hierarchies technique, we can use the individual elements as well as use the composed one, increasing the number of resources available for the modeller. Supplementary hierarchies can then be used to compose different languages, or as a way to add additional features not strictly related to the main language. This is similar to model weaving where the supplementary aspects can be seen as complements of the main language. Then, elements at the instance level can have as many types as hierarchies are being composed where each type can be seen as an aspect. We say that we achieve a *virtual composition*, rather than a *physical composition* (as the techniques described in Section 3.2). Virtuality refers to the capability of dynamically adding and removing new types to elements in a non-intrusive way. This can be seen as an aspect-like mechanism that we can use on-demand, being able to use aspects independently or in combination.

As mentioned above, the supplementary hierarchies idea presents some advantages w.r.t the merge and weaving operators and shares similarities with the linguistic extension approach. We have observed that our approach can answer the motivational points given by de Lara et al. [72]. They highlight why other approaches (e. g., merge) do not support such modularity and reusability. Indeed, we have corroborated that the supplementary hierarchies approach fulfils the four requirements defined by de Lara et al. [72]:

- *R1: Be modular and non-intrusive, so that there is no need to create or change existing models or metamodels.* In our approach, we support this by automating the process of incorporating and removing typing chains. In other words, we can directly extend the original language without polluting the models or without being forced to create additional artefacts.

- *R2: Allow objects to acquire new types, slots and constraints, likely specified in other metamodels, and which become transparently accessible.* Incorporating a supplementary hierarchy automatically allows using the elements defined on it. For instance, elements in the application hierarchy can use the newly available types. Also, the attributes that belong to the nodes that are defined in the included supplementary hierarchy are automatically available to be instantiated in the elements of the application hierarchy. We refer the reader to Section 4.3.3 of Paper F for illustrations of this process.

- *R3: Support both manual and automatic acquisition and loss of types, slots and constraints.* In our approach, one can remove the additional types of elements manually one by one, or remove the entire supplementary hierarchy, automatically getting rid of all the supplementary types and attributes that come from such a dimension. Also, MCMT rules can be written to add or remove these types as part of a model transformation (see [204] and Paper D for more details on this).
- *R4: Specification and automatic maintenance of relations (e. g., equality) between owned and acquired slots.* In [72], the authors include mechanisms to establish, for instance, equality between attributes, to avoid repeating attributes representing the same information, but, for instance, we can decide whether an attribute is instantiated or not. This allows us to solve problems such as attributes representing the same information being duplicated, by just instantiating one of them.

We also see similarities at the MT level, as de Lara et al. [72] have a domain-specific transformation language to manipulate facets, which is very similar to one of the

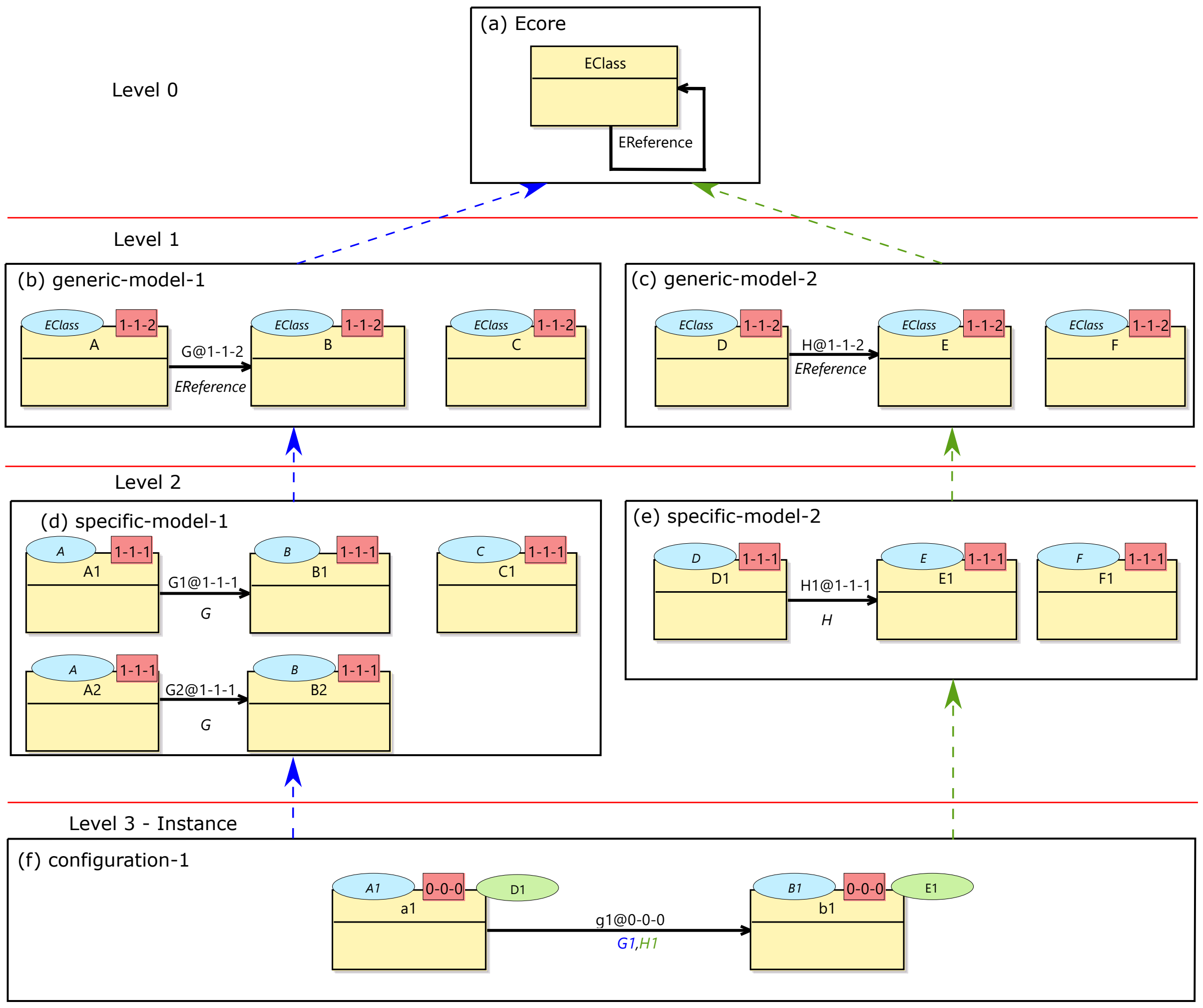


Fig. 3.1: Multilevel hierarchy with two typing chains

applications of the MCMTs in this regard. In conclusion, the facet-oriented modelling approach shares several similarities with our approach based on multiple typing and addresses similar problems from a different perspective.

In MultEcore, when composing different modelling languages, we can consider both working with several hierarchies and with several branches within the same hierarchy. Figure 3.1 displays the hierarchy shown in Figure 2.2 where we now distinguish two different typing chains for the model at Level 3. In this case, we specify one typing chain for each branch in the hierarchy: the left-hand branch where typing relations are shown as blue dashed arrows and the right-hand branch where typing relations are shown as green dashed arrows. Once the new typing chain is incorporated, all the elements of the model *configuration-1* will be (in addition to the original blue typing chain) also part of the green typing chain. The modeller may then use types from both the green and the blue typing chains in the model *configuration-1*. Details of this multi-typing procedure including the resolution of potency conflicts arising when adding supplementary hierarchies are given in Paper D.

The model *configuration-1* in Figure 3.1(f) shows an example of how elements may be multi-typed. One can see that node **a1** has two types associated, **A1** from the left-hand typing chain, and **D1** from the right-hand typing chain, which adds additional information to the node. We have a similar situation with reference **g1** and its two types **G1** and **H1**.

3.4.2 *Amalgamation of MCMTs*

When composing two or more modelling languages, we have to also to take into account the behavioural descriptions of each of the languages. A priori, each multilevel hierarchy would have an associated set of MCMT rules that describe the behaviour of each language. If such hierarchies are composed, it is also natural that their rules are amalgamated as well. Our approach makes use of the amalgamation of MT rules (as described in Section 3.3) to achieve behaviour composition. We focus on creating amalgamated versions of the MCMTs that describe entire execution steps taking into account each individual behavioural aspect. In other words, the application of the amalgamated MCMT rule on the multi-typed instance model creates a new state of the model where the behaviour of each MCMT rule of the composed hierarchies has been taken into account.

Since we use graphs to formalise models, we employ graph transformation rules to express the operational semantics of multilevel models. A graph transformation rule is defined by a left L (described by the **META** + **FROM** blocks in MCMTs) and a right R patterns (described by the **META** + **TO** blocks in MCMTs). In graph transformations, there exists a third component I that can be used to collect the whole context between L and R (i. e., the union of L and R) for those approaches that are based on the co-span version of graph transformation rules [84]. We refer the reader to Paper D for the formal details of MCMTs, which makes uses of the co-span version.

An essential step to achieve amalgamation (or, in general, composition) is the identification process, which is captured by the I component, where the elements that correspond to each other have to be identified.

We assume that the user provides the correspondences between elements in the

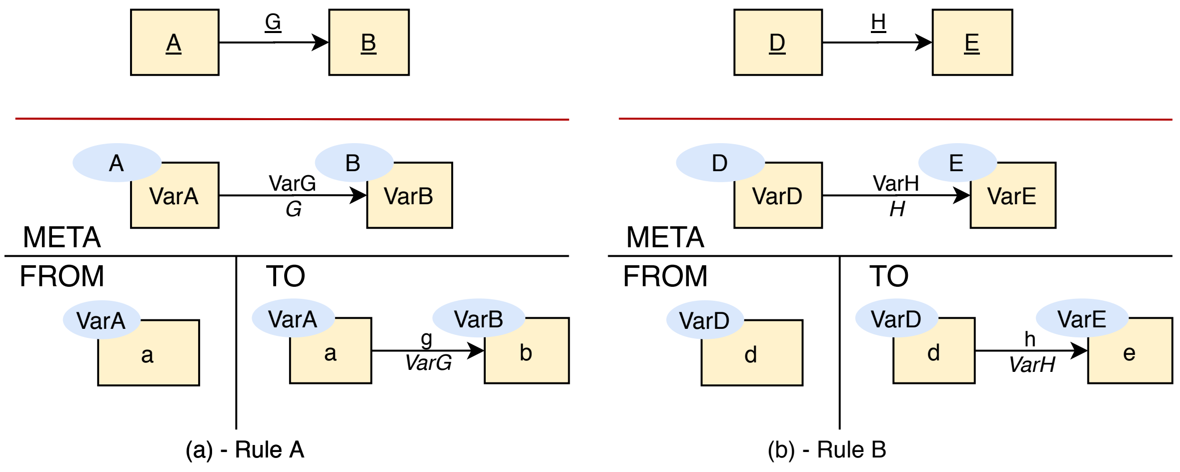


Fig. 3.2: MCMT rules candidates for amalgamation

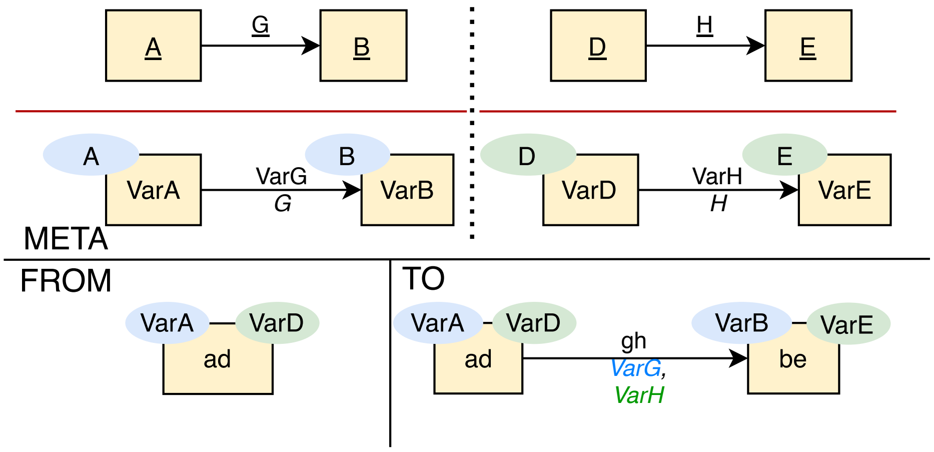


Fig. 3.3: *Rule M*: Amalgamated MCMT rule as result of composing *Rule A* and *Rule B*

rules which are to be amalgamated. We illustrate in Figure 3.2 two simple MCMT rules, namely *Rule A* and *Rule B*, that describe the behaviour of the right- and left-hand branches of Figure 3.1, respectively. In this example, we identify **a** of type **VarA** with **d** of type **VarD**, **g** of type **VarG** with **h** of type **VarH** and **b** of type **VarB** with **e** of type **VarE**. Elements **a**, **g** and **b** belong to *Rule A* (Figure 3.2 (a)) and **d**, **h** and **e** belong to *Rule B* (Figure 3.2 (b)), respectively. Once this identifications have been established, we can create the amalgamated version of the rules. In the amalgamated rule each element has two types, as shown in Figure 3.3. We identify in the **META** block both multilevel hierarchies (note they are separated by a vertical dotted line) involved in the two typing chains present in the **FROM** and **TO** blocks, as product of the amalgamation process. We have a single element in the **FROM** block called **ad** which types are **VarA** and **VarD**. In the **TO** block, we find **gh** typed by **VarG** and **VarH**, which connects **be** typed by **VarB** and **VarE** with **ad**. Employing the amalgamated rule on a multi-typed model, as the one shown in Figure 3.1, would produce a new state of the model where the amalgamated behaviour is applied.

The rule shown in Figure 3.3 represents a simple amalgamation scenario. There exist several cases depending on how elements are identified. These cases are summarised in Table 1 of Paper D. Some of these cases might present conflict situations which happen when, for instance, one identifies two elements, **a** and **d**, and while in the first rule **a** is being connected to a new element in the **TO** block, **d** is being deleted in the second rule.

The composition of the multilevel hierarchy and the amalgamation of the MCMT rules happen in MultEcore, and this composed setting is transferred to Maude to carry out the execution. Maude is agnostic to whether the multilevel setting that is received is a composed system or not. We have developed a guided procedure that the modeller follows to obtain a set of amalgamated rules. This procedure is defined within an EMF wizard that guides the user through the amalgamation process. The wizard takes four steps to complete, and at the end, the amalgamations of the selected MCMT rules are automatically calculated and produced. For conflicting cases, which are automatically detected, the user has to select what rule has to be prioritised in order to get an amalgamated MCMT rule. The details on how amalgamation is formally carried out, the description of each wizard step, and some illustrative screenshots taken from MultEcore can be found in Paper D.

CHAPTER 4

EXECUTION AND VERIFICATION

In this chapter, we describe techniques for model execution and verification that are based on MTs. Then, we introduce our infrastructure for the execution and verification of multilevel hierarchies using MCMTs and Maude.

4.1 Model transformations

We can find in the literature several approaches for the specification of the behavioural semantics of systems based on MTs. MTs can be used for diverse tasks in MDSE, e. g., for modifying, creating, adapting or merging models. In general, MTs [36, 214] are proposed to systematically manipulate models.

MTs can be classified according to different criteria, as detailed in [63]. For instance, they can be classified based on the target type. The transformation can be model-to-model (M2M), where the target model is incrementally built by finding patterns in the source model and by applying the appropriate actions to the target model. Also, the transformation can be model-to-text (M2T) where elements in the source model are mapped to fragments of text that are produced as outputs.

Another relevant classification is based on analysing the natures of the source and the target metamodels [161]. If the source and target metamodels are the same, the transformation is called *endogenous*. On the other hand, *exogenous transformations* map concepts between different metamodels. Exogenous transformations are often called *translation transformations* and are strongly related to translational semantics (see Section 4.2.1).

In the context of one-to-one MTs, the target model is created by modifying specific parts in the existing source model. In this case, the transformation is called *in-place transformation*. On the other hand, if a new model is freshly created with the corresponding changes, it is an *out-place transformation*.

4.2 Execution semantics

In the last decades, different attempts have been made to support the execution of models. In this thesis we focus on those techniques that engineer the execution of models by specifying their semantics. In this context, execution not only means specifying the execution semantics of the modelling language, but also the development or use of verification techniques that are tailored for them. To support the execution of models, an executable modelling language must provide *execution semantics*. There are several approaches for defining the execution semantics: denotational, translational, operational and axiomatic [47, 212]. These approaches are not necessarily mutually exclusive. Gupta and Pontelli [107] show that a complete language should offer all of the aforementioned kinds of semantics since each of them provides better support for different types of user.

4.2.1 *Denotational and Translational semantics*

The denotational semantics, also known as mathematical semantics, describes the semantics of a language by defining algebraic/mathematical terms [222]. This method maps a program directly to its meaning, called its denotation. The denotation is usually a mathematical object, such as a number or a function. In other words, it expresses the meaning of a DSML through functions that map its constructs to a formal target language where the semantics is well-defined.

In some situations, the specification of denotational semantics is the easiest mechanism in the context of DSMLs, allowing DSMLs to be mapped to more general modelling languages, for which the semantics is defined. With denotational mapping, we can reuse operations defined in the target language. For example, Petri nets (PNs) can both be simulated and analysed, and therefore mapping to PNs for simulation automatically provides analysis functionality to the DSML. Examples of approaches which use denotational semantics are Keywords-based modularisation [55] and MontiCore [133, 134].

In the case where the target language is not formal, the term translational semantics is favoured. The implementation of the translational semantics typically takes the form of a compiler. In this approach, the model is translated into another executable language. This can be done through exogenous MTs or code generation if the target language possesses a grammar. Existing work that uses translational semantics does it mainly to take advantage of the facilities and tools available in the target technical space (e. g., code generators, model-checkers, visualisation tools and simulators). Translational semantics are used, for example, by Cleenewerck et al. [56].

4.2.2 *Operational semantics*

Operational semantics (also called constructive or imperative [161]), provides a formal description of the behaviour of models. It is often defined in terms of atomic, elementary transitions, describing local behaviour [233]. One way to interpret the operational semantics is by implementing endogenous MTs that can be realised using an MT language (e. g., [31]). The interpreter first constructs a representation of a model execution state and then modifies this representation by executing the model through a series of model transitions from one state to the next one (e. g., [42]). These transitions are realised as applications of in-place (or out-place) MT rules.

Another alternative for operational semantics is to use a metaprogramming language to express directly the behavioural semantics as a set of operations for each concept. In contrast to translational approaches, operational approaches directly express the semantics in the same technical space and the concepts are naturally well-known by the expert. Some examples which are based on this alternative are the MOF action language [180] and Kermeta [124, 169]. The MOF action language [180] is an extension of MOF 2.0 that facilitates the specification of the behaviour of the modelled languages. In addition to capturing the behavioural aspect of the language, the MOF action semantics can also enhance automated model manipulation and intra-model transformations. Metadepth [65] and Melanee [12] (as discussed in Section 2.4.2) are some of the few MLM approaches that support model execution.

4.2.3 *Axiomatic semantics*

In the axiomatic semantics approach, the meaning of a program is not explicitly given. Instead, it represents a mechanism for checking whether the programs written in a DSML satisfy certain properties or not. These properties are typically expressed with axioms and inference rules from symbolic logic. Examples of such properties are equivalence between programs or functional correctness (e. g., checking whether the program is correct w.r.t. its specification in terms of pre- and post-conditions). Axiomatic semantics have been used mainly within the programming language [110] field and not that much in the MDSE context. This is mainly because it is not easy to fully specify the behaviour of the model [244] through pre- and post-conditions. Furthermore, axiomatic semantics cannot be made automatically or easily executable.

4.3 Model execution

We based the execution of our models on the definition of a set of endogenous, in-place MTs. The subsequent application of these rules produces *traces* which are sequences of models that have been obtained by such applications. The computed traces can be manually analysed. They may reveal inconsistencies, unexpected behaviours or undesired scenarios, which can then be fixed by the modeller. The absence of such inconsistencies increases the confidence level in the correctness of the model. A step towards the automatic analysis of the model regarding certain behavioural properties is the calculation of the so-called state space. While the full state space represents all possible executions of the model, i. e., all reachable states and all state changes of the model, simulation is used to examine a finite number of executions.

4.4 Model verification

Verification techniques are being used to establish that the design or product under consideration satisfies certain properties. The properties to be validated can be rather elementary, e. g., a system should never be able to reach a situation in which no progress can be made (a deadlock scenario). The system is considered to be "correct" whenever it satisfies all properties obtained from the model's specification. Thus, correctness is always relative to a specification and is not an absolute property of the system.

Formal methods [246] offer a large potential for the integration of verification techniques in the design process, to provide more effective verification techniques, and to reduce the verification time. In summary, formal methods refer to the application of mathematics for the modelling and analysis of systems. They aim to establish system correctness with mathematical rigour. There exist many and diverse formalisms for the specification of properties distributed in different categories:

- **Automata-based formalisms** such as finite-state automata [57] and timed and hybrid automata [231].
- **Logic-based formalisms** such as modal and temporal logic [85] (e. g., LTL [95] and CTL [86]) and rewriting logic [147] (e. g., the Maude system [54]).
- **Process algebra [89]** such as CSS [210] and **process calculus** [182], e. g., π-calculus [183].
- **Visual formalisms** such as Petri nets [170].

From these formalisms and approaches, we use temporal logic (both LTL and CTL) and rewriting logic (Maude), as we show in Papers A, B, C and E.

There exist a plethora of techniques that systematically explore all states of the system model, which provide the basis for a whole range of verification techniques. Some of these techniques are: (i) abstract interpretation [61], (ii) deductive verification (e. g., PVS [215] and STeP [156]), (iii) formal testing [177] and (iv) algorithmic verification, such as model checking. Model checking [28] is a verification technique that explores all possible system states (state space) in an exhaustive manner. In this way, it can be shown that a given system model satisfies a certain property. The research community is continuously working on techniques to alleviate the effect of the state explosion problem [237]. In Paper B [198], we use the sweep-line method [117] to reduce the memory usage during state space exploration. We use model checking as verification technique in MultEcore (see Paper E, where we also apply abstraction to reduce the size of the state space).

There exist some tools that handle the execution and verification of models based on MTs. Some examples are Henshin [9, 223], the GRaph-based Object-Oriented VErification (GROOVE) [126, 190], the GEMOC Studio [43, 59] and e-Motions [232].

4.5 Execution and Verification in MultEcore

To cope with the execution of models within the MLM context using MultEcore, the MCMTs were first formally introduced in [155] and has been extended in Papers C, E and F for the execution of MLM hierarchies. MCMTs are reusable, multilevel, in-place, endogenous MTs, which have several applications in addition to the specification of behaviour, for instance, to specify multilevel constraints (see Paper F for an example). The semantics of MultEcore is given by a transformation of MLM hierarchies and MCMT rules into rewriting logic in the form of Maude specifications [54, 79].

Maude [53] is a specification language based on rewriting logic [164, 165], a logic of change that can naturally deal with states and non-deterministic concurrent computations. A rewrite logic theory is a tuple $(\Sigma;E;R)$, where $(\Sigma;E)$ is an equational theory that specifies the system states as elements of the initial algebra $T_{(\Sigma;E)}$, and R is a set of rewrite rules that describe the one-step possible concurrent transitions in the system. Σ is a signature that specifies the type structure (e. g., sorts and subsorts) and operations, and E is a collection of equations and memberships. Rewrite specifications thus described are executable if they satisfy restrictions such as termination and confluence of the equational subspecification, and coherence of equations and rules. Maude provides support for rewriting modulo associativity, commutativity and identity, which captures the evolution of models made up of objects linked by references.

The syntactical facilities of Maude have allowed us to use a representation of MLM hierarchies and MCMT rules very close to that of MultEcore. Indeed, this minimal representation distance has eased the automation of the bidirectional transformation between them. These transformations give MultEcore users access to the Maude execution engine, which is the most efficient engine for rewriting modulo (combinations of) associativity, commutativity, and identity [80, 97]. In addition, it also gives access to Maude's formal tool environment, which includes, e. g., tools for reachability analysis, model checking, and confluence and termination analysis.

Maude is used as a backend tool, hidden to the user so that the interaction is entirely done through MultEcore, making the modeller unaware of Maude details. The overall MultEcore-Maude infrastructure is sketched in Figure 4.1. The left-hand side shows the **MultEcore** part, where we specify multilevel DSMLs by providing a **Multilevel Hierarchy** and a set of **MCMT rules**. The **Transformer MultEcore** ⟷ **Maude** has been developed as a transformation that takes MultEcore specifications (the MLM hierarchies and the associated MCMTs) and automatically generates the corresponding Maude specifications. The outputs produced as XML files by Maude, as the result of performing execution, analysis and verification operations, are then automatically translated back into MultEcore models that are directly displayed graphically. As a

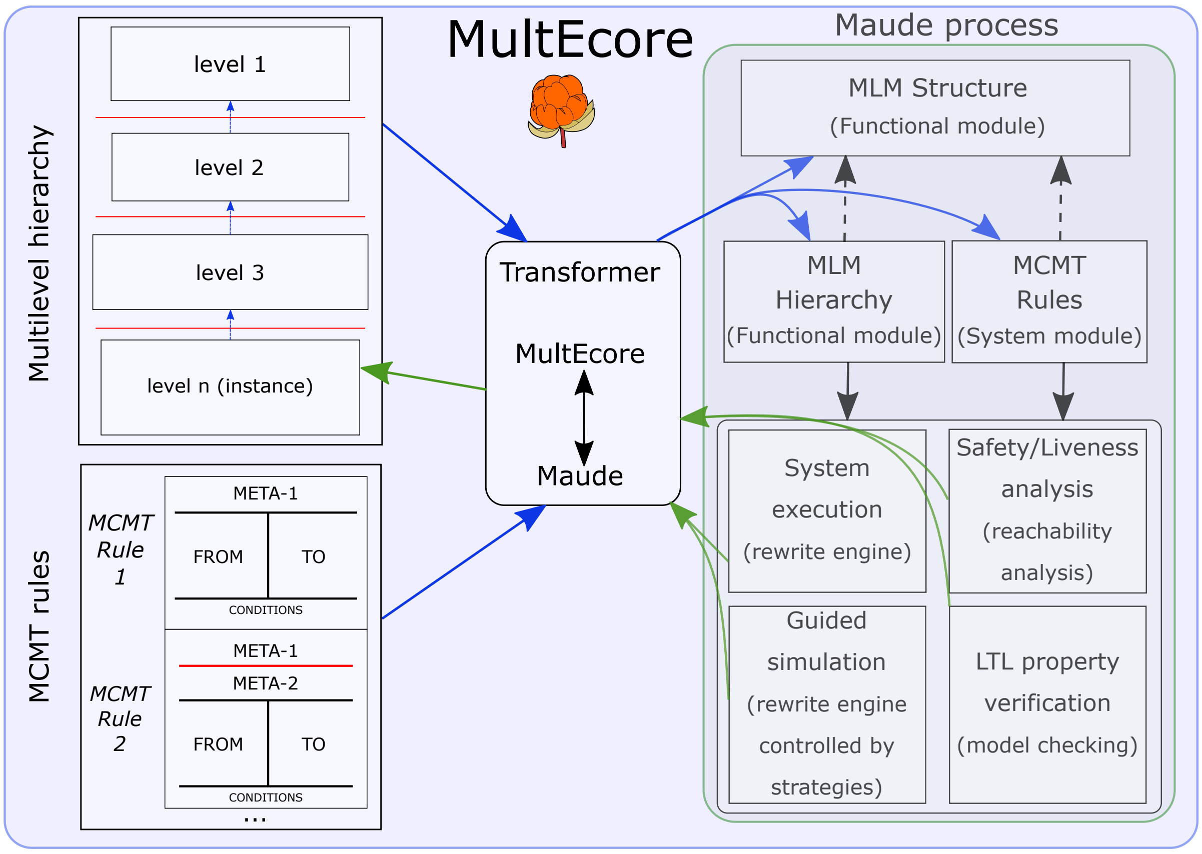

MultEcore
Multilevel hierarchy
level 1
level 2
level 3
level n (instance)
MCMT rules
MCMT Rule 1
META-1
FROM
TO
CONDITIONS
MCMT Rule 2
META-1
META-2
FROM
TO
CONDITIONS
...
Transformer
MultEcore
Maude
Maude process
MLM Structure (Functional module)
MLM Hierarchy (Functional module)
MCMT Rules (System module)
System execution (rewrite engine)
Safety/Liveness analysis (reachability analysis)
Guided simulation (rewrite engine controlled by strategies)
LTL property verification (model checking)

Fig. 4.1: Infrastructure for the execution and analysis of MLM hierarchies

consequence of the small representation distance between the two, the transformation is straightforward. This has in fact simplified not only the transformation from MultEcore into Maude, but also the transformation in the opposite direction to bring results of simulations and analysis back into the MultEcore tool.

To provide an intuition of how the transformation works, each MultEcore object (including both a hierarchy and its MCMTs) is mapped into a corresponding Maude object. References and conditions are handled in exactly the same way, by using references as names and using the same set of expressions (types and operators) for conditions. The rewriting modulo associativity, commutativity and identity available in Maude captures naturally the operational semantics of MCMTs. The major contributions have been handling boxes inside the MCMTs and performing the rewriting of multilevel hierarchies. Examples of how boxes are used can be found in Chapter 5 and Paper E.

The possibility to specify boxes allows us to define patterns where its unfolding would result in a collection of elements. In this way, a single rule can cover an entire set of rule variants where the number of elements may vary depending on the instance model. We have developed a two-steps process to handle boxes. For each MCMT, a rule without boxes is generated. When Maude finds a match for such a rule, it gets enough information to process the cardinalities of the most external boxes. Then, using the metaprogramming capabilities of Maude, a second rule with the corresponding number of replicas of the boxes is generated, which is used to take the corresponding rewriting step using the original partial substitution. Nested boxes are processed one level at a time, recursively unfolding the boxes, and expanding the matching until no further boxes are left. Lets and conditions inside boxes are processed at each step (see Chapter 5).

As mentioned at the end of Chapter 2, we have provided basic support for parameterisable languages to be used, for instance, in the manipulation or specification of attributes, the cardinality of boxes and the specification conditions on the MCMTs. The language to be used to specify attributes is a parameter of the model. Its instantiation provides a set of available types, a syntax, and an *eval* operation that gives semantics to the language. Optional operations, like *match* and *apply*, provide additional functionality, which may be useful for some instantiations. So far, we have developed definitions and corresponding instantiating views for OCL [51] and SML [234]. The support for OCL is based on the Maude semantics of OCL proposed in [205]. It is interesting to point out that if SML is used for attributes, we still have definitions of OCL for boxes' cardinalities and other MultEcore features. For this, we exploit Maude's functionalities for the replication of built-in types.

The right-hand side of Figure 4.1 shows the **Maude process** perspective. The transformer produces a functional module with the equational theory used to represent MLM hierarchies (**MLM Structure** and **MLM Hierarchy**), and a system module with rewrite theory that represents the **MCMT Rules**. More details on how a system is specified in Maude can be found in [54].

CHAPTER 5

THE COLOURED PETRI NETS CASE STUDY

There exist many ways in which MLM can help to alleviate the restrictions of traditional MDSE approaches. One of them is the ability to give extendibility support to existing modelling languages that are not easy to extend with, for instance, domain-specific features, or provide additional concepts that one might need in the course of modelling certain systems or environments. An example of this is the case of Coloured Petri nets (CPNs). In this section, we illustrate how we have modelled a multilevel hierarchy for PNs where a specific branch defines the CPN language. We also specify MCMT rules which describe the operational semantics that enables the execution of CPN models and illustrate how we have incorporated a preliminary Standard ML (SML) implementation within the MultEcore-Maude infrastructure. This opens the door to further extend the multilevel hierarchy with, e. g., domain-specific CPNs.

5.1 Coloured Petri nets

CPNs [116, 118] is a graphical modelling language in the domain of distributed systems that facilitates the specification of communication protocols [77], data networks [38], distributed algorithms [188], embedded systems [2], business process and workflow modelling [238], manufacturing systems [250] and agent systems [88]. CPNs belong to the family of high-level PNs [119], which are characterised by the combination of classical PNs [170, 189] and a programming language. The use of a programming language, e. g., SML [234] in CPNs, provides the primitives for the definition of data types, for describing data manipulation and for creating compact and parameterisable models. CPNs are widely used due to its rich body of theoretical results enabling analysis, and an enormous set of supporting tools. The modelling language is suited for discrete-event processes that include choice, iteration, and concurrent execution. In a nutshell, a CPN model of a system is an executable model representing the states of the system (places) and the events (transitions) connected to them (via arcs) that can cause the system to change its state.

The increasing complexity of systems has promoted the proliferation of a plethora of PNs variants and extensions during the last decades, as classical PNs are too basic to capture the needs of most of the nowadays systems. A high-level comparison of different kinds of PNs, divided into three categories, can be found in [32]. We refer the reader to [230] for a survey where several PN tools are explored. While the CPN language contains few (still powerful) constructs that the modeller needs to master in order to understand and use, the main tool for CPNs, CPN Tools [62], is not designed to be easily extended with domain-specific features, although DSMLs have proved to be one of the most important mechanisms of MDSE [168]. Furthermore, several recent applications of CPNs [217] have shown that it would be beneficial to be able to develop domain-specific variants that would make it possible to support:

- **Modelling patterns** representing commonly used approaches to capture concepts from the problem domain.
- **Modelling restrictions** forcing the modeller to use only certain constructs in the language when modelling concepts from the problem domain.
- **Subtyping of elements** allowing specific interpretations of certain model elements such as places, transitions, and arcs [204].

5.2 The CPN modelling language

To understand the CPN language and the CPN Tools, we developed a CPN model of the MQTT publish-subscribe protocol [30]. We decided to construct such a model because it was a sufficiently realistic case study to use the available functionalities of the CPN Tools and, furthermore, the MQTT specification was written in natural language and presented some ambiguities. Hence, the developed CPN model, described in Paper A, serves as formal specification of it. In this chapter, we describe what mechanisms we have developed in order to achieve some of the CPNs capabilities. To demonstrate this, we will use an excerpt of the CPN model that describes the basic interactions regarding subscription and unsubscription of clients to topics in MQTT. The CPN model fragment is shown in Figure 5.1.

A CPN model describes the states using *places* (drawn as ellipses) of the system and the events using *transitions* (drawn as rectangles) that can cause the system to change its state. The CPN model in Figure 5.1 contains three places, two transitions, a number of directed arcs connecting places and transitions, and some textual SML inscriptions next to the places, transitions, and arcs. Nodes (places and transitions) together with the directed arcs constitute the net structure. An arc always connects a place to a transition or a transition to a place. Each place has an associated *type* (also called *colour set*) determining the kind of data that tokens residing on the place may carry.

A place can hold an arbitrary number of *tokens* that constitute the *marking* of the place. The state of a CPN is a marking of the places of the CPN model. The actions of a CPN consist of occurrences of enabled transitions. For a transition to be enabled, it must be possible to find a binding of the variables that appear in the surrounding arc expressions of the transition such that the arc expression of each *input arc* evaluates to a multi-set of token colours that is present on the corresponding input place. The types of the arc expressions need to conform to the types of the places they are connected to. When a transition occurs with a given binding, it removes from each input place the multi-set of token colours to which the corresponding input arc expression evaluates. Analogously, it adds to each *output place* of the transition the multi-set of token colours to which the expression on the corresponding output arc evaluates. In addition to the arc expressions, it is possible to attach a boolean expression to each transition. This

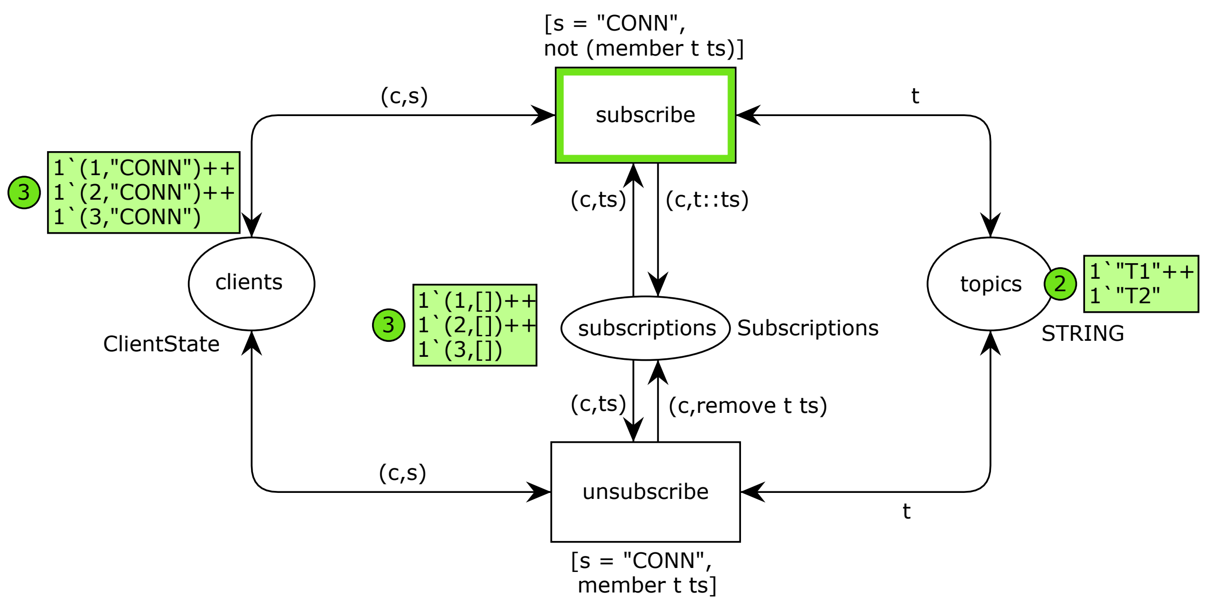


Fig. 5.1: CPN model excerpt for a simple communication flow

boolean expression is used as a *guard* on transitions (e. g., **[s = "CONN", member t ts]** at the bottom of Figure 5.1). It specifies an additional requirement for the transition to be enabled.

As shown in Figure 5.2, the **ClientState** colour set is composed by the product of two elements, of types **INT** and **STRING**, as a tuple, that represents the ID and the state of the client. The **clients** place has as colour set **ClientState**. Initially, we have three tokens in **clients** place, **(1, "CONN"), (2, "CONN") and (3, "CONN")**, where each one represents one client. They are displayed on the left of Figure 5.1 and, as one can observe, all the clients are in state **CONN**, which means they are connected. The place **Subscriptions** keeps track of the topics each client is subscribed to. The **Subscriptions** colour set is given by the product of **INT** and **Topics**, which is a list of **STRING** as shown in Figure 5.2. In the initial state of the net shown in Figure 5.1 no client is subscribed to any topic. The available topics are listed in the **topics** place (in the example, the two available topics are **T1** and **T2**).

The four variables (**c** of type **INT**, **s** and **t** of type **STRING** and **ts** of type **Topics**) declared in Figure 5.2 are used in the arc expressions and in the guards of the transitions. In the state depicted in Figure 5.1, the **subscribe** transition is enabled. A transition is enabled if the required tokens are present on places connected to input arcs of the

```
colset ClientState   = product INT * STRING;
colset Topics        = list STRING;
colset Subscriptions = product INT * Topics;

var c    : INT;
var s, t : STRING;
var ts   : Topics;
```

Fig. 5.2: Colour set definition and variables used in the CPN model in Figure 5.1

transition. As an example, the transition **subscribe** has three input arcs and three output arcs. Hence, an occurrence of this transition will remove tokens from the places **clients**, **topics** and **subscriptions**, and will add tokens to the same places (the three of them are both input and output places). The specific tokens added and removed by the occurrence of a transition are determined by the arc expressions, which are positioned next to the arcs. An occurrence of an enabled transition requires data values to be bound to the free CPN variables appearing in the guard, and in the input and output arc expressions of the transition. This is needed to evaluate the arc expressions and the guard. Note that the **subscribe** transition has two double arcs connected to it, from **clients** and **topics** places with **(c,s)** and **t** arc expressions, respectively. Double arcs are just syntactic sugar to represent two arcs in opposite directions with identical arc expressions. The **subscribe** transition has also one input arc from the **Subscriptions** place with **(c,ts)** as arc expression and one output arc towards the **Subscriptions** place with **(c,t::ts)** as its arc expression. The latter represents a basic list operation that appends the matched topic bound to the variable **t** to the existing list bound to the **ts** variable. In other words, it adds the topic to the client's topic list.

Notice also the guard expression of the **subscribe** transition, **[s = "CONN", not (member t ts)]**, which is composed of two boolean conditions. The first one, **s = "CONN"**, verifies that the candidate client must be connected. The second one, **not (member t ts)**, uses an auxiliary function (**member**) to check whether the topic is not already on the client's list. The unsubscription proceeds similarly, where the guard at the **unsubscribe** transition verifies that the client is connected and the topics list contains the candidate topic. The two arc expressions between **unsubscribe** and **Subscriptions** facilitates the deletion of a topic from a client's topics list once the transition is fired. It uses the **remove t ts** auxiliary function in the output arc of the transition to remove the topic from the list of subscribed topics.

In CPN Tools, double arcs (explained above) are the mechanism to recreate *read arcs* [240] where the tokens that are present on them are used only for readability purposes and they should never be removed or modified. The workaround with the double arc basically removes and adds the token which simulates the read arc effect. We describe in Section 5.3 how the extendibility capabilities of MultEcore allows to define new elements such as *Read Place* and *Read Arc*.

5.3 Coloured Petri nets in MultEcore

In Paper E we defined a multilevel hierarchy to capture regular PNs and PNs with reset and inhibitor arcs. A reset arc is an input arc that connects a place to a transition and that removes all the tokens of the place when the transition is fired. An inhibitor arc is an input arc which is used to reverse the logic of an input place. With an inhibitor arc, the absence of a token in the input place is what enables the connected transition (not its presence). Furthermore, we specified MCMT rules to execute instance models of these two PNs variants and demonstrated how we can execute, verify and analyse these instance models via the MultEcore-Maude infrastructure.

To handle CPNs, we have taken advantage of the PNs multilevel hierarchy shown in Paper E. We depict in Figure 5.3 an overview of the entire PNs DSML family considered. The first two levels are common, where the first one (*petri-net-concepts*), located at level

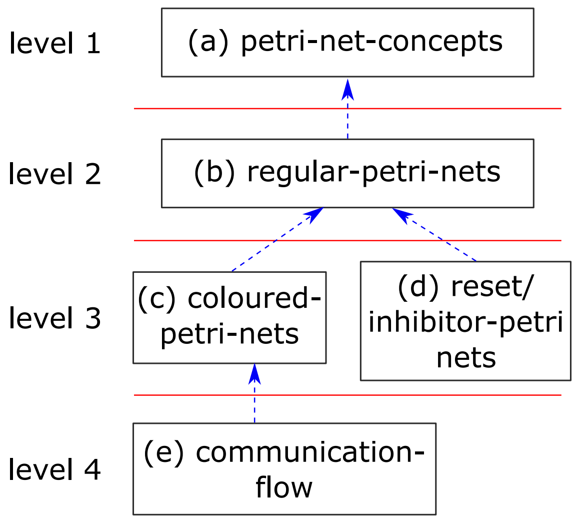


Fig. 5.3: PNs family multilevel hierarchy overview

1, represents the abstract concepts of PNs, and the model below (*regular-petri-nets*), at level 2, represents regular PNs. It is in level 3 where we branch the hierarchy, since the version shown in Paper E captures PNs with reset and inhibitor arcs (right-hand branch, composed by Figure 5.3 (a), (b) and (d)). For the CPNs case study, level 3 captures CPNs (*coloured-petri-nets* model) as shown in Figure 5.3 (c). At level 4, we define a CPN instance model, which will be detailed in the next sections and represents the model depicted in Figure 5.1 in MultEcore syntax. Note that any of the models in the intermediate levels can be considered a DSML which is used to define the level(s) below it, using the types they define in a structurally coherent manner, and satisfying the given constraints. In other words, regular PN models could also be created by instantiating directly elements in level 2. As one can observe, the definition of a language family, such as the PNs family allows to capture the common patterns once (levels 1 and 2) and use them as core language of the different potential branches that might emerge representing domain concretisations (level 3). In this regard, a multilevel hierarchy facilitates the *Modelling pattern* item discussed in Section 5.1. In the next sections, we explore the left-hand branch of the multilevel hierarchy and detail the content of each model.

5.3.1 *Petri nets concepts*

We show in Figure 5.4 (a) a PN metamodel aimed to capture the abstract concepts of Petri nets (*petri-nets-concepts*). The purpose of this model is merely structural, i. e., subsequent levels below should define the concrete semantics of the PN language(s). A **PetriNet** contains **Node**s, which can be either a **Place** or a **Transition**, and **Arc**s. The tool MultEcore allows us to make use of the *inheritance* relation and to mark **Node** as an abstract class, which cannot be instantiated (note the italics). Note that we only display the multiplicities on edges in those cases where it is not the default one (**0..***). For instance, **target** and **source** multiplicities are **1..1**.

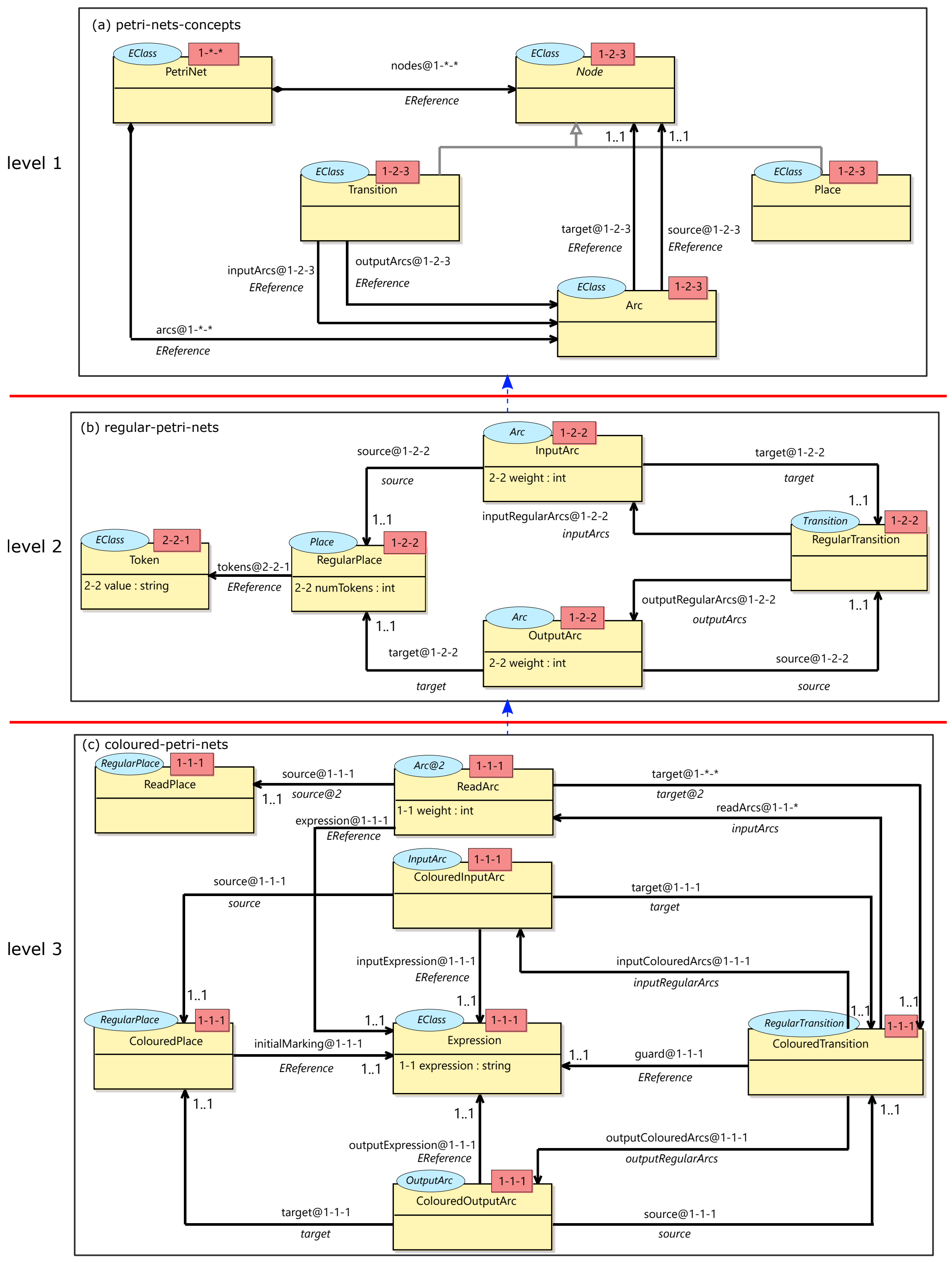

level 1
(a) petri-nets-concepts
EClass 1-*-* PetriNet
nodes@1-*-* EReference
EClass 1-2-3 Node
EClass 1-2-3 Transition
EClass 1-2-3 Place
target@1-2-3 EReference
source@1-2-3 EReference
inputArcs@1-2-3 EReference
outputArcs@1-2-3 EReference
EClass 1-2-3 Arc
arcs@1-*-* EReference
level 2
(b) regular-petri-nets
Arc 1-2-2 InputArc 2-2 weight : int
source@1-2-2 source
target@1-2-2 target
inputRegularArcs@1-2-2 inputArcs
EClass 2-2-1 Token 2-2 value : string
tokens@2-2-1 EReference
Place 1-2-2 RegularPlace 2-2 numTokens : int
Transition 1-2-2 RegularTransition
outputRegularArcs@1-2-2 outputArcs
Arc 1-2-2 OutputArc 2-2 weight : int
level 3
(c) coloured-petri-nets
RegularPlace 1-1-1 ReadPlace
Arc@2 1-1-1 ReadArc 1-1 weight : int
source@1-1-1 source@2
target@1-*-* target@2
readArcs@1-1-* inputArcs
expression@1-1-1 EReference
InputArc 1-1-1 ColouredInputArc
source@1-1-1 source
target@1-1-1 target
inputExpression@1-1-1 EReference
inputColouredArcs@1-1-1 inputRegularArcs
RegularPlace 1-1-1 ColouredPlace
EClass 1-1-1 Expression 1-1 expression : string
initialMarking@1-1-1 EReference
guard@1-1-1 EReference
RegularTransition 1-1-1 ColouredTransition
outputExpression@1-1-1 EReference
outputColouredArcs@1-1-1 outputRegularArcs
OutputArc 1-1-1 ColouredOutputArc
target@1-1-1 target
source@1-1-1 source
1..1

Fig. 5.4: Models (a), (b) and (c) from Figure 5.3

5.3.2 *Regular Petri nets*

The definition for regular PNs that we provide in this case study is not restricted to the so-called *Ordinary PNs* where input and output arcs consume or produce, respectively, only a single token [108]. We allow natural numbers on arcs so that more than one token can be added/removed at a time. This is also aligned with the CPNs language specification which allows using so-called *multi-sets* of tokens [118]. A multi-set is similar to a set, except that values can appear more than once. Following the PNs convention, we denote this number as the *weight* of the arc.

Figure 5.4 (b) displays the *regular-petri-nets* model, located at level 2 of the hierarchy, where we instantiate the concepts defined at level 1. Thus, in this model, we provide the structural basis for the modeller to be able to define further PN instance models. **InputArc** and **OutputArc** connect **Regularplace**s and **RegularTransition**s. A **RegularPlace** controls how many tokens it is holding via the **numTokens** attribute.[1] The weight of arcs is represented as attributes called **weight** of type **int** in classes **InputArc** and **OutputArc**.

5.3.3 *Coloured Petri nets*

As stated in Section 5.1, CPNs and, in general high-level PNs, are boosted with the capabilities that a programming language offers. The functional SML language is used in CPNs, and to cope with it, we have instrumented our MultEcore-Maude infrastructure to provide a preliminary parametric SML implementation to demonstrate its potential. The *coloured-petri-nets* model in Figure 5.4 (c) contains some relevant CPN concepts discussed in Section 5.1.

A significant difference of CPNs with respect to other PNs is that one can specify SML expressions and functions in different parts of the models. The **Expression** node, associated to arcs, places and transitions in Figure 5.4 (c), precisely captures this, where the concrete expression is provided by instantiating its **expression** attribute. Such an attribute is of type **string**. Instances of the **expression** attribute will be written as strings in MultEcore and will be appropriately processed in Maude. The rest of the elements in this level make use of expressions in different ways: **ColouredPlace**s can have **initialMarking**s (edge from **ColouredPlace** to **Expression**), **ColouredTransition**s might have **guard**s as boolean conditions, and **ColouredInputArc**s and **ColouredOutputArc**s can have **inputExpression**s and **outputExpression**s, respectively. All these concepts (except the initial marking, which we do not instantiate in level 4 of the hierarchy) have been shown in Section 5.1.

Another fundamental aspect regarding flexibility is the possibility to easily extend the modelling language represented by the multilevel hierarchy. To demonstrate this, we have extended the *coloured-petri-nets* model by adding the **ReadPlace** and **ReadArc** nodes (the semantics of these elements is given in Section 5.2). A **ReadArc** connects a **ReadPlace** and a **ColouredTransition**, which at the same time keeps track of the number of **ReadArc**s connected to it (via **readArcs** edge). **ReadArc**s can also have an **expression** (edge from **ReadArc** to **Expression**). By simply adding the **ReadPlace**

[1]Note that the number of tokens may be calculated with the OCL expression **rp.tokens→size()**. The attribute is however used to speed up calculations and to illustrate the manipulation of attributes in MCMT rules.

and **ReadArc** nodes, as well as the edges that connect them, allows us to extend the language and demonstrate the application of *Modelling restrictions* and *Subtyping of elements* listed in Section 5.1. The former is achieved by structurally representing that each **ReadPlace** must be connected to a **ColouredTransition** via a **ReadArc**, forbidding, for instance, that a **ReadPlace** is connected to a **ColouredInputArc**. The latter is given by freshly creating a new interpretation of a place and an arc that has been decided a-posteriori. Note that subtyping does not strictly refer to the use of a specialisation mechanism, e. g., inheritance, but the possibility of specifying new concretisation of certain elements in levels below.

5.4 Behaviour of Coloured Petri nets

As for other PNs, the dynamic behaviour of a CPN is given by the *token game*, representing various states of the system. This token game is based on the firing of transitions that leads to the consumption/production of tokens; each fired transition produces a new model state. We have already specified the semantics of other PNs (e. g., regular PNs and reset/inhibitor PNs in Paper E) in a similar way. We have defined a single MCMT rule that can cover every possible execution of a single transition in CPNs. However, the fact that we have to take into account that CPNs incorporate a programming language, SML in our case, has increased the complexity of the designed MCMT rule. Furthermore, it has encouraged us to enrich MCMTs with additional functionalities that will be explained in this section. The rule is shown in Figure 5.5. We do not enter into details of the **META** block as it has already been explained in Chapter 2. Hence, we refer the reader to Paper E for details on how the **META** block works. In the following, we focus on the **FROM** and **TO** blocks.

The use of the boxing mechanism has facilitated the specification of a single rule for every possible permutation and combination of different arcs and places, as well as any number of tokens connected to them. The *Fire transition* MCMT rule has been designed to cover different cases that can be simultaneously present regarding places connected to a transition in the CPNs context. Each case is encapsulated within a dashed box, which we explain in the following, from top to bottom of the **FROM** part of Figure 5.5:

1. The first box covers input places connected via input arcs to the transition **tr**. It is surrounded by a dashed box where its cardinality is given by the following OCL expression: **tr.inputColouredArcs→select(a | tr.outputColouredArcs→exists(b | a.source != b.target))→size()**. This expression is needed to count places that are only connected via input arcs to the transition and to distinguish them from those places that are connected via both input and output arcs (this is the case number 4, depicted at the bottom of Figure 5.5). The elements placed inside this first box, **p1**, **a1** and **a1e**, together with their corresponding edges, **a1s**, **a1t** and **a1exp**, specify the place-arc-expression pattern that can be found an arbitrary number of times. The inner box, that encapsulates **tk1** and **p1tk** is used to match any number of tokens given by the **weight a1w** of the arc **a1**, which is used as cardinality of this box.

   To deal with the fact that the free variables of a transition are evaluated taking into account all the arcs connected to them, we have incorporated *let* clauses to

MCMTs. Let clauses allow to store each partial match of the variables of the arc expressions (e. g., **a1exp** that captures the matched arc expression of an element in the model) with the values of the tokens (e. g., **v1**). These partial matches that are calculated when the rule is unfolded must be tracked back and put together in the general context of the rule. For instance, the **let subst1i = match(a1exp, v1)** creates one variable **subst1i** per matched token. Then, the outer expression, **let subst1 = U subst1i**, creates one variable **subst1** per instance of **ColouredPlace**, which gets assigned the union (**U**) of the substitution variables from all the matched tokens. We explain below how all the matches are put together and used to create new tokens based on the evaluation of the arc expressions.

2. The second box handles read arcs connected to the transition. The number of read places and arcs connected to the transition is calculated with the OCL expression **tr.readArcs→size()**. The two let clauses defined within this box work analogously as explained for the first case. The cardinality of the inner box is given by the **weight a2w** of the arc **a2**.

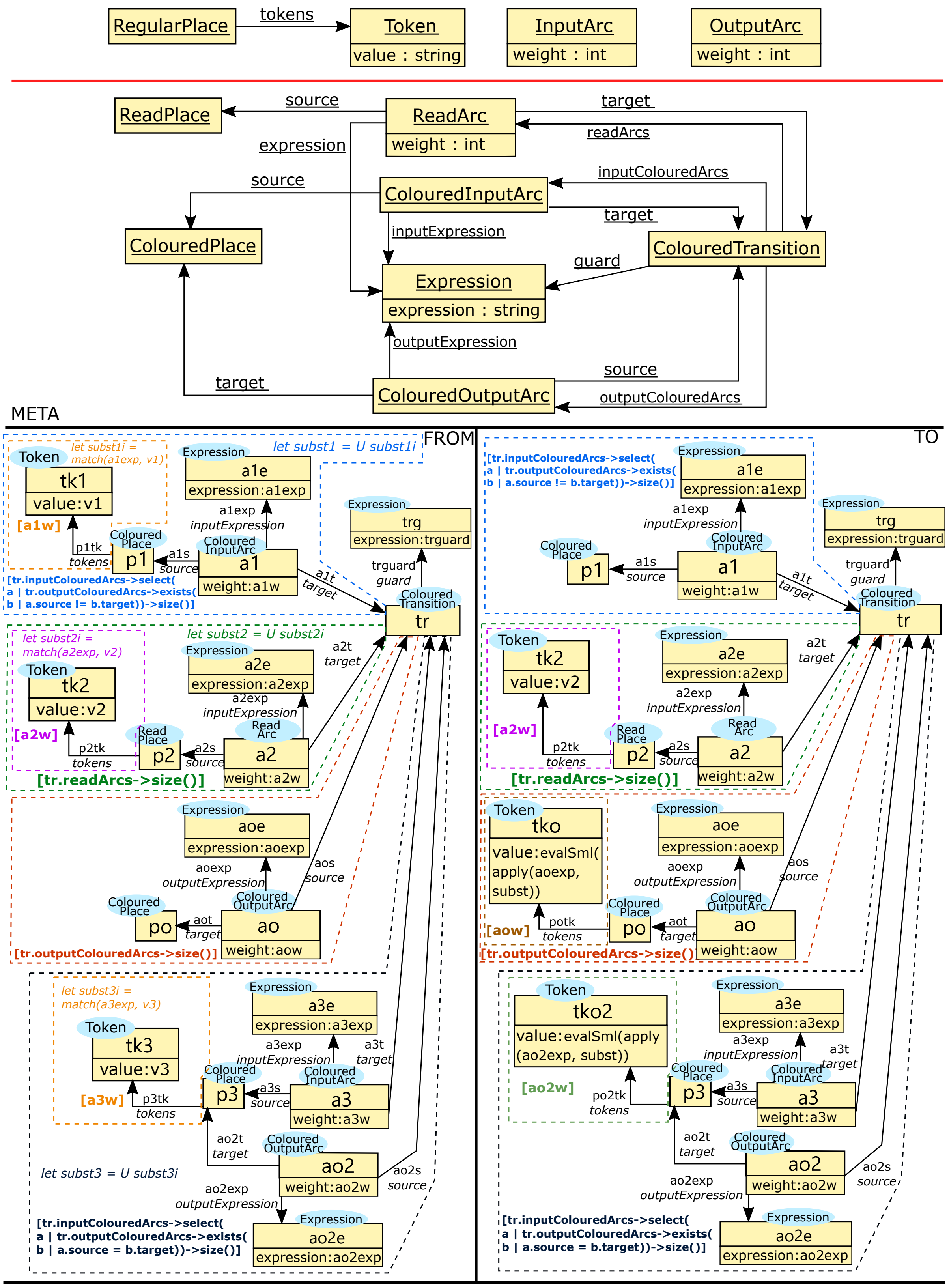
3. The third box covers all the output places connected to **tr**. The number of these kind of places (instances of **ColouredPlace**s, **po**, connected to instances of **Coloured-OutputArc**s, **ao**) is given by the expression **tr.outputColouredArcs→size()**.

4. The last box deals with places that are simultaneously input and output of the transition, i. e., it has an input arc and an output arc connected to each of them. The cardinality of the box is given by the OCL expression complementary to the one shown in the first case: **tr.inputColouredArcs→select(a | tr.outputColouredArcs→exists(b | a.source = b.target))→size()**. Note that the **let** elements follow the same logic as in cases 1 and 2. The difference is that we have **a3** to capture the input arc, and **ao2** to capture the output arc, with their corresponding expressions, **a3e** and **ao2e**, respectively.

In order to execute this rule, there are also some additional requirements that must be satisfied. At the bottom of Figure 5.5, we have the conditions of the rule. The first line, **let subst = subst1 U subst2 U subst3**, defines a variable **subst** that takes the union of all the matches captured in cases 1, 2 and 4. This variable contains all the information of the matches of the unfolded variables with the corresponding unfolded values which corresponds to the *binding* of the transition (see [136] for details on bindings in CPNs). The second line in the **Conditions** block checks whether the guard **trguard** (attribute located in **trg** node) is satisfied. To achieve this, we use the expression **evalSml(apply(trguard, subst))**, which checks whether there exist some combination of variables with values assigned that fulfil the boolean expression of the transition that might use some of these variables.

The right-hand side of the rule is shown in the **TO** block of Figure 5.5. We again describe each of the cases, from top to bottom:

1. The tokens that have been matched from the corresponding input places have been removed, hence they do not appear in the **TO** side.



Conditions:
let subst = subst1 U subst2 U subst3
evalSml(apply(trguard, subst))

Fig. 5.5: *Fire transition* MCMT rule for CPNs

2. In read places, the tokens are only used to read their information, but are left untouched. That is why the execution does not perform any change on read places.

3. Output places are going to get new tokens whose number is given by the weight **aow** of the arc **ao**. The values of the tokens are given by computing the arc expression **aoexp** with the **subst** variable which contains the information of the tokens consumed previously. This value is therefore calculated using the function **evalSml(apply(aoexp,subst))**.

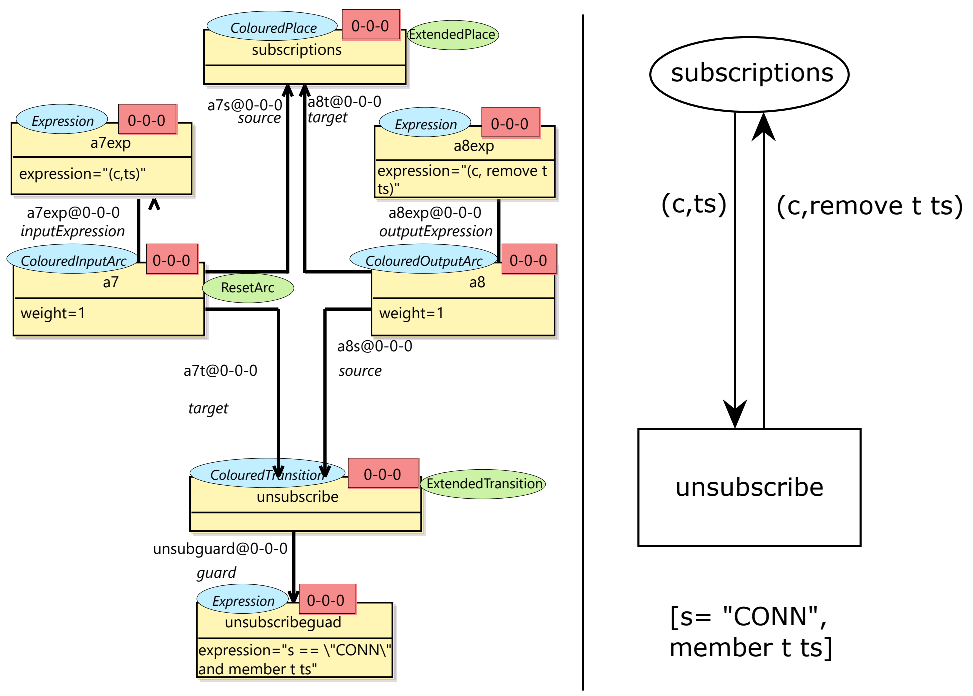
4. Analogously to case number 3, the new tokens created in the place that acts as input and output at the same time are calculated by the expression **evalSml(apply(ao2exp,subst))**. Also, the tokens that were matched in the **FROM** part are removed from the place (**p3**).

The potential of the boxes has allowed us to use a single rule to cover every possible execution of a transition in a CPN model. The incorporation of the *let* clauses and operations, such as the union, has made it possible to preserve and save information across boxes and compose it for producing the new tokens or evaluate the guard of the transition. We refer the reader to Paper E for additional information on the available facilities for execution and verification of PNs.

5.5 Composition of Petri net languages

We have illustrated the PNs family in the form of a multilevel hierarchy in Figure 5.3. The two models located at level 3, namely *coloured-petri-nets* and *reset/inhibitor-petri-nets*, symbolise different sublanguages the instances of which can represent concrete configurations for CPNs or for PNs with reset and inhibitor arcs (e. g., the *gas station* model shown in Paper E). However, these two languages are not mutually exclusive and they could be used together, as some concepts of one of them might be useful for the other. As we have discussed in Chapter 3, these kinds of languages can be composed by using the notion of supplementary hierarchies. For instance, we will show in this section how CPNs can incorporate some concepts from the reset/inhibitor PNs to enhance some elements at the instance level. It is important to mention that although CPN Tools already supports read and reset arcs, for comparison reasons, we chose this example to illustrate the feasibility of extending a language through supplementary hierarchies. Similar to this example, we can add other functionalities that are not currently supported in CPNs.

Figure 5.6 (a) shows an excerpt of the *communication-flow* model in MultEcore syntax. It comprises the unsubscription situation which analogous part in CPN syntax (extracted from Figure 5.1) is depicted in Figure 5.6 (b). We can find in Figure 5.6 (a), for instance, the guard of the **unsubscribe** transition (as an instance of the **expression** attribute of the **unsubscribeguard** node) **"s==\"CONN\" and member t ts"**. We can also find the two arc expressions attached to the arcs that connect the **subscriptions** place (of type **ColouredPlace**) with the **unsubscribe** transition of type **ColouredTransition**). One can observe that these two arc expressions, namely **a7exp** and **a8exp** are the same as those reflected in the corresponding arcs in Figure 5.6 (b).



(a) communication-flow model excerpt in MultEcore (b) communication-flow model excerpt in CPN Tools

Fig. 5.6: *Communication-flow* model in MultEcore syntax (a) (composed version) and in CPN Tools (b)

The model shown in Figure 5.6 (a) represents an excerpt of the full model in MultEcore syntax which is illustrated in Figure 5.7. This model is now an instance of both the *coloured-petri-nets* model located in Figure 5.4 (c) and the *reset/inhibitor-petri-nets* model located in Figure 5.4 (d). However, the key point of Figure 5.6 (a) is that some elements are multi-typed by elements defined in the other branch of the hierarchy (specifically in the *reset/inhibitor-petri-nets* model). Note that the complete model in Figure 5.7 do not represent a composed model, but we have opted to use the model excerpt in Figure 5.6 (a) to illustrate a composition example and its analogous part in Figure 5.6 (b) (without any composition detail) to facilitate its readability. Still, the complete model shown in Figure 5.7 could be also multi-typed as we have exemplified in Figure 5.6 (a).

The current semantics of the **unsubscribe** transition is that, every time it is fired, a client unsubscribes from one of the topics it was subscribed to. Let us consider now that we want to change the semantics where **unsubscribe** would remove all the tokens at once, i. e., all clients unsubscribe from all the topics they were subscribed to. One of the concepts defined at level 3 of the right-hand branch is the notion of reset arc which is an input arc that connects a place to a transition and that removes all the tokens of the place when the transition is fired. Then, we can make use of the **ResetArc**, **ExtendedPlace** and **ExtendedTransition** concepts defined in that model (these elements can be found in Figure 6 of Paper E) and use them to multi-type elements in our model at level 4. In this case, the **subscriptions** place in Figure 5.6 is typed by **ColouredPlace**, from the *coloured-petri-nets* model, and is typed by **ExtendedPlace**, from the supplementary branch (green ellipse at the top right corner of the node). Analogously, **a7** is typed by **ColouredInputArc** and **ResetArc** and **unsubscribe** is typed by **ColouredTransition**

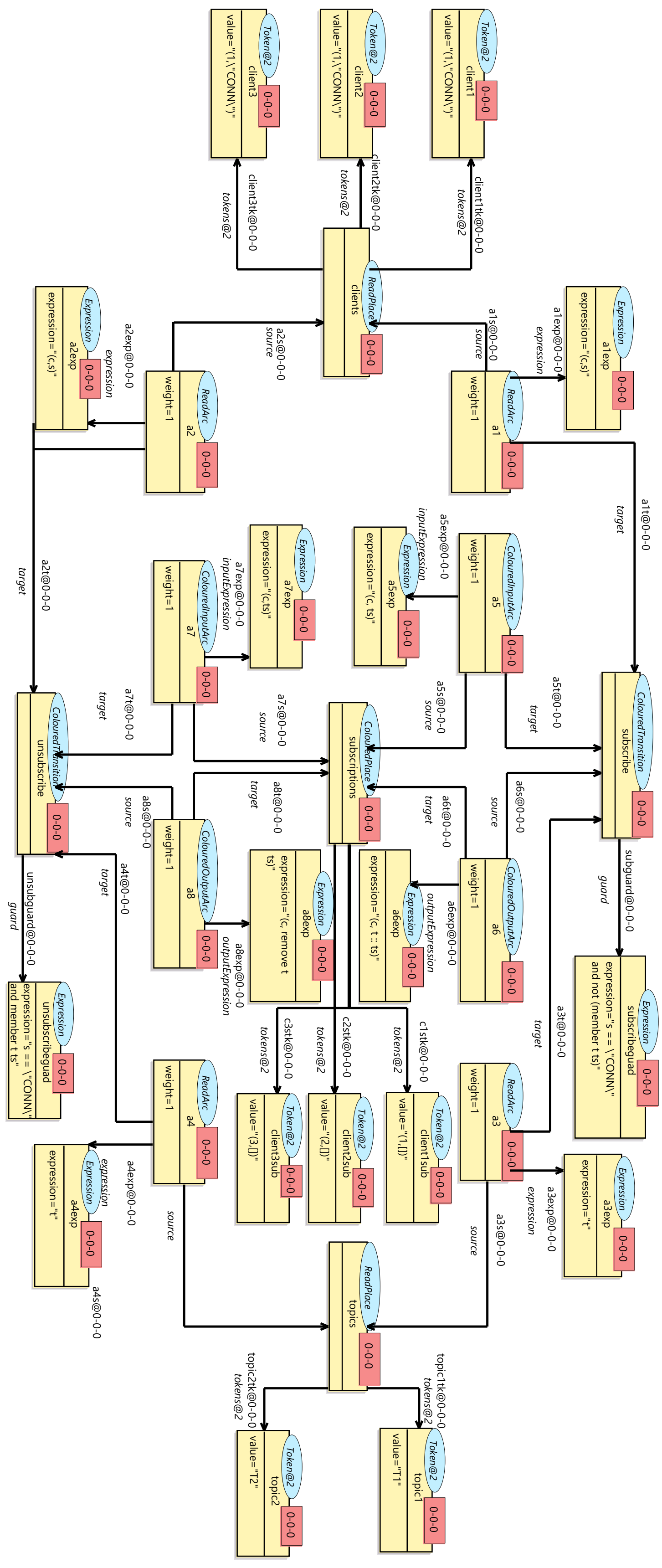

Token@2 client1 0-0-0 value="(1,\"CONN\")"
Token@2 client2 0-0-0 value="(1,\"CONN\")"
Token@2 client3 0-0-0 value="(1,\"CONN\")"
client1tk@0-0-0 tokens@2
client2tk@0-0-0 tokens@2
client3tk@0-0-0 tokens@2
ReadPlace clients 0-0-0
Expression a1exp 0-0-0 expression="(c,s)"
a1exp@0-0-0 expression
a1s@0-0-0 source
ReadArc a1 0-0-0 weight=1
a1t@0-0-0 target
Expression a2exp 0-0-0 expression="(c,s)"
a2exp@0-0-0 expression
a2s@0-0-0 source
ReadArc a2 0-0-0 weight=1
a2t@0-0-0 target
ColouredInputArc a5 0-0-0 weight=1
Expression a5exp 0-0-0 expression="(c, ts)"
a5exp@0-0-0 inputExpression
a5s@0-0-0 source
a5t@0-0-0 target
ColouredInputArc a7 0-0-0 weight=1
Expression a7exp 0-0-0 expression="(c,ts)"
a7exp@0-0-0 inputExpression
a7s@0-0-0 source
a7t@0-0-0 target
ColouredTransition subscribe 0-0-0
ColouredTransition unsubscribe 0-0-0
ColouredPlace subscriptions 0-0-0
subguard@0-0-0 guard
unsubguard@0-0-0 guard
Expression subscribeguad 0-0-0 expression="s == \"CONN\" and not (member t ts)"
Expression unsubscribeguad 0-0-0 expression="s == \"CONN\" and member t ts"
ColouredOutputArc a6 0-0-0 weight=1
Expression a6exp 0-0-0 expression="(c, t :: ts)"
a6exp@0-0-0 outputExpression
a6s@0-0-0 source
a6t@0-0-0 target
ColouredOutputArc a8 0-0-0 weight=1
Expression a8exp 0-0-0 expression="(c, remove t ts)"
a8exp@0-0-0 outputExpression
a8s@0-0-0 source
a8t@0-0-0 target
c1stk@0-0-0 tokens@2
c2stk@0-0-0 tokens@2
c3stk@0-0-0 tokens@2
Token@2 client1sub 0-0-0 value="(1,[])"
Token@2 client2sub 0-0-0 value="(2,[])"
Token@2 client3sub 0-0-0 value="(3,[])"
ReadArc a3 0-0-0 weight=1
a3t@0-0-0 target
a3s@0-0-0 source
Expression a3exp 0-0-0 expression="t"
a3exp@0-0-0 expression
ReadArc a4 0-0-0 weight=1
a4t@0-0-0 target
a4s@0-0-0 source
Expression a4exp 0-0-0 expression="t"
a4exp@0-0-0 expression
ReadPlace topics 0-0-0
topic1tk@0-0-0 tokens@2
topic2tk@0-0-0 tokens@2
Token@2 topic1 0-0-0 value="T1"
Token@2 topic2 0-0-0 value="T2"

Fig. 5.7: *Communication-flow* model in MultEcore syntax

and **ExtendedTransition**, respectively. Having this, the semantics of the multi-typed elements are given as a combination of the semantics of each respective model (*coloured-petri-nets* from the left-hand branch and *reset/inhibitor-petri-nets* from the right-hand branch) where the types are located.

In this chapter, we have described how the three shortcomings highlighted in Section 5.1 are alleviated. First, we have shown that the construction of a language family in the form of a multilevel hierarchy captures *modelling patterns*. In this case, levels 1 and 2 of the hierarchy illustrated in Figure 5.3 capture general PNs patterns that can be reused by models in different branches in levels below. Second, we have demonstrated how we can define *modelling restrictions*, for example, to delimit the use of certain constructs such as read arcs, where only read places can be connected to them (shown in Figure 5.4). Finally, we have depicted how, by constructing a multilevel hierarchy, *subtyping of elements* can be achieved in levels below. For instance, we have specified different forms of places (e. g., **ColouredPlace** and **ReadPlace**), arcs (e. g., **ResetArc** and **ColouredOutputArc**) and transitions (e. g., **RegularTransition** and **ColouredTransition**).

CHAPTER 6

RELATED WORK, CONCLUSIONS AND FUTURE WORK

In this chapter, we first discuss related work and summarise the thesis by revisiting the research questions and discussing the main contributions. Then, we outline directions for future work and conclude the thesis.

6.1 Related work

In this section, we summarise related work of the three main aspects of this thesis, namely, Multilevel Modelling (MLM), composition and execution.

6.1.1 *Multilevel Modelling*

In Chapter 2, we discussed the key concepts of MLM and described various MLM languages and tools. The most widely used approach to the specification of MLM frameworks is the Orthogonal Classification Architecture (OCA) [21]. It has been a reference since it emerged and mature tools such as Metadepth [65], OMME [241], Melanee [12] and OMLM [112] use it. However, we do not limit ourselves to this architecture but MultEcore follows the golden braid architecture [111] with Ecore as the topmost metamodel where each model at any level can have access to it or it can be transitively instantiated as many times as required to create a multilevel hierarchy. Other approaches, such as FMMLx [94] and DMLA [228] follow the same principle. Using a fixed metamodel can present some shortcomings since the user is forced to use such a metamodel and it is not allowed to specify more than one type for each element. To solve this, one can make use of the supplementary hierarchies where different types can be added to elements as explained in Chapter 2.

MultEcore can be positioned as a level-adjuvant approach since we classify elements in the multilevel hierarchy in a level-based manner. Level-adjuvant approaches such as the DPF Workbench [141], OMLM [112], Metadepth [65], Melanee [12], Dual Deep Modelling (DDM) [176] and FMMLx [50, 94] support some form of potency as described in Chapter 2. The MultEcore potency implementation (3-value range potency) has been influenced by Multi-potency [206] and star-potency [99]. The fact that we can use three values and the star to represent unbounded potency, has allowed us to give the capability to modellers to define multilevel hierarchies with the desired degree of precision. A more in-depth analysis of all the MLM approaches can be found in Chapter 2.

Regarding the specification of behaviour using MTs, in MLM there exist a few MLM approaches that handle behaviour. For instance, Metadepth, which takes advantage of the Epsilon languages, and Melanee, which is based on a service API and a plug-in mechanism, facilitate the execution of models. We further discuss these two approaches in Paper E.

One of the key features we have incorporated into MCMTs is the possibility to use parametric nested boxes. The specification of boxes in an MCMT rule allows it to be parametric and fit the concrete model configuration at runtime. Therefore, the rule can cover every possible situation regarding the replicated number of elements. For instance, the *Fire transition* MCMT rule shown in Figure 5.5 in Chapter 5 allows us to cover any number of arcs connected to a transition, and also any number of tokens that are present in each place. There exist other works in the literature that bring solutions to pattern definition and application in the context of graph transformations. Some examples are the star operator [148], the collection operator [104], recursion [105] and rule-nesting [9]. A more in-depth discussion of these approaches can be found in Paper E.

6.1.2 Composition

As described in Chapter 3, there exist several techniques for structure composition. The development of our approach for composition based on the use of supplementary hierarchies was highly influenced by the study of the merge and weaving operators. We studied the literature regarding the merge operator and analysed the shortcomings it presented, mainly, the fact that merging implied creating additional artefacts and losing the individuality of the elements prior to the operation (as discussed in Chapter 3). There exist some approaches in 2-level modelling that support some composition operator. In Melange [75], the operational semantics of a DSL involves the use of an action language to define methods that are statically introduced in the concepts of the DSL abstract syntax. In our approach, we define the semantics separately, by means of MCMTs, avoiding the need to change the abstract syntax (for us, the multilevel hierarchy) of the DSML. GeKo [135]) operates only on the structure, while our approach also provides support for the composition of behaviour by amalgamating MCMT rules. Further details on some approaches that support some composition operator operator in 2-level modelling (such as AToM^3 [73], Melange [75] and the GReAT tool [29]) and our approach based on supplementary hierarchies can be found in Paper D.

There are other approaches that are worth mentioning, for instance, Gromp [159] that supports modular languages design and language modules composition through a merge operator. The Atlas Model Weaver (AMW) [35] is a facility that implements a weaving operator for establishing relationships between elements from different models. These links are captured in a weaving model that conforms to a weaving metamodel, declaring the kind of relationship that can be modelled. It also provides a composition language that allows language designers to manually describe the composition of several language modules. LISA [162, 163] is an approach that makes use of the inheritance operator to achieve composition. In LISA, language modules are defined as attribute grammars that can have inheritance relationships among them. LISA supports modular language design and language modules composition, and uses ideas from object-oriented programming. MontiCore [133, 134] supports the construction of textual DSLs where the abstract and concrete syntaxes are defined in BNF-like grammars, and semantics are defined denotationally in a theorem prover. It implements the inheritance operator to support modular languages design and language modules composition.

Note that MultEcore relies on Maude for execution and that we perform the composition of MCMTs via an amalgamation process. CoorMaude [211] also relies on Maude and a coordination model that employs so-called *Coordination Roles* (CRs). The coordination model is an exogenous, transparent model, that promotes the separation of functional and coordination aspects. With such a separation, the system behaviour can be simulated, defining different objects configuring the system and different sequences of operations. In summary, CoorMaude performs the composition directly on Maude, while we do it in MultEcore prior to the transformation of the multilevel setting to Maude specification.

To the best of our knowledge, only Metadepth has contributed to structural composition in the MLM context. Metadepth supports facet-oriented modelling [72], where slots can be dynamically added or removed from elements, constraints and types through the notion of *facets*. Facet-oriented modelling, which is discussed in detail in Chapter 3, shares similarities with our composition approach based on supplementary hierarchies.

6.1.3 Execution and verification

In this thesis, we have focused on the execution of models based on the specification of MTs by means of Multilevel Coupled Model Transformations (MCMTs). Among the techniques discussed in Chapter 4, we can classify the MCMTs approach as specifying operational semantics.

In the context of MLM, Metadepth [65] is integrated with the Epsilon family of languages [87], which permits using both the EOL [131] as an action language to define behaviour for metamodels, and also the Epsilon Validation Language (EVL) [132] for expressing constraints. Metadepth compiles the models for execution and simulation purposes as we do in MultEcore. However, for the execution they are forced to flatten their multilevel language to a two-level version in order to run the models. As shown in Section 4.5, we rely on the Maude system [54] for execution. Maude is agnostic of the notion of levels as every element is an object in the specification, and hence we can directly use the multilevel setting avoiding the intermediate flatting step.

In the context of 2-level modelling, we find other approaches similar to our work that also support state space calculation and verification of temporal properties. Henshin [9, 223], is an EMF-based tool that can operate directly on EMF models. Furthermore, it supports modal μ-calculus [44] model checking by integrating the CADP [96] model checker and validation of the models through OCL invariants. GROOVE [126, 190] is implemented in Java under Eclipse and it is a graph-based transformation tool. It uses graphs to represent model states and transitions are performed via the application of graph production rules. To generate the state space, the GROOVE Simulator engine recursively computes and applies all enabled graph production rules at each state. GEMOC [43, 59] is an Eclipse-based tool built on top of EMF. It supports both the design and construction of DSMLs and also the execution of the models conforming to such DSMLs. The GEMOC execution framework provides an API that integrates any kind of metaprogramming approach used to define discrete-event operational semantics into an execution engine. In our case, we have taken advantage of the different analysis techniques supported by Maude, like reachability analysis, bounded and unbounded model checking of invariants and LTL formulas. We detail additional related work regarding execution and verification in Paper C and Paper E.

6.2 Research Questions revisited

The research work of this thesis has been dedicated to the MLM and Coloured Petri nets (CPNs) fields. In this section, we revisit our research questions (RQs) and discuss how we have addressed them.

RQ1: *How can MLM be used to alleviate the shortcomings of Coloured Petri nets and the CPN Tools?*

While the lack of focus on some ideas such as composition and verification within MLM was the primary motivation for this thesis, the fact that we identified some challenges in CPNs that could be solved by applying MLM techniques motivated us to apply our results in the context of the CPNs case study. We have demonstrated that MultEcore's flexibility in the horizontal dimension allows defining an arbitrary number of domain-specific languages. Furthermore, its vertical flexibility promotes the specification of families of languages in combination with the horizontal extension, giving rise to a variety of available DSML languages within the same multilevel hierarchy. Further details on vertical and horizontal flexibility are given in Paper C and in Paper F. Furthermore, we have successfully defined a multilevel hierarchy for a family of PN languages (detailed in Paper E) where one branch of it leads to define CPN models (presented in Chapter 5) together with their corresponding semantics. The semantics specification has been possible due to the major improvements we have introduced to the MCMTs.

RQ2: *How can reuse across related multilevel DSMLs be facilitated?*

We were aware that there existed a lot of research work regarding the reuse of (modelling) languages within traditional MDSE approaches and from other fields, such as Aspect-oriented programming or software language engineering. Our goal was to reflect some of the ideas present in the literature in our underlying theory and the MultEcore tool. We analysed how the merge and weaving operators had been used in the literature and decided to develop our own solution, which resulted in the notion of supplementary hierarchy to multi-type elements. Using supplementary hierarchies to achieve composition was part of our development and we used it not only as a structural resource to expand the multilevel hierarchy with new concepts but also to foster composition. The supplementary dimension has turned into an elegant solution to enhance reusability and modularity in a non-intrusive way across different MLM hierarchies. We compared our approach to other relevant and commonly accepted approaches to composition in the MLM community, like the linguistic extension approach [72].

RQ3: *How can the underlying theory be adapted to achieve the composition of MLM hierarchies and amalgamation of their respective MCMT rules?*

We extended the underlying theory to consider the composition of multilevel hierarchies and the amalgamation of the MCMT rules. The intuition behind the formal extension of the theory we made relies on being able to specify more than one typing chain and providing the model instances with several types, one from each typing chain. In practice, each multilevel hierarchy (the application, and the supplementary) would represent a typing chain. Note that we also allow specifying different typing chains within the same multilevel hierarchy if each branch represents a language that could be composed to multi-type elements at the instance level (see [204] and Chapter 5). We not only extended the formalisation, but also applied the ideas to a case study and developed a guided engine to semi-automatically produce the amalgamated MCMT rules. The details of the theoretical extensions, the case study and the description of the MultEcore amalgamation engine can be found in Paper D.

RQ4: *How can a term-rewriting engine like Maude be used as an execution engine for MCMTs with the goal to simulate and verify multilevel DSMLs?*

In MultEcore, it was already possible to specify MCMT rules using the MultEcore textual DSML [149]. However, there was no engine to rely on to execute these MCMTs in order to simulate the MLM hierarchies. In this regard, we had two choices. On one hand, we could implement an engine that: (i) takes care of the matching of the MCMT rules with the multilevel hierarchy; (ii) applies such matches to generate new model states; (iii) implements a state space generator and some property specification logic such as LTL or CTL to verify behavioural properties. This idea was proposed but not fully implemented by Macias [149]. Instead, we decided to take advantage of an existing system, specifically Maude, which was powerful enough to handle all the machinery we needed in MultEcore. This has resulted in the MultEcore-Maude infrastructure that allows us to execute models and verify them by integrating Maude as a background process into MultEcore, making the user agnostic of all the low-level details. This infrastructure and some of its applications can be explored in Papers C, E and F and in Chapter 5.

6.3 Summary of contributions

Following the constructive research method (detailed in Section 1.8) has allowed us to contribute in many aspects to the fields of CPNs and MLM, in an incremental way. In this section, we summarise the contributions to these areas.

6.3.1 Contributions to Coloured Petri nets

Our initial goal was to bring MLM to the next step towards the composition and execution of multilevel DSMLs. The fact that we realised that CPNs was presenting some shortcomings regarding extension and customisation of the language [217] motivated us to use it as a case study in which we could apply our MLM contributions. Thus, our first effort was to understand the language, familiarise ourselves with the CPN Tools and discern their strong and weak points, to then address them by applying MLM techniques. Our study of the tool consisted of specifying a realistic enough protocol (we chose the MQTT protocol), modelling it and performing verification on it. Choosing MQTT was not an arbitrary decision, as we were aware that the MQTT protocol was documented in natural language and presented some ambiguities. Therefore, we decided to develop an MQTT CPN model that constituted a formal specification of the protocol that could be simulated and verified, which turned into a journal publication (see Paper A).

The obtained results were satisfactory, but there was an inherent problem regarding the size of the computed state spaces of some of the scenarios we simulated for the first publication. This encouraged us to further work on the MQTT case, leading us to implement a method to alleviate the state explosion problem. We implemented the sweep-line method which deletes states from memory during state space exploration. We implemented this method within the CPN Tools using the SML language and, in addition, we incorporated support for certain CTL formulas that can be verified on-the-fly at the same time as the sweep-line method is running. The verification results were consistent with those obtained in the first publication and the simulated scenarios presented a substantial reduction in memory usage. These contributions were also published as a journal publication (see Paper B).

6.3.2 Contributions to Multilevel Modelling

The contributions made to the MLM field have been related to the MultEcore approach and its underlying theory. The MultEcore's state [152] that was taken as the starting point of this work allowed the user to specify multilevel hierarchies and sketch the MCMT rules. However, there were some tool limitations when defining the different models that shaped the hierarchy and the MCMTs were not executable at all.

Also, while the concept of supplementary hierarchy had been theoretically proposed [149, 154] it was not implemented within the MultEcore tool. We have implemented it in MultEcore and applied it to different cases (see [202, 204] and Papers D and F). In the following, we list the contributions made from both theoretical and practical perspectives:

- The concept of supplementary hierarchy has its origin in the concept of typing chain, present in the existing formalisation (see [245]). However, this was an unexploited feature and the formalisation did not take into account multiple typing chains (or supplementary hierarchies). Exploring the idea of multiple supplementary hierarchies led us to extend the formalisation not only to be able to define multiple typing chains, but also to combine them. The result was a formal extension that considered both the combination of MLM hierarchies and also the amalgamation of MCMTs to describe combined behaviour. Naturally, we applied these new incorporations to MultEcore, where the user can add and remove supplementary hierarchies in a natural, non-invasive and semi-automatic way, and carry on the composition of multilevel hierarchies and amalgamation of their MCMT rules (through a guided wizard). This work is covered by several publications: the formal details together with their application into a case study can be found in Paper D. The usage of the supplementary hierarchies for different purposes can be seen in Paper F and in [202, 204].

- The MCMTs were theoretically presented in [149, 154, 155] and the MultEcore tool supported the specification of MCMT rules. However, no engine was implemented that could execute the rules against the models to simulate them. Our goal was to make this possible, and we developed an initial infrastructure that connected MultEcore with Maude, which possessed the necessary functionalities that were required for execution. This preliminary infrastructure, with some limitations, was presented in Paper C [193]. For this publication, we had to improve the expressive power of the MCMTs due to the practical needs of the case study. These improvements enhanced the flexibility of the MCMTs for their application into the multilevel hierarchy, within the horizontal and vertical dimensions (see Paper C [Section 4.2] for more details), and incorporated a boxing mechanism to define submodel patterns. This mechanism allowed us to define parts of the MCMT rule that would be replicated to match an arbitrary number of times in the model. Still, this boxing mechanism was very limited, as one could only specify boxes on the left-hand side of the rule and nested boxes as well as cross-level boxes were not supported. The infrastructure built at this time had a bidirectional transformation between MultEcore and Maude, producing executable Maude files from MultEcore that the user had to run manually to produce XML output files. Then, the user had to load each XML file manually, creating the corresponding MultEcore model files.

- The current MultEcore-Maude infrastructure has been widely improved and extended with respect to the preliminary version presented in Paper C. The improvements were motivated by our final goal which was to achieve execution and verification and apply it to the CPNs case study via a defined multilevel infrastructure and the specification of the behavioural MCMT rules. Still, the CPNs complexity given by the programming language (SML in CPNs) made us reconsider an intermediate step. Such a step consisted of building a multilevel hierarchy to handle ordinary PNs with some extensions. Note that to finally handle the CPNs case we reused the multilevel hierarchy constructed for the PNs case described in Paper E. The CPNs can be seen as one of the available DSMLs of the PNs language family described by the multilevel hierarchy (see Chapter 5 for details). We successfully constructed such a hierarchy, specified the MCMT rules, and highly improved the infrastructure, hiding the Maude part to the user by a Maude process encapsulation within MultEcore. The MCMTs were again improved and extended with new features. It is important to mention that the MCMTs supported in the current MultEcore version, as presented in this thesis, have significantly evolved with respect to its original version presented in [149].

The current version of the MCMTs implements the following functionalities:

– Attributes definition and manipulation which bring additional expressiveness to the specification of behaviour. Examples of this are depicted in Papers D and E.

– Rule application conditions that add extra requirements for a rule to be applied. Conditions have been used, for instance, in Paper E.

– Nested parametric boxes to handle submodel patterns, improving expressiveness and reducing the proliferation of rules. The boxes can be used on left- and right-hand sides and their cardinalities can be computed by using an OCL expression (see, for instance, Paper E). Furthermore, there might be boxes crossing multiple levels (see Paper F for an example).

– Possibility to handle SML or OCL expressions, both in the models and in the MCMT rules, by parametrising which language is being used. Note that the OCL and SML supported versions are still basic and both languages can be used in the MCMTs for the manipulation of attribute values, for the specification of conditions, and to express the cardinality of the boxes. They can also be used to instantiate attributes in the multilevel hierarchy with a valid expression for the selected language. See Paper E and Chapter 5 for more details.

Furthermore, the produced Maude files are organised in a modular, parameterised way. This facilitates adding further languages by parametrising certain parts of the Maude representation files. Also, the user can use the supported Maude execution and verification tools directly from the MultEcore graphical model editor, and the results are directly obtained and graphically displayed in MultEcore. These improvements are detailed on Paper E, Chapter 4 and Chapter 5.

6.3.3 *Case studies*

To evaluate the contributions detailed in Sections 6.3.1 and 6.3.2 we used different case studies. The contributions related to the CPNs field that resulted from the study of the CPN Tools are reflected in the MQTT CPN model (in Papers A and B). The theoretical contribution for composition and amalgamation was evaluated with a case study where two multilevel hierarchies, one to handle process management and one that captures human-being aspects (in Paper D).

The incrementally built MultEcore-Maude infrastructure was first used to execute a multilevel DSML for Product Line Systems (in Paper C). The latest version of the infrastructure has been evaluated with: (i) a multilevel hierarchy for PNs (in Paper E); (ii) a solution to the Multi Process Challenge [5] where not only all the mandatory and optional requirements were fulfilled but also we extended the challenge with additional enhancements such as model execution (in Paper F); (iii) and finally the construction and execution of an extended version of the PNs multilevel hierarchy used originally in Paper E where a branch of the hierarchy captures CPNs (described in Chapter 5).

6.4 Future work

The work presented in this thesis opens several research lines for future work regarding CPNs and MLM. In this section, we discuss potential research directions for future work in these areas.

6.4.1 *Coloured Petri nets*

Even though our initial goal regarding CPNs was to face some of the challenges they presented and use them as a case study of our MLM approach, the work carried out in the CPNs field turned into two journal publications. The development of the MQTT CPN model (Paper A) was based on the most recent MQTT specification version (3.1.1. [30]) at that time. There may come new versions that would require slight modifications of the MQTT CPN model in order to keep it updated. While the model supported all the basic features specified in the MQTT specification, there is room for improvements, such as simulation of loss of packets and persistence of data. Note also that the model does not consider real time and the simulated scenarios should be appropriately interpreted. The key point of the model we have designed is to inspect the correct exchange of messages and focus on an accurate communication between the clients and the broker.

Regarding the sweep-line method implementation (Paper B), we experimented with a set of controlled scenarios, where, for instance, the number of clients, or the packet identifiers were scoped to have small controlled configurations. A bigger pool of scenarios would help to optimise the implementation of the sweep-line method and further analyse the *progress measure* [117]. This would be relevant to make other analyses and study, for example, how the reduction factor grows with the value of the parameter of the progress measure. Furthermore, there are also several possibilities for improving the implementation of the property-specific CTL model checking algorithms that we employ, where in the current version only a selection of properties is supported. This is because it is challenging to combine CTL model checking with the sweep-line method since conventional algorithms for CTL model checking propagates information backwards from a state to its predecessors. This follows the opposite workflow than the forward progress-first exploration that the sweep-line method performs.

6.4.2 *Multilevel Modelling*

Within the field of MLM, and using our MultEcore approach, we have mainly contributed to the composition and execution aspects. In the next subsections we explore future work in these directions.

6.4.2.1 *Composition*

Our formalisation allows us to deal with an arbitrary number of typing chains. However, we have so far explored the case of composing two multilevel hierarchies and the amalgamation of pairs of MCMT rules. Even though MultEcore's current version allows to specify three or more supplementary hierarchies, it would be interesting to evaluate the composition of such scenarios where not only compose multiple hierarchies, but also amalgamate three or more MCMT rules into one. This would also require improvements in the conflicting cases, where the prioritisation formulae given in Paper D currently consider two possible solutions depending on which rules the user prioritises. The amalgamation engine described in Section 4 of Paper D can also be further improved. First, from the design level, where more advanced techniques could be used to design the wizard and be more oriented towards providing a better user experience. Second, the resulting amalgamated MCMT rules obtained could be integrated into the MultEcore-Maude infrastructure.

The MultEcore-Maude infrastructure does not yet consider the transformation of several multilevel hierarchies, although its implementation should not be a major challenge. Note that the representation distance between the hierarchies and the MCMTs in MultEcore and its corresponding Maude representation is rather small. Thus, given that the MultEcore side already contains all the necessary information to handle composition and amalgamation, it would only be necessary to extend the Maude signature and adapt the transformer to take into account the new information.

It would be appropriate to develop case studies that encourage the behavioural execution of composed multilevel hierarchies to evaluate the practicality of the amalgamated MCMT rules that create new model states based on the composed model languages. To have a first intuition, the first case that should be tested is the case study shown in Section 4.5 of Paper D.

While we have demonstrated how MCMT rules can be combined in Paper D, these are rather simple. Currently, there are several limitations regarding amalgamation of more complex situations. For example, the amalgamation of MCMT rules similar to those used for the PNs language family that incorporates additional features provided by the programming languages cannot be amalgamated. Another example, is how to handle the amalgamation of MCMT rules that specify boxes, which we have not yet considered.

6.4.2.2 *MultEcore-Maude infrastructure for execution and verification*

We discuss in this subsection future work in the direction of the MultEcore-Maude infrastructure.

Petri nets family multilevel hierarchy The PNs family first presented in Paper E and then reused in Chapter 5 to model CPNs helped greatly to improve both the theoretical foundation and the implementation of the MultEcore framework. The execution and verification parts detailed in Paper E were tested with a gas station model. The verification we performed was based on a manually-abstracted version of such gas station model. Abstraction is key for the analysis of cases in which the reachable state space is infinite, and becomes also very important even when it is finite to improve efficiency. Future versions of the MultEcore-Maude infrastructure should support some user-friendly way of specifying abstracted versions of the models that are going to be verified.

Regarding the CPNs branch presented in Chapter 5, we have demonstrated how we can achieve some key points such as modelling patterns and restrictions, as well as easily extend and include new concepts. The design choice of combining the reset/inhibitor

PNs with CPNs was to take advantage of the already existing part which was used in Paper E. Note that CPN Tools supports reset and inhibitor arcs which we intentionally have used as examples to demonstrate how composition can help to easily incorporate new concepts. In other words, we could introduce new concepts in a hierarchy in the same way as we have added reset and inhibitor arcs to our CPNs model.

Extension of the MultEcore-Maude infrastructure The current state of the MultEcore-Maude infrastructure (Paper E) integrates a limited number of Maude functionalities, such as sequential execution given a certain number of steps, custom rule-based execution and the verification of LTL formulae. Still, Maude has several other functionalities that can be incorporated into the infrastructure, such as reachability analysis, a tool for the specification of abstracted versions of models and the customisation of the *strategy* language for execution [207, 208]. Some improvements can also be done in MultEcore and the infrastructure, mostly to enhance its usability. For instance, a more intelligent way to parameterise the language that is used (i) for the attributes in the multilevel hierarchy; (ii) in the MCMT rules; (iii) in the cardinality of the boxes; and (iv) in the conditions.

The MQTT model version in MultEcore represents an excerpt of the model originally submitted to Paper A. While the main aspects of CPNs, such as the use of a programming language as SML, have been successfully applied, the model should be extended so that it reaches an analogous state to the version created in the CPN Tools. Also, to enhance the verification capabilities, it could be interesting to provide Maude with a CTL implementation and test whether the sweep-line method (Paper B) can be extrapolated to this environment.

Preserving graphical positioning of unchanged model elements While the infrastructure automatically creates the new model states in MultEcore upon applying MCMT rules, and provides their graphical representation, the elements are not graphically placed based on their previous distribution. This feature would highly improve the simulation of the multilevel hierarchies, and would even open the path for implementing other interesting techniques, such as highlighting the newly created elements by some layering technique or some kind of execution animation. There are different alternatives to achieve this. While currently in MultEcore the new states are freshly created, we could preserve and reuse the files that contain the graphical distribution (similarly as done by GEMOC [43]) of elements and synchronise it with the new model state, which would conserve the unmodified elements' positions. Another alternative technique would be to keep the information that is stored in the graphical representation file in another artefact. This artefact would maintain the information of the existing elements together with their graphical distribution and would serve to preserve this information for new graphical representation files.

State space viewer Regarding verification, the current implementation allows the user to verify LTL properties in a Maude-like syntax from MultEcore. An important feature, especially to facilitate the user to carry out model checking, would be the creation of an LTL-like language in MultEcore using some textual DSML that facilitates the specification and verification of the behavioural properties. Another integration that could be incorporated is a graphical viewer of the state space of the system being modelled, such that the user could inspect the different model states and analyse them, for example like GROOVE does [190].

Performance and efficiency Our main goal has never been to evaluate the performance of the infrastructure that takes care of the execution and verification parts. Moreover, we have not aimed to compare the infrastructure with other mature non-MLM-based tools which goal is execution and verification, such as the CPN Tools itself which contains many optimisations in this regard. Although Maude is one of the most efficient engines for rewriting, the current MultEcore-Maude infrastructure and representation present a high level of complexity. Note that our approach focuses on enhancing flexibility, which comes with a cost in performance, as the more variables the system has, the more costly it is to find valid matches for all of these variables. An interesting research line is to evaluate the performance of the framework, and how to balance flexibility and generality of the approach with executions that are feasible and reasonable in time cost. Also, as mentioned in Section 6.4.2.2, other techniques can be used to reduce the consumption time, such as defining abstracted versions of the system.

6.5 Conclusions

In this thesis, we have presented our infrastructure for the definition, composition and execution of multilevel DSMLs. These DSMLs are encapsulated into multilevel hierarchies which can be defined in a flexible way using the MultEcore tool. This definition is supported by a graphical editor, and functionalities like the use of a three-value potency allow models to be defined as precise or generic as needed. In addition to enable the definition of the structural dimension of multilevel hierarchies, we facilitate the specification of the behaviour of domain-specific modelling languages by means MCMT rules.

A key contribution in this thesis is the possibility to compose multilevel hierarchies and amalgamate the involved MCMT rules. This composition, by using supplementary hierarchies, enhances reusability and modularity. Our approach for supplementary hierarchies has been formally described and successfully implemented in MultEcore, where both model elements and MCMT rule elements may have multiple types.

Another fundamental contribution has been the construction of an infrastructure that connects MultEcore with Maude to further explore the behavioural dimension of multilevel hierarchies. The integration of Maude as a process within MultEcore allows us to execute and verify the instance models directly from the MultEcore interface and obtain new results automatically.

BIBLIOGRAPHY

[1] R. Acerbis, A. Bongio, M. Brambilla, M. Tisi, S. Ceri, and E. Tosetti. Developing eBusiness Solutions with a Model Driven Approach: The Case of Acer EMEA. In *Web Engineering, 7th International Conference, ICWE 2007, Como, Italy, July 16-20, 2007, Proceedings*, pages 539–544, 2007. 1.2

[2] M. A. Adamski, A. Karatkevich, and M. Wegrzyn. *Design of embedded control systems*, volume 267. Springer, 2005. 5.1

[3] J. P. A. Almeida, C. M. Fonseca, and V. A. de Carvalho. A Comprehensive Formal Theory for Multi-level Conceptual Modeling. In H. C. Mayr, G. Guizzardi, H. Ma, and O. Pastor, editors, *Conceptual Modeling - 36th International Conference, ER 2017, Valencia, Spain, November 6-9, 2017, Proceedings*, volume 10650 of *Lecture Notes in Computer Science*, pages 280–294. Springer, 2017. 2.3.1, 2.4.1

[4] J. P. A. Almeida, A. Rutle, M. Wimmer, and T. Kühne. The MULTI process challenge. In L. Burgueño, A. Pretschner, S. Voss, M. Chaudron, J. Kienzle, M. Völter, S. Gérard, M. Zahedi, E. Bousse, A. Rensink, F. Polack, G. Engels, and G. Kappel, editors, *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019*, pages 164–167. IEEE, 2019. 2.4.3

[5] J. P. A. Almeida, A. Rutle, M. Wimmer, and T. Kühne. The MULTI Process Challenge. *Enterprise Modelling and Information Systems Architectures*, 2021. Available at https://bit.ly/3b3cQZV. 1.8, 1.9.6, 1.10, 2.4.3, 6.3.3

[6] J. M. Álvarez, A. Evans, and P. Sammut. Mapping between Levels in the Metamodel Architecture. In M. Gogolla and C. Kobryn, editors, *«UML» 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings*, volume 2185 of *Lecture Notes in Computer Science*, pages 34–46. Springer, 2001. 2.2

[7] F. Arbab. The IWIM Model for Coordination of Concurrent Activities. In *Coordination Languages and Models, First International Conference, COORDINATION '96, Cesena, Italy, April 15-17, 1996, Proceedings*, pages 34–56, 1996. 3.3.2

[8] F. Arbab. Reo: a channel-based coordination model for component composition. *Math. Struct. Comput. Sci.*, 14(3):329–366, 2004. 3.3.2

[9] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In D. C. Petriu, N. Rouquette, and Ø. Haugen, editors, *Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I*, volume 6394 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 2010. 4.4, 6.1.1, 6.1.3

[10] T. Asikainen and T. Männistö. Nivel: a metamodelling language with a formal semantics. *Softw. Syst. Model.*, 8(4):521–549, 2009. 2.4.1

[11] C. Atkinson. Meta-modeling for distributed object environments. In *1st International Enterprise Distributed Object Computing Conference (EDOC '97), 24-26 October 1997, Gold Coast, Australia, Proceedings*, page 90. IEEE Computer Society, 1997. 2.2

[12] C. Atkinson and R. Gerbig. Flexible Deep Modeling with Melanee. In S. Betz and U. Reimer, editors, *Modellierung 2016*, volume 255 of *LNI*, pages 117–122, Bonn, 2016. Gesellschaft für Informatik. 1.3, 2.3.1, 2.4.2, 4.2.2, 6.1.1

[13] C. Atkinson, R. Gerbig, and T. Kühne. Comparing multi-level modeling approaches. In C. Atkinson, G. Grossmann, T. Kühne, and J. de Lara, editors, *Proceedings of the Workshop on Multi-Level Modelling co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2014), Valencia, Spain, September 28, 2014*, volume 1286 of *CEUR Workshop Proceedings*, pages 53–61. CEUR-WS.org, 2014. 2.2, 2.3.1, 2.3.1, 2.4.3

[14] C. Atkinson, R. Gerbig, and N. Metzger. On the Execution of Deep Models. In *1st International Workshop on Executable Modeling co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS 2015).*, pages 28–33, 2015. 1.5

[15] C. Atkinson and T. Kühne. Meta-level independent modelling. In *International Workshop on Model Engineering at 14th European Conference on Object-Oriented Programming*, volume 12, page 16, 2000. 1.3, 2.2

[16] C. Atkinson and T. Kühne. Strict Profiles: Why and How. In A. Evans, S. Kent, and B. Selic, editors, *«UML» 2000 - The Unified Modeling Language, Advancing the Standard, Third International Conference, York, UK, October 2-6, 2000, Proceedings*, volume 1939 of *Lecture Notes in Computer Science*, pages 309–322. Springer, 2000. 2.2

[17] C. Atkinson and T. Kühne. Processes and Products in a Multi-Level Metamodeling Architecture. *Int. J. Softw. Eng. Knowl. Eng.*, 11(6):761–783, 2001. 1.3, 2.2

[18] C. Atkinson and T. Kühne. The Essence of Multilevel Metamodeling. In M. Gogolla and C. Kobryn, editors, *«UML» 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings*, volume 2185 of *Lecture Notes in Computer Science*, pages 19–33. Springer, 2001. 1.2, 2.2, 2.3.2, 2.4.1, 2.4.2

[19] C. Atkinson and T. Kühne. Rearchitecting the UML infrastructure. *ACM Trans. Model. Comput. Simul.*, 12(4):290–321, 2002. 2.2, 2.3.2

[20] C. Atkinson and T. Kühne. Model-Driven Development: A Metamodeling Foundation. *IEEE Softw.*, 20(5):36–41, 2003. 1.3, 2.1, 2.2

[21] C. Atkinson and T. Kühne. Concepts for Comparing Modeling Tool Architectures. In L. C. Briand and C. Williams, editors, *Model Driven Engineering Languages and*

Systems, 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005, Proceedings, volume 3713 of *Lecture Notes in Computer Science*, pages 398–413. Springer, 2005. 2.1, 2.2, 6.1.1

[22] C. Atkinson and T. Kühne. Reducing accidental complexity in domain models. *Software & Systems Modeling*, 7(3):345–359, 2008. 1.2, 1.3, 2.2

[23] C. Atkinson and T. Kühne. In defence of deep modelling. *Inf. Softw. Technol.*, 64:36–51, 2015. 1.2, 1.3, 2.4.2

[24] C. Atkinson and T. Kühne. On Evaluating Multi-level Modeling. In *Proceedings of MULTI @ MODELS*, pages 274–277, 2017. 1.3

[25] C. Atkinson, T. Kühne, and J. de Lara. Editorial to the theme issue on multi-level modeling. *Softw. Syst. Model.*, 17(1):163–165, 2018. 2.2

[26] C. Atkinson, T. Kühne, and B. Henderson-Sellers. Systematic stereotype usage. *Softw. Syst. Model.*, 2(3):153–163, 2003. 2.2

[27] C. W. Bachman and M. Daya. The Role Concept in Data Models. In *Proceedings of the Third International Conference on Very Large Data Bases - Volume 3*, VLDB '77, page 464–476. VLDB Endowment, 1977. 3.2.4

[28] C. Baier and J.-P. Katoen. *Principles of model checking*. MIT press, 2008. 1.5, 4.4

[29] D. Balasubramanian, A. Narayanan, S. Neema, F. Shi, R. Thibodeaux, and G. Karsai. A Subgraph Operator for Graph Transformation Languages. *ECEASST*, 6, 2007. 3.3.1, 6.1.2

[30] A. Banks and R. Gupta. MQTT Version 3.1.1. *OASIS standard*, 29, 2014. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html. 1.9.1, 5.2, 6.4.1

[31] W. Bast, M. Murphree, L. Michael, K. Duddy, M. Belaunde, C. Griffin, S. Sendall, V. Didier, J. Steel, L. Tratt, et al. MOF QVT final adopted specification: meta object facility (MOF) 2.0 query/view/transformation specification. *Object Management Group specification*, 2005. 4.2.2

[32] L. Bernardinello and F. de Cindio. A survey of basic net models and modular net classes. In *Advances in Petri Nets 1992, The DEMON Project*, pages 304–351. Springer, 1992. 5.1

[33] J. Bettin and T. Clark. Advanced modelling made simple with the Gmodel metalanguage. In J. Bézivin, R. M. Soley, and A. Vallecillo, editors, *Proceedings of the First International Workshop on Model-Driven Interoperability, MDI@MoDELS 2010, Oslo, Norway, October 3-5, 2010*, pages 79–88. ACM, 2010. 2.4.1

[34] J. Bézivin. On the unification power of models. *Softw. Syst. Model.*, 4(2):171–188, 2005. 1.1, 2.1

[35] J. Bézivin, S. Bouzitouna, M. D. D. Fabro, M. Gervais, F. Jouault, D. S. Kolovos, I. Kurtev, and R. F. Paige. A canonical scheme for model composition. In *Model Driven Architecture - Foundations and Applications, 2nd European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006, Proceedings*, pages 346–360, 2006. 2.5.1, 3.2.2, 6.1.2

[36] M. Biehl. Literature study on model transformations. *Royal Institute of Technology, Tech. Rep. ISRN/KTH/MMK*, 291, 2010. 4.1

[37] E. Biermann, H. Ehrig, C. Ermel, U. Golas, and G. Taentzer. Parallel Independence of Amalgamated Graph Transformations Applied to Model Transformation. In *Graph Transformations and Model-Driven Engineering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday*, pages 121–140, 2010. 3.3.1

[38] J. Billington and M. Diaz. *Application of Petri nets to Communication Networks: Advances in Petri nets*, volume 1605. Springer Science & Business Media, 1999. 1.6, 5.1

[39] P. Boehm, H. Fonio, and A. Habel. Amalgamation of Graph Transformations: A Synchronization Mechanism. *J. Comput. Syst. Sci.*, 34(2/3):377–408, 1987. 3.3.1

[40] M. M. Bonsangue, F. Arbab, J. W. de Bakker, J. J. M. M. Rutten, A. Secutella, and G. Zavattaro. A transition system semantics for the control-driven coordination language MANIFOLD. *Theor. Comput. Sci.*, 240(1):3–47, 2000. 3.3.2

[41] G. Booch, J. Rumbaugh, and I. Jacobson. Unified modeling language semantics and notation guide 1.0. *Rational Software Corporation, San Jose, California*, 1997. 2.1, 2.2

[42] A. Boronat, J. Á. Carsí, and I. Ramos. Algebraic specification of a model transformation engine. In *International Conference on Fundamental Approaches to Software Engineering*, pages 262–277. Springer, 2006. 4.2.2

[43] E. Bousse, T. Degueule, D. Vojtisek, T. Mayerhofer, J. DeAntoni, and B. Combemale. Execution framework of the GEMOC studio (tool demo). In T. van der Storm, E. Balland, and D. Varró, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*, pages 84–89. ACM, 2016. 1.5, 4.4, 6.1.3, 6.4.2.2

[44] J. C. Bradfield and C. Stirling. Modal mu-calculi. In P. Blackburn, J. F. A. K. van Benthem, and F. Wolter, editors, *Handbook of Modal Logic*, volume 3 of *Studies in logic and practical reasoning*, pages 721–756. North-Holland, 2007. 6.1.3

[45] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice, Second Edition*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2017. 1.1, 1.2

[46] J. Bruel, B. Combemale, E. Guerra, J. Jézéquel, J. Kienzle, J. de Lara, G. Mussbacher, E. Syriani, and H. Vangheluwe. Comparing and classifying model transformation reuse approaches across metamodels. *Softw. Syst. Model.*, 19(2):441–465, 2020. 2.2

[47] B. R. Bryant, J. Gray, M. Mernik, P. J. Clarke, R. B. France, and G. Karsai. Challenges and directions in formalizing the semantics of modeling languages. *Comput. Sci. Inf. Syst.*, 8(2):225–253, 2011. 4.2

[48] A. Bucchiarone, A. Cicchetti, and A. Marconi. Exploiting Multi-level Modelling for Designing and Deploying Gameful Systems. In M. Kessentini, T. Yue, A. Pretschner, S. Voss, and L. Burgueño, editors, *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2019, Munich, Germany, September 15-20, 2019*, pages 34–44. IEEE, 2019. 1.3

[49] L. Cardelli. Structural Subtyping and the Notion of Power Type. In J. Ferrante and P. Mager, editors, *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, pages 70–79. ACM Press, 1988. 2.3.1, 2.3.2, 2.4.1

[50] T. Clark, P. Sammut, and J. S. Willans. Applied metamodelling: A foundation for language driven development (third edition). *CoRR*, abs/1505.00149, 2015. 2.3.1, 2.4.1, 6.1.1

[51] T. Clark and J. Warmer. *Object Modeling With the OCL: The Rationale Behind the Object Constraint Language*, volume 2263. Springer, 2003. 1.9.5, 2.4.2, 4.5

[52] T. Clark and J. Willans. Software Language Engineering with XMF and XModeler. *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, 2:311–340, 01 2012. 2.4.1

[53] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martı-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002. 4.5

[54] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007. 1.7, 1.10, 4.4, 4.5, 4.5, 6.1.3

[55] T. Cleenewerck. Component-Based DSL Development. In *Generative Programming and Component Engineering, Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003, Proceedings*, pages 245–264, 2003. 4.2.1

[56] T. Cleenewerck and I. Kurtev. Separation of concerns in translational semantics for DSLs in model engineering. In *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC), Seoul, Korea, March 11-15, 2007*, pages 985–992, 2007. 4.2.1

[57] A. Cleeremans, D. Servan-Schreiber, and J. L. McClelland. Finite State Automata and Simple Recurrent Networks. *Neural Comput.*, 1(3):372–381, 1989. 4.4

[58] P. Coad. Object-oriented patterns. *Commun. ACM*, 35(9):152–159, 1992. 2.2

[59] B. Combemale, O. Barais, and A. Wortmann. Language Engineering with the GEMOC Studio. In *2017 IEEE International Conference on Software Architecture Workshops, ICSA Workshops 2017, Gothenburg, Sweden, April 5-7, 2017*, pages 189–191. IEEE Computer Society, 2017. 4.4, 6.1.3

[60] T. M. W. Committee. The MULTI Process Challenge. *CEUR Workshop Proceedings*, 2018. Available at https://bit.ly/3gKalPD. 2.4.3

[61] P. Cousot. Abstract Interpretation. *ACM Comput. Surv.*, 28(2):324–328, 1996. 4.4

[62] CPN tools. http://cpntools.org/. 5.1

[63] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–646, 2006. 2.5.2, 4.1

[64] V. A. de Carvalho and J. P. A. Almeida. Toward a well-founded theory for multi-level conceptual modeling. *Softw. Syst. Model.*, 17(1):205–231, 2018. 2.3.1, 2.4.1

[65] J. de Lara and E. Guerra. Deep meta-modelling with MetaDepth. In *Objects, Models, Components, Patterns*, volume 6141 of *Lecture Notes in Computer Science*, pages 1–20. Springer International Publishing, July 2010. 1.3, 1.5, 2.3.1, 2.4.2, 3.2.4, 4.2.2, 6.1.1, 6.1.3

[66] J. de Lara and E. Guerra. Generic Meta-modelling with Concepts, Templates and Mixin Layers. In *Model Driven Engineering Languages and Systems - 13th International Conference, MODELS*, pages 16–30, 2010. 1.3, 2.2

[67] J. de Lara and E. Guerra. *A Posteriori* Typing for Model-Driven Engineering: Concepts, Analysis, and Applications. *ACM Trans. Softw. Eng. Methodol.*, 25(4):31:1–31:60, 2017. 2.2

[68] J. de Lara and E. Guerra. Refactoring Multi-Level Models. *ACM Trans. Softw. Eng. Methodol.*, 27(4):17:1–17:56, 2018. 2.2, 2.4.2

[69] J. de Lara and E. Guerra. Multi-level Model Product Lines - Open and Closed Variability for Modelling Language Families. In H. Wehrheim and J. Cabot, editors, *Fundamental Approaches to Software Engineering - 23rd International Conference, FASE 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, volume 12076 of *Lecture Notes in Computer Science*, pages 161–181. Springer, 2020. 2.2

[70] J. de Lara, E. Guerra, R. Cobos, and J. Moreno-Llorena. Extending Deep Meta-Modelling for Practical Model-Driven Engineering. *Comput. J.*, 57(1):36–58, 2014. 2.2, 2.3.2

[71] J. de Lara, E. Guerra, and J. S. Cuadrado. Model-driven engineering with domain-specific meta-modelling languages. *Softw. Syst. Model.*, 14(1):429–459, 2015. 1.2, 1.3, 2.2, 2.4.2

[72] J. de Lara, E. Guerra, J. Kienzle, and Y. Hattab. Facet-oriented modelling: open objects for model-driven engineering. In D. Pearce, T. Mayerhofer, and F. Steimann, editors, *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018, Boston, MA, USA, November 05-06, 2018*, pages 147–159. ACM, 2018. 2.5.1, 3.2.4, 3.4.1, 3.4.1, 6.1.2, 6.2

[73] J. de Lara and H. Vangheluwe. AToM 3: A Tool for Multi-formalism and Meta-modelling. In *International Conference on Fundamental Approaches to Software Engineering*, pages 174–188. Springer, 2002. 3.3.1, 6.1.2

[74] J. de Lara Jaramillo, C. Ermel, G. Taentzer, and K. Ehrig. Parallel Graph Transformation for Model Simulation applied to Timed Transition Petri Nets. *Electron. Notes Theor. Comput. Sci.*, 109:17–29, 2004. 3.3.1

[75] T. Degueule, B. Combemale, A. Blouin, O. Barais, and J.-M. Jézéquel. Melange: A meta-language for modular and reusable development of dsls. In *Proceedings of the 2015 SLE Conference*, pages 25–36. ACM, 2015. 1.4, 3.2.1, 6.1.2

[76] A. Demuth, R. E. Lopez-Herrejon, and A. Egyed. Cross-layer modeler: a tool for flexible multilevel modeling with consistency checking. In T. Gyimóthy and A. Zeller, editors, *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 452–455. ACM, 2011. 2.4.2

[77] J. Desel, W. Reisig, and G. Rozenberg, editors. *Lectures on Concurrency and Petri Nets, Advances in Petri Nets*, volume 3018 of *LNCS*. Springer, 2004. 1.6, 5.1

[78] G. Dodig Crnkovic. *Constructive Research and Info-Computational Knowledge Generation*, volume 314, pages 359–380. Springer International Publishing, 01 1970. 1.8

[79] F. Durán, S. Eker, S. Escobar, N. Martí-Oliet, J. Meseguer, R. Rubio, and C. L. Talcott. Programming and symbolic computation in Maude. *J. Log. Algebraic Methods Program.*, 110, 2020. 4.5

[80] F. Durán and H. Garavel. The rewrite engines competitions: A rectrospective. In D. Beyer, M. Huisman, F. Kordon, and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Proceedings, Part III*, volume 11429 of *Lecture Notes in Computer Science*, pages 93–100. Springer, 2019. 4.5

[81] F. Durán, A. Moreno-Delgado, F. Orejas, and S. Zschaler. Amalgamation of domain specific languages with behaviour. *Journal of Logical and Algebraic Methods in Programming*, 86:208–235, 2017. 1.4, 2.5.1, 3.2.2

[82] S. Easterbrook, J. Singer, M. D. Storey, and D. E. Damian. Selecting Empirical Methods for Software Engineering Research. In *Guide to Advanced Empirical Software Engineering*, pages 285–311. Springer International Publishing, 2008. 1.8

[83] H. Ehrig, K. Ehrig, C. Ermel, F. Hermann, and G. Taentzer. Information preserving bidirectional model transformations. In M. B. Dwyer and A. Lopes, editors, *Fundamental Approaches to Software Engineering, 10th International Conference, FASE 2007, Held as Part of the Joint European Conferences, on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4422 of *Lecture Notes in Computer Science*, pages 72–86. Springer, 2007. 2.5.2

[84] H. Ehrig, F. Hermann, and U. Prange. Cospan DPO approach: An alternative for DPO graph transformations. *Bulletin of the EATCS*, 98:139–149, 2009. 3.4.2

[85] E. A. Emerson. Temporal and Modal Logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 995–1072. 1990. 4.4

[86] E. A. Emerson and J. Srinivasan. Branching time temporal logic. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop, Noordwijkerhout, The Netherlands, May 30 - June 3, 1988, Proceedings*, pages 123–172, 1988. 4.4

[87] Epsilon. http://www.eclipse.org/epsilon/, 2012. 2.4.2, 6.1.3

[88] J. M. Fernandes and O. Belo. Modeling multi-agent systems activities through Colored Petri nets. In *16th IASTED International Conference on Applied Infomatics (AI'98)*, pages 17–20, 1998. 5.1

[89] W. J. Fokkink. Process Algebra: An Algebraic Theory of Concurrency. In *Algebraic Informatics, Third International Conference, CAI 2009, Thessaloniki, Greece, May 19-22, 2009, Proceedings*, pages 47–77, 2009. 4.4

[90] C. Fonseca. *ML2: An Expressive Multi-Level Conceptual Modeling Language*. PhD thesis, Universidad Federal Do Espirito Santo, 09 2017. 2.3.1, 2.4.1

[91] C. M. Fonseca, J. P. A. Almeida, G. Guizzardi, and V. A. de Carvalho. Multi-level conceptual modeling: From a formal theory to a well-founded language. In *Conceptual Modeling - 37th International Conference, ER 2018, Xi'an, China, October 22-25, 2018, Proceedings*, volume 11157 of *Lecture Notes in Computer Science*, pages 409–423. Springer, 2018. 2.4.1

[92] R. B. France, F. Fleurey, R. Reddy, B. Baudry, and S. Ghosh. Providing support for model composition in metamodels. In *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007), 15-19 October 2007, Annapolis, Maryland, USA*, pages 253–266, 2007. 2.5.1, 3.1

[93] R. B. France and B. Rumpe. Domain specific modeling. *Softw. Syst. Model.*, 4(1):1–3, 2005. 1.2

[94] U. Frank. Multilevel Modeling - Toward a New Paradigm of Conceptual Modeling and Information Systems Design. *Bus. Inf. Syst. Eng.*, 6(6):319–337, 2014. 2.3.1, 2.4.1, 6.1.1

[95] D. M. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the Temporal Analysis of Fairness. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nevada, USA, January 1980*, pages 163–173, 1980. 4.4

[96] H. Garavel, R. Mateescu, F. Lang, and W. Serwe. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In W. Damm and H. Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 158–163. Springer, 2007. 6.1.3

[97] H. Garavel, M. Tabikh, and I. Arrada. Benchmarking implementations of term rewriting and pattern matching in algebraic, functional, and object-oriented languages - the 4th rewrite engines competition. In V. Rusu, editor, *Rewriting Logic and Its Applications - 12th International Workshop, WRLA 2018, Held as a Satellite Event of ETAPS, Proceedings*, volume 11152 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2018. 4.5

[98] D. Gelernter. Generative Communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985. 3.3.2

[99] R. Gerbig. *The Level-agnostic Modeling Language: Language Specification and Tool Implementation*. PhD thesis, Universität Mannheim, 2011. 2.3.2, 6.1.1

[100] R. Gerbig, C. Atkinson, J. de Lara, and E. Guerra. A Feature-based Comparison of Melanee and Metadepth. In C. Atkinson, G. Grossmann, and T. Clark, editors, *Proceedings of the 3rd International Workshop on Multi-Level Modelling co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2016), Saint-Malo, France, October 4, 2016*, volume 1722 of *CEUR Workshop Proceedings*, pages 25–34. CEUR-WS.org, 2016. 2.2, 2.4.3

[101] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of software engineering*. Prentice Hall, 1991. 1.1

[102] M. Goedicke, B. Enders, T. Meyer, and G. Taentzer. ViewPoint-Oriented Software Development: Tool Support for Integrating Multiple Perspectives by Distributed Graph Transformation. In *Tools and Algorithms for Construction and Analysis of Systems, 6th International Conference, TACAS 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS*, pages 43–47, 2000. 3.3.1

[103] C. Gonzalez-Perez and B. Henderson-Sellers. A powertype-based metamodelling framework. *Softw. Syst. Model.*, 5(1):72–90, 2006. 1.3, 2.2

[104] R. Grønmo, S. Krogdahl, and B. Møller-Pedersen. A collection operator for graph transformation. *Software and Systems Modeling*, 12(1):121–144, 2013. 2.5.2, 6.1.1

[105] E. Guerra and J. de Lara. Adding Recursion to Graph Transformation. *ECEASST*, 6, 2007. 6.1.1

[106] E. Guerra and J. de Lara. On the Quest for Flexible Modelling. In A. Wasowski, R. F. Paige, and Ø. Haugen, editors, *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018, Copenhagen, Denmark, October 14-19, 2018*, pages 23–33. ACM, 2018. 2.2, 2.4.2

[107] G. Gupta and E. Pontelli. Specification, Implementation, and Verification of Domain Specific Languages: A Logic Programming-Based Approach. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I*, pages 211–239, 2002. 4.2

[108] A. Halder and A. Venkateswarlu. A study of petri nets modeling analysis and simulation. *Department of Aerospace Engineering Indian Institute of Technology Kharagpur, India*, 2006. 5.3.2

[109] B. Henderson-Sellers, T. Clark, and C. Gonzalez-Perez. On the search for a level-agnostic modelling language. In C. Salinesi, M. C. Norrie, and O. Pastor, editors, *Advanced Information Systems Engineering - 25th International Conference, CAiSE 2013, Valencia, Spain, June 17-21, 2013. Proceedings*, volume 7908 of *Lecture Notes in Computer Science*, pages 240–255. Springer, 2013. 2.3.1

[110] C. A. R. Hoare and N. Wirth. An Axiomatic Definition of the Programming Language PASCAL. *Acta Informatica*, 2:335–355, 1973. 4.2.3

[111] D. R. Hofstadter et al. *Gödel, Escher, Bach: an eternal golden braid*, volume 13. Basic books New York, 1979. 2.4.1, 6.1.1

[112] M. Igamberdiev, G. Grossmann, M. Selway, and M. Stumptner. An integrated multi-level modeling approach for industrial-scale data interoperability. *Softw. Syst. Model.*, 17(1):269–294, 2018. 2.3.1, 2.4.2, 6.1.1

[113] S. P. Jácome-Guerrero and J. de Lara. *TOTEM*: Reconciling multi-level modelling with standard two-level modelling. *Comput. Stand. Interfaces*, 69:103390, 2020. 2.2, 2.4.2, 2.4.3

[114] M. Jarke, R. Gallersdörfer, M. A. Jeusfeld, and M. Staudt. ConceptBase - A Deductive Object Base for Meta Data Management. *J. Intell. Inf. Syst.*, 4(2):167–192, 1995. 2.4.1, 2.4.1

[115] M. Jarke, M. A. Jeusfeld, H. W. Nissen, C. Quix, and M. Staudt. Metamodelling with Datalog and Classes: ConceptBase at the Age of 21. In M. C. Norrie and M. Grossniklaus, editors, *Object Databases, Second International Conference, ICOODB 2009, Zurich, Switzerland, July 1-3, 2009. Revised Papers*, volume 5936 of *Lecture Notes in Computer Science*, pages 95–112. Springer, 2009. 2.4.1

[116] K. Jensen and L. M. Kristensen. *Coloured Petri nets: modelling and validation of concurrent systems*. Springer Science & Business Media, 2009. 1.6, 5.1

[117] K. Jensen, L. M. Kristensen, and T. Mailund. The sweep-line state space exploration method. *Theor. Comput. Sci.*, 429:169–179, 2012. 4.4, 6.4.1

[118] K. Jensen, L. M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 9(3):213–254, Jun 2007. 1.6, 5.1, 5.3.2

[119] K. Jensen and G. Rozenberg. *High-level Petri nets: theory and application*. Springer Science & Business Media, 2012. 1.6, 5.1

[120] M. A. Jeusfeld. DeepTelos Demonstration. In *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019*, pages 98–102. IEEE, 2019. 2.4.1

[121] M. A. Jeusfeld. DeepTelos for ConceptBase: A Contribution to the MULTI Process Challenge. In *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019*, pages 66–77. IEEE, 2019. 2.4.1

[122] M. A. Jeusfeld, J. P. A. Almeida, V. A. Carvalho, C. M. Fonseca, and B. Neumayr. Deductive reconstruction of MLT* for multi-level modeling. In E. Guerra and L. Iovino, editors, *MODELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18-23 October, 2020, Companion Proceedings*, pages 83:1–83:10. ACM, 2020. 1.3, 2.4.1

[123] M. A. Jeusfeld and B. Neumayr. DeepTelos: Multi-level Modeling with Most General Instances. In I. Comyn-Wattiau, K. Tanaka, I. Song, S. Yamamoto, and M. Saeki, editors, *Conceptual Modeling - 35th International Conference, ER 2016, Gifu, Japan, November 14-17, 2016, Proceedings*, volume 9974 of *Lecture Notes in Computer Science*, pages 198–211, 2016. 2.3.1, 2.4.1

[124] J. Jézéquel, B. Combemale, O. Barais, M. Monperrus, and F. Fouquet. Mashup of metalanguages and its implementation in the Kermeta language workbench. *Software and Systems Modeling*, 14(2):905–920, 2015. 1.4, 4.2.2

[125] S. Jurack and G. Taentzer. Towards Composite Model Transformations Using Distributed Graph Transformation Concepts. In *Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009*, pages 226–240, 2009. 3.3.1

[126] H. Kastenberg and A. Rensink. Model Checking Dynamic States in GROOVE. In A. Valmari, editor, *Model Checking Software, 13th International SPIN Workshop, Vienna, Austria, March 30 - April 1, 2006, Proceedings*, volume 3925 of *Lecture Notes in Computer Science*, pages 299–305. Springer, 2006. 1.5, 4.4, 6.1.3

[127] S. Kelly and J. Tolvanen. *Domain-Specific Modeling - Enabling Full Code Generation*. Wiley, 2008. 1.2

[128] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings*, pages 220–242, 1997. 3.1

[129] J. Kienzle, G. Mussbacher, B. Combemale, and J. Deantoni. A unifying framework for homogeneous model composition. *Software & Systems Modeling*, 18(5):3005–3023, 2019. 1.4, 3.1, 3.2.1, 3.3.2

[130] P. Knirsch and S. Kuske. Distributed Graph Transformation Units. In *Graph Transformation, First International Conference, ICGT 2002, Barcelona, Spain, October 7-12, 2002, Proceedings*, pages 207–222, 2002. 3.3.1

[131] D. S. Kolovos, R. F. Paige, and F. Polack. The Epsilon Object Language (EOL). In A. Rensink and J. Warmer, editors, *Model Driven Architecture - Foundations and Applications, 2nd European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006, Proceedings*, volume 4066 of *Lecture Notes in Computer Science*, pages 128–142. Springer, 2006. 2.4.2, 6.1.3

[132] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages. In J. Abrial and U. Glässer, editors, *Rigorous Methods for Software Construction and Analysis*, volume 5115 of *Lecture Notes in Computer Science*, pages 204–218. Springer, 2009. 6.1.3

[133] H. Krahn, B. Rumpe, and S. Völkel. MontiCore: Modular Development of Textual Domain Specific Languages. In *Objects, Components, Models and Patterns, 46th International Conference, TOOLS EUROPE 2008, Zurich, Switzerland, June 30 - July 4, 2008. Proceedings*, pages 297–315, 2008. 3.2.3, 4.2.1, 6.1.2

[134] H. Krahn, B. Rumpe, and S. Völkel. Monticore: a framework for compositional development of domain specific languages. *Int. J. Softw. Tools Technol. Transf.*, 12(5):353–372, 2010. 3.2.3, 4.2.1, 6.1.2

[135] M. E. Kramer, J. Klein, J. R. H. Steel, B. Morin, J. Kienzle, O. Barais, and J. Jézéquel. Achieving Practical Genericity in Model Weaving through Extensibility. In *Theory and Practice of Model Transformations - 6th International Conference, ICMT 2013, Budapest, Hungary, June 18-19, 2013. Proceedings*, pages 108–124, 2013. 3.2.2, 6.1.2

[136] L. M. Kristensen and S. Christensen. Implementing Coloured Petri Nets Using a Functional Programming Language. *High. Order Symb. Comput.*, 17(3):207–243, 2004. 1.6, 5.4

[137] T. Kühne. Exploring Potency. In A. Wasowski, R. F. Paige, and Ø. Haugen, editors, *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018, Copenhagen, Denmark, October 14-19, 2018*, pages 2–12. ACM, 2018. 2.3.2

[138] T. Kühne. A story of levels. In *Proceedings of MULTI @ MODELS*, pages 673–682, 2018. 2.3.1, 2.5.1

[139] T. Kühne and A. Lange. Meaningful metrics for multi-level modelling. In E. Guerra and L. Iovino, editors, *MODELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18-23 October, 2020, Companion Proceedings*, pages 85:1–85:9. ACM, 2020. 2.4.3

[140] T. Kühne and D. Schreiber. Can programming be liberated from the two-level style: multi-level programming with deepjava. In R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., editors, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 229–244. ACM, 2007. 1.3, 2.3.1, 2.4.1

[141] Y. Lamo, X. Wang, F. Mantz, Ø. Bech, A. Sandven, and A. Rutle. DPF Workbench: a multi-level language workbench for MDE. In *Proceedings of the Estonian Academy of Sciences 62(1):3-15*, 2013. 1.3, 2.3.1, 2.4.2, 3.3.1, 6.1.1

[142] A. Lange. dACL: the deep constraint and action language for static and dynamic semantic definition in Melanee. Master's thesis, University of Mannheim, 2016. 2.4.2

[143] A. Lange and C. Atkinson. Multi-level modeling with MELANEE. In R. Hebig and T. Berger, editors, *Proceedings of MODELS 2018 Workshops, Copenhagen, Denmark, October, 14, 2018*, volume 2245 of *CEUR Workshop Proceedings*, pages 653–662. CEUR-WS.org, 2018. 2.4.2

[144] A. Lange and C. Atkinson. On the Rules for Inheritance in LML. In *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019*, pages 113–118. IEEE, 2019. 2.4.2

[145] J. D. Lara, E. Guerra, and J. S. Cuadrado. When and how to use multilevel modelling. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2):12, 2014. 1.2, 1.3, 2.2

[146] M. E. V. Larsen, J. DeAntoni, B. Combemale, and F. Mallet. A Behavioral Coordination Operator Language (BCOoL). In T. Lethbridge, J. Cabot, and A. Egyed, editors, *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015, Ottawa, ON, Canada, September 30 - October 2, 2015*, pages 186–195. IEEE Computer Society, 2015. 3.3.2

[147] P. Lincoln, N. Martí-Oliet, and J. Meseguer. Specification, Transformation, and Programming of Concurrent Systems in Rewriting Logic. In *Specification of Parallel Algorithms, Proceedings of a DIMACS Workshop, Princeton, New Jersey, USA, May 9-11, 1994*, pages 309–339, 1994. 4.4

[148] J. Lindqvist, T. Lundkvist, and I. Porres. A Query Language With the Star Operator. *ECEASST*, 6, 2007. 2.5.2, 6.1.1

[149] F. Macías. *Multilevel modelling and domain-specific languages*. PhD thesis, Western Norway University of Applied Sciences and University of Oslo, 2019. 1.10, 2.3.1, 2.3.2, 2.4.3, 2.5, 2.5.1, 2.5.2, 6.2, 6.3.2

[150] F. Macías, E. Guerra, and J. de Lara. Towards Rearchitecting Meta-Models into Multi-level Models. In H. C. Mayr, G. Guizzardi, H. Ma, and O. Pastor, editors, *Conceptual Modeling - 36th International Conference, ER 2017, Valencia,*

Spain, November 6-9, 2017, Proceedings, volume 10650 of *Lecture Notes in Computer Science*, pages 59–68. Springer, 2017. 1.10, 2.3.2, 2.4.2

[151] F. Macías and A. Rodríguez. MultEcore webpage. https://ict.hvl.no/multecore/, August 2021. 1.10, 2.5.2

[152] F. Macías, A. Rutle, and V. Stolz. Multecore: Combining the best of fixed-level and multilevel metamodelling. In C. Atkinson, G. Grossmann, and T. Clark, editors, *Proceedings of the 3rd International Workshop on Multi-Level Modelling co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2016), Saint-Malo, France, October 4, 2016*, volume 1722 of *CEUR Workshop Proceedings*, pages 66–75. CEUR-WS.org, 2016. 2.5, 6.3.2

[153] F. Macías, A. Rutle, and V. Stolz. A tool for the convergence of multilevel modelling approaches. In R. Hebig and T. Berger, editors, *Proceedings of MODELS 2018 Workshops, Copenhagen, Denmark, October, 14, 2018*, volume 2245 of *CEUR Workshop Proceedings*, pages 633–642. CEUR-WS.org, 2018. 2.4.2

[154] F. Macías, A. Rutle, V. Stolz, R. Rodríguez-Echeverría, and U. Wolter. An Approach to Flexible Multilevel Modelling. *Enterp. Model. Inf. Syst. Archit. Int. J. Concept. Model.*, 13:10:1–10:35, 2018. 2.3.2, 6.3.2

[155] F. Macías, U. Wolter, A. Rutle, F. Durán, and R. Rodriguez-Echeverria. Multilevel Coupled Model Transformations for Precise and Reusable Definition of Model Behaviour. *Journal of Logical and Algebraic Methods in Programming*, 106:167–195, 2019. 1.3, 1.10, 2.3.2, 2.5.1, 4.5, 6.3.2

[156] Z. Manna and H. Sipma. Deductive Verification of Hybrid Systems Using STeP. In *Hybrid Systems: Computation and Control, First International Workshop, HSCC'98, Berkeley, California, USA, April 13-15, 1998, Proceedings*, pages 305–318, 1998. 4.4

[157] J. Y. Marchand, B. Combemale, and B. Baudry. A categorical model of model merging and weaving. In *Proceedings of the 4th International Workshop on Modeling in Software Engineering, MiSE 2012, Zurich, Switzerland, June 2-3, 2012*, pages 70–76, 2012. 1.4, 3.2.2

[158] R. C. Martin, D. Riehle, and F. Buschmann, editors. *Pattern Languages of Program Design 3*. Addison-Wesley Longman Publishing Co., Inc., USA, 1997. 2.2

[159] I. Melo, M. E. Sánchez, and J. Villalobos. Composing graphical languages. In *Proceedings of the First Workshop on the Globalization of Domain Specific Languages, GlobalDSL@ECOOP 2013, Montpellier, France, July 1, 2013*, pages 12–17, 2013. 3.2.1, 6.1.2

[160] D. Méndez-Acuña, J. A. Galindo, T. Degueule, B. Combemale, and B. Baudry. Leveraging Software Product Lines Engineering in the development of external DSLs: A systematic literature review. *Computer Languages, Systems & Structures*, 46:206–235, 2016. 1.4, 2.5.2, 3.1, 3.2

[161] T. Mens and P. V. Gorp. A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.*, 152:125–142, 2006. 1.5, 2.4.2, 2.5.2, 4.1, 4.2.2

[162] M. Mernik. An object-oriented approach to language compositions for software language engineering. *J. Syst. Softw.*, 86(9):2451–2464, 2013. 3.2.3, 6.1.2

[163] M. Mernik, V. Zumer, M. Lenic, and E. Avdicausevic. Implementation of multiple attribute grammar inheritance in the tool LISA. *ACM SIGPLAN Notices*, 34(6):68–75, 1999. 3.2.3, 6.1.2

[164] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theor. Comput. Sci.*, 96(1):73–155, 1992. 4.5

[165] J. Meseguer. Twenty years of rewriting logic. *J. Log. Algebr. Program.*, 81(7-8):721–781, 2012. 4.5

[166] Meta Object Facility (MOF) specification 2.5.1. https://www.omg.org/spec/MOF. 1.2, 2.1

[167] H. Mili, F. Pachet, et al. Patterns for metamodeling, 1998. 2.2

[168] P. Mohagheghi, W. Gilani, A. Stefanescu, M. A. Fernández, B. Nordmoen, and M. Fritzsche. Where does model-driven engineering help? Experiences from three industrial cases. *Software & Systems Modeling*, 12(3):619–639, 2013. 1.2, 5.1

[169] P. Muller, F. Fleurey, and J. Jézéquel. Weaving Executability into Object-Oriented Meta-languages. In L. C. Briand and C. Williams, editors, *Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005, Proceedings*, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278. Springer, 2005. 4.2.2

[170] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989. 4.4, 5.1

[171] G. Mussbacher, O. Alam, M. Alhaj, S. Ali, N. Amálio, B. Barn, R. Bræk, T. Clark, B. Combemale, L. M. Cysneiros, U. Fatima, R. France, G. Georg, J. Horkoff, J. Kienzle, J. C. Leite, T. C. Lethbridge, M. Luckey, A. Moreira, F. Mutz, A. P. A. Oliveira, D. C. Petriu, M. Schöttle, L. Troup, and V. M. B. Werneck. Assessing Composition in Modeling Approaches. In *Proceedings of the CMA 2012 Workshop*, CMA '12, New York, NY, USA, 2012. Association for Computing Machinery. 1.4, 3.1

[172] G. Mussbacher, D. Amyot, and J. Whittle. Composing goal and scenario models with the aspect-oriented user requirements notation based on syntax and semantics. In *Aspect-Oriented Requirements Engineering*, pages 77–99. Springer International Publishing, 2013. 3.1, 3.3.2

[173] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: Representing Knowledge About Information Systems. *ACM Trans. Inf. Syst.*, 8(4):325–362, 1990. 2.4.1

[174] B. Neumayr, M. A. Jeusfeld, M. Schrefl, and C. G. Schütz. Dual Deep Instantiation and Its ConceptBase Implementation. In *Advanced Information Systems Engineering - 26th International Conference, CAiSE 2014, Thessaloniki, Greece, June 16-20, 2014. Proceedings*, volume 8484 of *Lecture Notes in Computer Science*, pages 503–517. Springer, 2014. 1.3, 2.4.1

[175] B. Neumayr, C. G. Schuetz, C. Horner, and M. Schrefl. Deepruby: Extending ruby with dual deep instantiation. In *Proceedings of MODELS 2017 Workshops, Austin, TX, USA, September, 17, 2017*, volume 2019 of *CEUR Workshop Proceedings*, pages 252–260. CEUR-WS.org, 2017. 2.3.1, 2.4.1

[176] B. Neumayr, C. G. Schuetz, M. A. Jeusfeld, and M. Schrefl. Dual deep modeling: multi-level modeling with dual potencies and its formalization in f-logic. *Softw. Syst. Model.*, 17(1):233–268, 2018. 2.3.1, 2.3.2, 2.4.1, 6.1.1

[177] S. Nidhra and J. Dondeti. Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, 2(2):29–50, 2012. 4.4

[178] J. Odell. Power types. *J. Object Oriented Program.*, 7(2):8–12, 1994. 2.2, 2.3.1, 2.3.2, 2.4.1

[179] A. Oyegoke. The constructive research approach in project management research. *International Journal of Managing Projects in Business*, 4(4):573–595, 2011. 1.8

[180] R. F. Paige, D. S. Kolovos, and F. Polack. An action semantics for MOF 2.0. In H. Haddad, editor, *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC), Dijon, France, April 23-27, 2006*, pages 1304–1305. ACM, 2006. 4.2.2

[181] G. A. Papadopoulos and F. Arbab. Coordination Models and Languages. *Adv. Comput.*, 46:329–400, 1998. 3.3.2

[182] M. Papathomas. A Unifying Framework for Process Calculus Semantics of Concurrent Object-Oriented Languages. In *Object-Based Concurrent Computing, ECOOP'91 Workshop, Geneva, Switzerland, July 15-16, 1991, Proceedings*, pages 53–79, 1991. 4.4

[183] J. Parrow. An introduction to the π-calculus. In *Handbook of Process Algebra*, pages 479–543. 2001. 4.4

[184] P. Perrotta. *Metaprogramming Ruby 2: Program Like the Ruby Pros*. Facets of Ruby series. Pragmatic Bookshelf, 2014. 2.4.1

[185] F. Ramalho, J. Robin, and R. S. M. de Barros. XOCL - an XML Language for Specifying Logical Constraints in Object Oriented Models. *J. Univers. Comput. Sci.*, 9(8):956–969, 2003. 2.4.1

[186] U. Ranger and M. Lüstraeten. Search Trees for Distributed Graph Transformation Systems. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, 4, 2006. 3.3.1

[187] W. Reisig. *Petri nets: an introduction*, volume 4. Springer Science & Business Media, 2012. 1.6

[188] W. Reisig. *Elements of distributed algorithms: modeling and analysis with Petri nets*. Springer Science & Business Media, 2013. 1.6, 5.1

[189] W. Reisig. *Understanding Petri Nets - Modeling Techniques, Analysis Methods, Case Studies*. Springer, 2013. 5.1

[190] A. Rensink. The GROOVE simulator: A tool for state space generation. In *International Workshop on Applications of Graph Transformations with Industrial Relevance*, pages 479–485. Springer, 2003. 3.3.1, 4.4, 6.1.3, 6.4.2.2

[191] A. Rensink and J. Kuperus. Repotting the Geraniums: On Nested Graph Transformation Rules. *ECEASST*, 18, 2009. 3.3.1

[192] A. Rodríguez. MultEcore-Maude webpage. `https://ict.hvl.no/multecore-maude/`, August 2021. 1.10, 2.5.2

[193] A. Rodríguez, F. Durán, A. Rutle, and L. M. Kristensen. Executing Multilevel Domain-Specific Models in Maude. *Journal of Object Technology*, 18(2):4:1–21, 2019. 1.9, 1.9.3, 1.10, 1.11, 6.3.2

[194] A. Rodriguez, F. Durán, and L. M. Kristensen. Execution and Analysis of MultEcore Multilevel Modelling Languages using Maude. Manuscript submitted for publication to the *International Journal on Software and Systems Modeling*. Submitted version available at: `https://bit.ly/3ug4iWZ`, 2020. 1.9, 1.9.5, 1.11

[195] A. Rodríguez, L. M. Kristensen, and A. Rutle. On Modelling and Validation of the MQTT IoT Protocol for M2M Communication. In D. Moldt, E. Kindler, and H. Rölke, editors, *Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE'18), Bratislava, Slovakia, June 24-29, 2018*, volume 2138 of *CEUR Workshop Proceedings*, pages 99–118. CEUR-WS.org, 2018. 1.12

[196] A. Rodríguez, L. M. Kristensen, and A. Rutle. Formal Modelling and Incremental Verification of the MQTT IoT Protocol. *Trans. Petri Nets Other Model. Concurr.*, 14:126–145, 2019. 1.9, 1.9.1, 1.11

[197] A. Rodríguez, L. M. Kristensen, and A. Rutle. On CTL Model Checking of the MQTT IoT Protocol using the Sweep-Line Method. In D. Moldt, E. Kindler, and M. Wimmer, editors, *Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE'19), Aachen, Germany, June 23-28, 2019*, volume 2424 of *CEUR Workshop Proceedings*, pages 57–72. CEUR-WS.org, 2019. 1.12

[198] A. Rodríguez, L. M. Kristensen, and A. Rutle. Verification of the MQTT IoT Protocol Using Property-Specific CTL Sweep-Line Algorithms. *Trans. Petri Nets Other Model. Concurr.*, 15:165–183, 2021. 1.9, 1.9.2, 1.11, 4.4

[199] A. Rodríguez and F. Macías. Multilevel Modelling with MultEcore: A Contribution to the MULTI Process Challenge. In *Proceedings of MULTI @ MODELS*, pages 152–163, 2019. 1.10, 1.12, 2.2

[200] A. Rodriguez and F. Macías. Multilevel Modelling with MultEcore: A contribution to the Multi-Level Process Challenge. Manuscript submitted for publication to the *Enterprise Modelling and Information Systems Architectures Journal*. Submitted version available at: `https://bit.ly/3tjzgw9`, 2021. 1.9, 1.9.6, 1.11

[201] A. Rodriguez, F. Macías, F. Durán, A. Rutle, and U. Wolter. Composition of Multilevel Domain-Specific Modelling Languages. Manuscript submitted for publication to the *Journal of Logical and Algebraic Methods in Programming*. Submitted version available at: `https://bit.ly/2SncxCp`, 2020. 1.9, 1.9.4, 1.11, 2.5.1

[202] A. Rodríguez, A. Rutle, F. Durán, L. M. Kristensen, and F. Macías. Multilevel modelling of coloured petri nets. In R. Hebig and T. Berger, editors, *Proceedings of MODELS 2018 Workshops, Copenhagen, Denmark, October, 14, 2018*, volume 2245 of *CEUR Workshop Proceedings*, pages 663–672. CEUR-WS.org, 2018. 1.12, 2.5.1, 6.3.2

[203] A. Rodriguez, A. Rutle, F. Durán, L. Kristensen, F. Macías, and U. Wolter. Composition of multilevel modelling hierarchies. In *The Nordic Workshop on Programming Theory*, 01 2020. 1.12

[204] A. Rodríguez, A. Rutle, L. M. Kristensen, and F. Durán. A Foundation for the Composition of Multilevel Domain-Specific Languages. In *MULTI@ MoDELS*, pages 88–97, 2019. 1.10, 1.12, 2.5.1, 3.4.1, 5.1, 6.2, 6.3.2

[205] M. Roldán and F. Durán. Dynamic validation of OCL constraints with mOdCL. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, 44, 2011. 4.5

[206] A. Rossini, J. de Lara, E. Guerra, A. Rutle, and U. Wolter. A formalisation of deep metamodelling. *Formal Aspects Comput.*, 26(6):1115–1152, 2014. 2.3.2, 6.1.1

[207] R. Rubio, N. Martí-Oliet, I. Pita, and A. Verdejo. Parameterized Strategies Specification in Maude. In J. L. Fiadeiro and I. Tutu, editors, *Recent Trends in Algebraic Development Techniques - 24th IFIP WG 1.3 International Workshop, WADT 2018, Egham, UK, July 2-5, 2018, Revised Selected Papers*, volume 11563 of *Lecture Notes in Computer Science*, pages 27–44. Springer, 2018. 6.4.2.2

[208] R. Rubio, N. Martí-Oliet, I. Pita, and A. Verdejo. Strategies, Model Checking and Branching-Time Properties in Maude. In S. Escobar and N. Martí-Oliet, editors, *Rewriting Logic and Its Applications - 13th International Workshop, WRLA 2020, Virtual Event, October 20-22, 2020, Revised Selected Papers*, volume 12328 of *Lecture Notes in Computer Science*, pages 156–175. Springer, 2020. 6.4.2.2

[209] A. Rutle. *Diagram Predicate Framework: A formal approach to MDE*. PhD thesis, University of Bergen, 2010. 2.4.2

[210] G. Salaün, L. Bordeaux, and M. Schaerf. Describing and reasoning on Web Services using Process Algebra. *Int. J. Bus. Process. Integr. Manag.*, 1(2):116–128, 2006. 4.4

[211] M. Sánchez, P. J. Clemente, J. M. Murillo, and J. H. Núñez. CoordMaude: Simplifying Formal Coordination Specifications of Cooperation Environment. *Electron. Notes Theor. Comput. Sci.*, 82(3):643–658, 2003. 3.3.2, 6.1.2

[212] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. William C. Brown Publishers, USA, 1986. 4.2

[213] D. C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006. 1.2, 3.1

[214] S. Sendall and W. Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Softw.*, 20(5):42–45, 2003. 4.1

[215] N. Shankar, S. Owre, J. M. Rushby, and D. W. Stringer-Calvert. Pvs prover guide. *Computer Science Laboratory, SRI International, Menlo Park, CA*, 1:11–12, 2001. 4.4

[216] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artif. Intell.*, 138(1-2):181–234, 2002. 2.4.1

[217] K. I. F. Simonsen, L. M. Kristensen, and E. Kindler. Pragmatics annotated coloured petri nets for protocol software generation and verification. In *Transactions on Petri Nets and Other Models of Concurrency XI*, pages 1–27. Springer, 2016. 1.6, 5.1, 6.3.1

[218] J. Snyder and U. Flemming. The Object Modeling Language (OML) Specification. *J. Object Oriented Program.*, 03 1995. 2.2

[219] T. Stahl, M. Völter, J. Bettin, A. Haase, and S. Helsen. *Model-driven software development - technology, engineering, management*. Pitman, 2006. 1.2

[220] F. Steimann. On the representation of roles in object-oriented and conceptual modelling. *Data Knowl. Eng.*, 35(1):83–106, 2000. 3.2.4

[221] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008. 1.2, 2.1, 2.5.1

[222] J. E. Stoy. *Denotational semantics: the Scott-Strachey approach to programming language theory*. MIT press, 1981. 4.2.1

[223] D. Strüber, K. Born, K. D. Gill, R. Groner, T. Kehrer, M. Ohrndorf, and M. Tichy. Henshin: A Usability-Focused Framework for EMF Model Transformation Development. In J. de Lara and D. Plump, editors, *Graph Transformation - 10th International Conference, ICGT 2017, Held as Part of STAF 2017, Marburg, Germany, July 18-19, 2017, Proceedings*, volume 10373 of *Lecture Notes in Computer Science*, pages 196–208. Springer, 2017. 1.5, 4.4, 6.1.3

[224] P. Stünkel, H. König, Y. Lamo, and A. Rutle. Towards multiple model synchronization with comprehensive systems. In *Fundamental Approaches to Software Engineering - 23rd International Conference, FASE 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Proceedings*, volume Accepted for publication of *Lecture Notes in Computer Science*. Springer, 2020. 3.2.2, 3.4.1

[225] E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. Van Mierlo, and H. Ergin. AToMPM: A web-based modeling environment. In *MODELS-JP 2013*, volume 1115 of *CEUR Workshop Proceedings*, pages 21–25, 2013. 1.3, 2.4.2

[226] G. Taentzer. *Parallel and distributed graph transformation - formal description and application to communication-based systems*. Berichte aus der Informatik. Shaker, 1996. 3.3.1

[227] A. R. Tena, F. Macıas, L. M. Kristensen, and A. Rutle. Towards domain-specific cpn modelling languages. *Marina Waldén (Editor)*, page 62, 2017. 1.12

[228] Z. Theisz, S. Bácsi, G. Mezei, F. A. Somogyi, and D. Palatinszky. By Multi-layer to Multi-level Modeling. In *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019*, pages 134–141. IEEE, 2019. 1.3, 2.3.1, 2.4.2, 6.1.1

[229] Z. Theisz, S. Bácsi, G. Mezei, F. A. Somogyi, and D. Palatinszky. Join potency: a way of combining separate multi-level models. In E. Guerra and L. Iovino, editors, *MODELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18-23 October, 2020, Companion Proceedings*, pages 84:1–84:5. ACM, 2020. 2.3.2

[230] W. J. Thong and M. A. Ameedeen. A survey of petri net tools. In H. A. Sulaiman, M. A. Othman, M. F. I. Othman, Y. A. Rahim, and N. C. Pee, editors, *Advanced Computer and Communication Engineering Technology*, pages 537–551, Cham, 2015. Springer International Publishing. 5.1

[231] S. Tripakis and T. Dang. *Modeling, Verification, and Testing Using Timed and Hybrid Automata*. 11 2009. 4.4

[232] J. Troya, J. M. Bautista, F. López-Romero, and A. Vallecillo. Lightweight Testing of Communication Networks with *e-Motions*. In *Tests and Proofs - 5th International Conference, TAP@TOOLS 2011, Zurich, Switzerland, June 30 - July 1, 2011. Proceedings*, pages 187–204, 2011. 4.4

[233] D. Turi and G. D. Plotkin. Towards a Mathematical Operational Semantics. In *Proceedings, 12th Annual IEEE Symposium on Logic in Computer Science, Warsaw, Poland, June 29 - July 2, 1997*, pages 280–291, 1997. 4.2.2

[234] J. D. Ullman. *Elements of ML programming*. Prentice-Hall, Inc., 1994. 1.6, 4.5, 5.1

[235] The Unified Modelling Language (UML) specification 2.5.1. https://www.omg.org/spec/UML. 1.2

[236] D. Urbán, Z. Theisz, and G. Mezei. Self-describing Operations for Multi-level Meta-modeling. In S. Hammoudi, L. F. Pires, and B. Selic, editors, *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018, Funchal, Madeira - Portugal, January 22-24, 2018*, pages 519–527. SciTePress, 2018. 2.4.2

[237] A. Valmari. The State Explosion Problem. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996*, pages 429–528, 1996. 4.4

[238] W. Van Der Aalst, K. M. Van Hee, and K. van Hee. *Workflow management: models, methods, and systems*. MIT press, 2004. 5.1

[239] S. Van Mierlo, B. Barroca, H. Vangheluwe, E. Syriani, and T. Kühne. Multi-level modelling in the Modelverse. In *MULTI@ MoDELS*, volume 1286 of *CEUR Workshop Proceedings*, pages 83–92, 2014. 1.3, 2.4.2

[240] W. Vogler, A. L. Semenov, and A. Yakovlev. Unfolding and Finite Prefix for Nets with Read Arcs. In *CONCUR '98: Concurrency Theory, 9th International Conference, Nice, France, September 8-11, 1998, Proceedings*, pages 501–516, 1998. 5.2

[241] B. Volz and S. Jablonski. Towards an Open Meta Modeling Environment. In *Proceedings of the 10th Workshop on Domain-Specific Modeling, DSM'10*, 10 2010. 2.4.2, 6.1.1

[242] J. Whittle, J. Hutchinson, and M. Rouncefield. The state of practice in model-driven engineering. *IEEE software*, 31(3):79–85, 2014. 1.2

[243] J. Whittle, P. K. Jayaraman, A. M. Elkhodary, A. Moreira, and J. Araújo. MATA: A unified approach for composing UML aspect models based on graph transformation. *LNCS Trans. Aspect Oriented Softw. Dev.*, 6:191–237, 2009. 3.2.2

[244] G. Winskel. *The formal semantics of programming languages: an introduction*. MIT press, 1993. 4.2.3

[245] U. Wolter, F. Macías, and A. Rutle. The Category of Typing Chains as a Foundation of Multilevel Typed Model Transformations. Technical Report 2019-417, University of Bergen, Department of Informatics, November 2019. 2.5.1, 2.5.1, 2.5.2, 6.3.2

[246] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. S. Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4):19:1–19:36, 2009. 4.4

[247] A. Wortmann, O. Barais, B. Combemale, and M. Wimmer. Modeling languages in Industry 4.0: an extended systematic mapping study. *Software and Systems Modeling*, 19(1):67–94, 2020. 2.2

[248] G. Yang, M. Kifer, H. Wan, and C. Zhao. FLORA-2: User's Manual. *Flora-2 portal on Sourceforge*, December 2020. 2.4.1

[249] G. Zhang and M. M. Hölzl. Hila: High-level aspects for UML state machines. In *Models in Software Engineering, Workshops and Symposia at MODELS 2009, Denver, CO, USA, October 4-9, 2009, Reports and Revised Selected Papers*, pages 104–118, 2009. 2.5.1, 3.1

[250] M. Zhou and A. D. Robbi. *Applications of Petri net methodology to manufacturing systems*, pages 207–230. Springer Netherlands, Dordrecht, 1994. 5.1

[251] S. Zschaler, D. S. Kolovos, N. Drivalos, R. F. Paige, and A. Rashid. Domain-Specific Metamodelling Languages for Software Language Engineering. In M. van den Brand, D. Gasevic, and J. Gray, editors, *Software Language Engineering, Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers*, volume 5969 of *Lecture Notes in Computer Science*, pages 334–353. Springer, 2009. 1.4, 3.1

Part II

ARTICLES

PAPER A

FORMAL MODELLING AND INCREMENTAL VERIFICATION OF THE MQTT IOT PROTOCOL

Alejandro Rodríguez, Lars Michael Kristensen, Adrian Rutle

In Transactions on Petri Nets and Other Models of Concurrency XIV, volume 11790 of Lecture Notes in Computer Science, pages 126-145, **Springer Nature**, 2019

Formal Modelling and Incremental Verification of the MQTT IoT Protocol

Alejandro Rodríguez(✉), Lars Michael Kristensen, and Adrian Rutle

Department of Computing, Mathematics, and Physics,
Western Norway University of Applied Sciences, Bergen, Norway
{arte,lmkr,aru}@hvl.no

Abstract. Machine to Machine (M2M) communication and Internet of Things (IoT) are becoming still more pervasive with the increase of communicating devices used in cyber-physical environments. A prominent approach to communication between distributed devices in highly dynamic IoT environments is the use of publish-subscribe protocols such as the Message Queuing Telemetry Transport (MQTT) protocol. MQTT is designed to be light-weight while still being resilient to connectivity loss and component failures. We have developed a Coloured Petri Net model of the MQTT protocol logic using CPN Tools. The model covers all three quality of service levels provided by MQTT (at most once, at least once, and exactly once). For the verification of the protocol model, we show how an incremental model checking approach can be used to reduce the effect of the state explosion problem. This is done by exploiting that the MQTT protocol operates in phases comprised of connect, subscribe, publish, unsubscribe, and disconnect.

1 Introduction

Publish-subscribe messaging systems support data-centric communication and have been widely used in enterprise networks and applications. A main reason for this is that a software system architecture based on publish-subscribe messaging provides better support for scalability and adaptability than the traditional client-server architecture used in distributed systems. The interaction and exchange of messages between clients based on the publish-subscribe paradigm are based on middleware usually referred to as a *broker* (or a bus) that manages *topics*. The broker provides space decoupling [9] allowing a client acting as a publisher on a given topic to send messages to other clients acting as subscribers to the topic without the need to know the identity of the receiving clients. The broker also provides synchronisation decoupling in that clients can exchange messages without being executing at the same time. Furthermore, the processing in the broker can be parallelized and handled using event-driven techniques.

The loose coupling and support for asynchronous point-to-multipoint messaging, make the publish-subscribe paradigm attractive also in the context of Internet of Things (IoT) which has experienced significant growth in applications and adoptability in recent years [17]. The IoT paradigm blends the virtual

© Springer-Verlag GmbH Germany, part of Springer Nature 2019
M. Koutny et al. (Eds.): ToPNoC XIV, LNCS 11790, pp. 126–145, 2019.
https://doi.org/10.1007/978-3-662-60651-3_5

and the physical worlds by bringing different concepts and technical components together: pervasive networks, miniaturisation of devices, mobile communication, and new ecosystems [6]. Moreover, the implementation of a connected product typically requires the combination of multiple software and hardware components distributed in a multi-layer stack of IoT technologies.

MQTT [3] is a publish-subscribe messaging protocol for IoT designed with the aim of being light-weight and easy to implement. These characteristics make it a suitable candidate for constrained environments such as Machine-to-Machine communication (M2M) and IoT contexts where a small memory footprint is required and where network bandwidth is often a scarce resource. Even though MQTT has been designed to be easy to implement, it still contains relatively complex protocol logic for handling connections, subscriptions, and the various quality of service levels related to message delivery. Furthermore, MQTT is expected to play a key role in future IoT applications and will be implemented for a wide range of platforms and in a broad range of programming languages making interoperability a key issue. This, combined with the fact that MQTT is only backed by an (ambiguous) natural language specification, motivated us to develop a formal and executable specification of the MQTT protocol.

We have used Coloured Petri Nets (CPNs) [12] for the modelling and verification of the MQTT specification. The main reason is that CPNs have been successfully applied in earlier work to build formal specifications of communication protocols [8], data networks [5], and embedded systems [1]. To ensure the proper operations of the constructed CPN model, we have validated the CPN model using simulation and verified an elaborate set of behavioural properties of the constructed model using model checking and state space exploration. In the course of our work on the MQTT specification [3] and the development of the CPN model, we have identified a number of issues related in particular to the implementation of the quality of service levels. These issues are a potential source of interoperability problems between implementations. For the construction of the model we have applied some general modelling patterns for CPN models of publish-subscribe protocols. Compared to earlier work on modelling and verification of publish-subscribe protocols [4,10,18] (which we discuss in more details towards the end of this paper) our work specifically targets MQTT, and we consider a more extensive set of behavioural properties.

The rest of this paper is organised as follows. In Sect. 2 we present the MQTT protocol context and give a high-level overview of the constructed CPN model. Section 3 details selected parts of the CPN model of the MQTT protocol. In Sect. 4 we present our experimental results on using simulation and model checking to validate and verify central properties of MQTT and the CPN model. Finally, in Sect. 5 we sum up the conclusions, discuss related work, and outlines directions for future work. Due to space limitations, we cannot present the complete CPN model of the MQTT protocol. The constructed CPN model is available via [15]. The reader is assumed to be familiar with the basic concepts of Petri Nets and High-level Petri Nets [12].

2 MQTT Protocol and CPN Model Overview

MQTT [3] runs over the TCP/IP protocol or other transport protocols that provide ordered, lossless and bidirectional connections. MQTT applies topic-based filtering of messages with a topic being part of each published message. An MQTT client can subscribe to a topic to receive messages, publish on a topic, and clients can subscribe to as many topics as they are interested in. As described in [14], an MQTT client can operate as a publisher or subscribe and we use the term client to generally refer to a publisher or a subscriber. The MQTT broker [14] is the core of any publish/subscribe protocol and is responsible for keeping track of subscriptions, receiving and filtering messages, deciding to which clients they will be dispatched, and sending them to all subscribed clients. There are no restrictions in terms of hardware to run as an MQTT client, and any device equipped with an MQTT library and connected to an MQTT broker can operate as a client.

2.1 Modelling Roles and Messages

Figure 1 shows the top-level module of the CPN MQTT model which consists of two *substitution transitions* (drawn as rectangles with double-lined borders) representing the Clients and the Broker roles of MQTT. Substitution transitions constitute the basic syntactical structuring mechanism of CPNs and each of the substitution transitions has an associated *module* that models the detailed behaviour of the clients and the broker, respectively. The name of the (sub)module associated with a substitution transition is written in the rectangular tag positioned next to the transition.

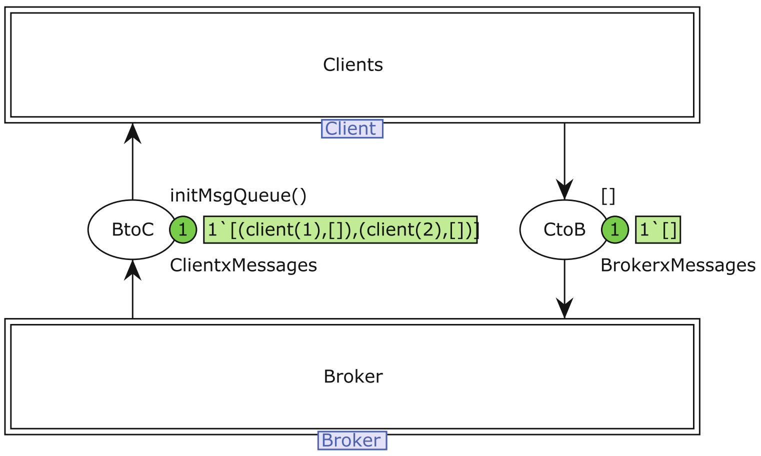


Fig. 1. The top-level module of the MQTT CPN model.

The complete CPN model of the MQTT protocol consists of 24 modules organised into six hierarchical levels. We have constructed a parametric CPN model which makes it easy to change the number of clients and topics without making changes to the net-structure. This makes it possible to investigate different configuration of MQTT and it is a main benefit provided by CPNs in comparison to ordinary Petri Nets.

The two substitution transitions in Fig. 1 are connected via directed arcs to the two places CtoB and BtoC. The clients and the broker interact by producing and consuming tokens on the places. Figure 2 shows the central data type definitions used for the colour sets of the places CtoB and BtoC and the modelling of clients and messages. The colour sets `QoS` is used for modelling the three quality of service levels supported by MQTT (see below), and the colour set `PID` is used for modelling the packet identifiers which plays a central role in the MQTT protocol logic. We have abstracted from the actual payload of the published messages. The reason for this is that the message transport structure and the protocol logic of MQTT is agnostic to the payload contained, i.e., the actual content that will be sent in the messages. For similar reasons, we also abstract from the hierarchical structuring of topics.

```
val T = 5;  (* number of topics  *)
val C = 2;  (* number of clients *)

colset Client = index client with 1..C;
colset Topic  = index topic with 1..T;
colset QoS    = index QoS with 0..2; (* quality of service *)
colset PID    = INT;                 (* packet identifiers *)

colset TopicxPID     = product Topic * PID;
colset TopicxQoSxPID = product Topic * QoS * PID;

colset Message = union CONNECT + CONNACK +
  SUBSCRIBE : TopicxQoSxPID + UNSUBSCRIBE : TopicxPID +
  SUBACK    : TopicxQoSxPID + UNSUBACK    : TopicxPID
  PUBLISH   : TopicxQoSxPID +
  PUBACK    : TopicxPID     + PUBREC      : TopicxPID +
  PUBREL    : TopicxPID     + PUBCOMP     : TopicxPID +
  DISCONNECT;

colset Messages = list Message;

colset ClientxMessage      = product Client * Message;
colset BrokerxMessages     = list ClientxMessage;

colset ClientxMessageQueue = product Client * Messages;
colset ClientxMessages     = list ClientxMessageQueue;
```

Fig. 2. Client and message colour set definitions (Color figure online)

The places CtoB and BtoC are designed to behave as queues. The queue mechanism offers some advantages that the MQTT specification implicitly indicates. The purpose of this is to ensure the ordered message distribution as assumed from the transport service on top of which MQTT operates. Even so, the CtoB and BtoC places are slightly different; while CtoB is modelled as a single queue that the broker manages to consume messages, BtoC is designed to maintain an incoming queue of messages for each client. This construction assures that all clients will have their own queue, individually respecting the ordered reception of messages. The function `initMsgQueue()` initialises the queues according to the number of clients specified by the symbolic constant `C`. The `BrokerxMessages` colour set for the CtoB place used at the bottom of Fig. 2 consists of a list of `ClientxMessage` which are pairs of `Client` and `Messages`.

We represent all the messages that the clients and the broker can use by means of the `Message` colour set. We use the terms packet and message indistinguishably when we refer to control packets. The control information used depends on the messages considered. As an example, a `Connect` message (packet) does not contain control information, but a `Publish` message requires a specific `Topic`, `QoS`, and `PID`. The `Topic` and `QoS` colour sets are both indexed types containing values (`topic(1)`, `topic(2)` ... `topic(T)` depending on the constant `T`, and `QoS(0)`, `QoS(1)` and `QoS(2)`, respectively. The `ClientxMessages` colour set for the BtoC place encapsulates all the queues (each one declared as a pair of `Client` and `Messages` in the `ClientxMessageQueue` colour set) in one single queue. This modelling pattern allows us to deal with the distribution of multiple messages in a single step in the broker side which in turn simplifies the modelling of the broker and reduces the number of reachable states of the model.

2.2 Quality of Service

The MQTT protocol delivers application messages according to the three Quality of Service (QoS) levels defined in [3]. The QoS levels are motivated by the different needs that IoT applications may have in terms of reliable delivery of messages. It should be noted that even if MQTT has been designed to operate over a transport service with lossless and ordered delivery, then message reliability still must be addressed as logical transport connections may be lost.

The delivery protocol is symmetric, and the clients and the broker can each take the role of either a sender or a receiver. The delivery protocol is concerned solely with the delivery of an application message from a single sender to a single receiver. When the broker is delivering an application message to more than one client, each client is treated independently. The QoS level used to deliver an outbound message from the broker could differ from the QoS level designated in the inbound message. Therefore, we need to distinguish two different parts of delivering a message: a client that publishes to the broker and the broker that forwards the message to the subscribing clients. The three MQTT QoS levels for message delivery are:

At most once: (QoS level 0): The message is delivered according to the capabilities of the underlying network. No response is sent by the receiver and no retry is performed by the sender. The message arrives at the receiver either once or not at all. An application of this QoS level is in environments where sensors are constantly sending data and it does not matter if an individual reading is lost as the next one will be published soon after.

At least once (QoS level 1): Where messages are assured to arrive, but duplicates can occur. It fits adequately for situations where delivery assurance is required but duplication will not cause inconsistencies. An application of this are idempotent operations on actuators, such as closing a valve or turning on a motor.

Exactly once (QoS level 2): Where messages are assured to arrive exactly once. This is for use when neither loss nor duplication of messages are acceptable. This level could be used, for example, with billing systems where duplicate or lost messages could lead to incorrect charges being applied.

When a client subscribes to a specific topic with a certain QoS level it means that the client is determining the maximum QoS that can be expected for that topic. When the broker transfers the message to a subscribing client it uses the QoS of the subscription made by the client. Hence QoS guarantees can get downgraded for a particular receiving client if subscribed with a lower QoS. This means that if a receiver is subscribed to a topic with a QoS level 0, no matter if a sender publishes in this topic with a QoS level 2, then the receiver will proceed with its QoS level 0.

3 Modelling the Protocol Roles and Their Interaction

We now consider the different phases and the client-broker interaction in the MQTT protocol, and show how we have modelled the MQTT protocol logic using CPNs. MQTT defines five main operations: connect, subscribe, publish, unsubscribe, and disconnect. Such operations, except the connect which must be the first one for the clients, are independent of each other and can be triggered in parallel by either the clients or the broker. The model is organized following a modelling pattern that ensures modularity and therefore, encapsulation of the protocol logic and behaviour of such operations. This offers advantages both for readability and understandability of the model and also, for making easier to detect and fix errors during the incremental verification.

3.1 Interaction Overview

In order to show how the clients and the broker interact, we describe the different actions that clients may carry out by considering an example. Figure 3 shows a sequence diagram for a scenario where two clients connect, perform subscribe, publish and unsubscribe, and finally disconnect from the broker. The numbers on each step of the communication define the interaction of the protocol as follows:

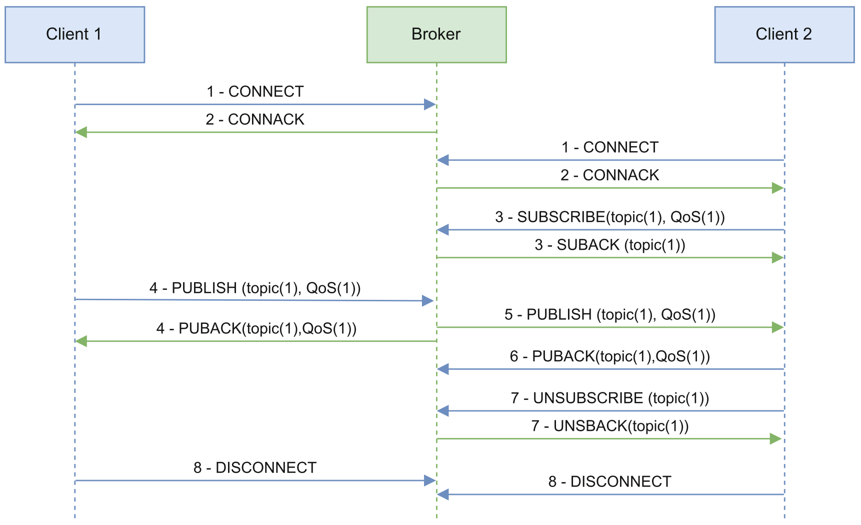

Client 1
Broker
Client 2
1 - CONNECT
2 - CONNACK
1 - CONNECT
2 - CONNACK
3 - SUBSCRIBE(topic(1), QoS(1))
3 - SUBACK (topic(1))
4 - PUBLISH (topic(1), QoS(1))
5 - PUBLISH (topic(1), QoS(1))
4 - PUBACK(topic(1),QoS(1))
6 - PUBACK(topic(1),QoS(1))
7 - UNSUBSCRIBE (topic(1))
7 - UNSBACK(topic(1))
8 - DISCONNECT
8 - DISCONNECT

Fig. 3. Message sequence diagram illustrating the MQTT phases.

1. Client 1 and Client 2 request a connection to the Broker.
2. The Broker sends back a connection acknowledgement to confirm the establishment of the connection.
3. Client 2 subscribes to topic 1 with a QoS level 1, and the Broker confirms the subscription with a subscribe acknowledgement message.
4. Client 1 publishes on topic 1 with a QoS level 1. The Broker responds with a corresponding publish acknowledgement.
5. The Broker transmits the publish message to Client 2 which is subscribed to the topic.
6. Client 2 gets the published message, and sends a publish acknowledgement back as a confirmation to the Broker that it has received the message.
7. Client 2 unsubscribes to topic 1, and the Broker responds with an unsubscribe acknowledgement.
8. Client 1 and Client 2 disconnect.

3.2 Client and Broker State Modelling

The colour sets defined for modelling the client state are shown in Fig. 4. The place Clients (top-left place in Fig. 5) uses a token for each client to store their respective state during the communication. This is a modelling pattern that allows not only to parameterize the model so we can change the number of clients without modifying the structure, but also to maintain all the clients independently in only one place and with a proper data structure that encapsulates all the information required. The states of the clients are represented by the `ClientxState` colour set which is a product of `Client` and `ClientState`. The record colour set `ClientState` is used to represent the state of a client which consists of a list of `TopicxQoS`, a `State`, and a `PID`. Using this, a client stores the topics it is subscribed to, and the quality of service level of each subscription.

```
colset State = with READY | DISC | CON | WAIT;

colset TopicxQoS     = product Topic * QoS;
colset ListTopicxQoS = list TopicxQoS;

colset ClientState   = record topics : ListTopicxQoS *
                              state  : State *
                              pid    : PID;

colset ClientxState = product Client * ClientState;
```

Fig. 4. Colour set definitions used for modelling client state. (Color figure online)

The `State` colour set is an enumeration type containing the values `READY` (for the initial state), `WAIT` (when the client is waiting to be connected), `CON` (when the client is connected), and `DISC` (for when the client has disconnected).

Below we present selected parts of the model by first presenting a high-level view of the clients and broker sides, and then illustrating how the model captures the execution scenario described in Sect. 3.1 where two clients connects, one subscribes to a topic, and the other client publishes on this topic. The unsubscribe and the disconnection phases are not detailed due to space limitations.

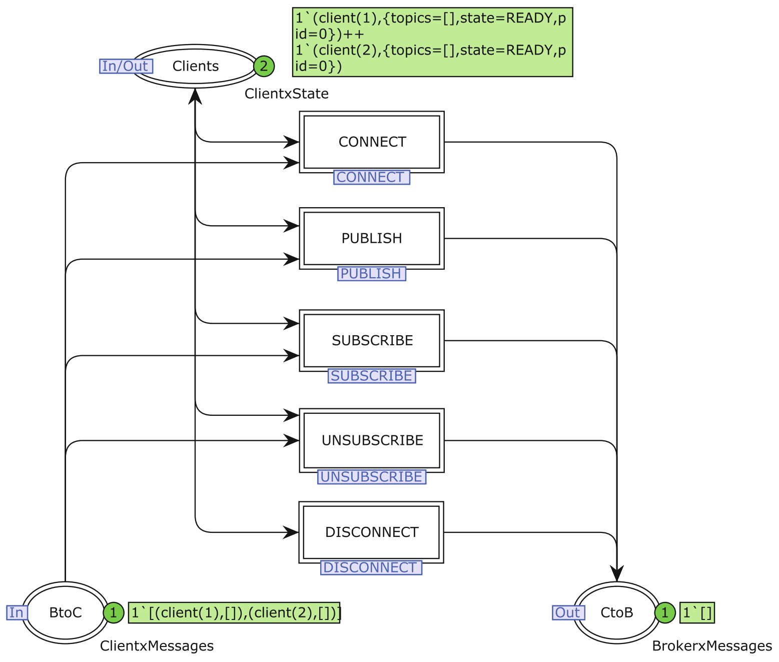


Fig. 5. ClientProcessing submodule.

3.3 Client Modelling

The ClientProcessing submodule in Fig. 5 models all the operations that a client can carry out. Clients can behave as senders and receivers, and the five substitution transitions `CONNECT`, `PUBLISH`, `SUBSCRIBE`, `UNSUBSCRIBE` and `DISCONNECT` has been constructed to capture both behaviours.

The socket place Clients stores the information of all the clients that are created at the beginning of the execution of the model. In this scenario there are two clients, and the value of the tokens representing the state of the two clients is provided in the green rectangle (the marking of the place) next to the Clients place. The BtoC and CtoB port places are associated with the socket places already shown in Fig. 2.

3.4 Broker Modelling

We have modelled the broker similarly as we have done for clients. This can be seen from Fig. 6 which shows the BrokerProcessing submodule. The Connected-Clients place keeps the information of all clients as perceived by the broker. This place is designed as a central storage, and it is used by the broker to distribute the messages over the network. The broker behaviour is different from that of the clients, since it will have to manage all the requests and generate responses for several clients at the same time.

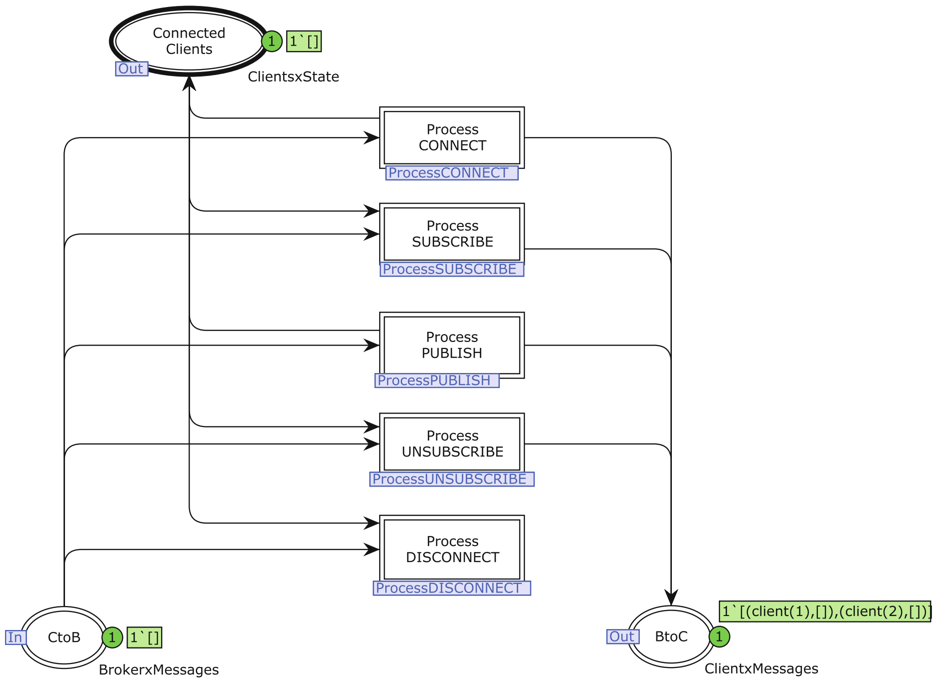


Fig. 6. The BrokerProcessing module.

3.5 Connection Phase

The first step for a client to be part of the message exchange is to connect to the broker. A client will send a CONNECT request, and the broker will respond with a CONNACK message to complete the connection establishment. Figure 7 shows the CONNECT submodule in a marking where client(1) has sent a CONNECT request and it is waiting (state = WAIT) for the broker acknowledgement processing to finish such that the connection state can be set to CON.

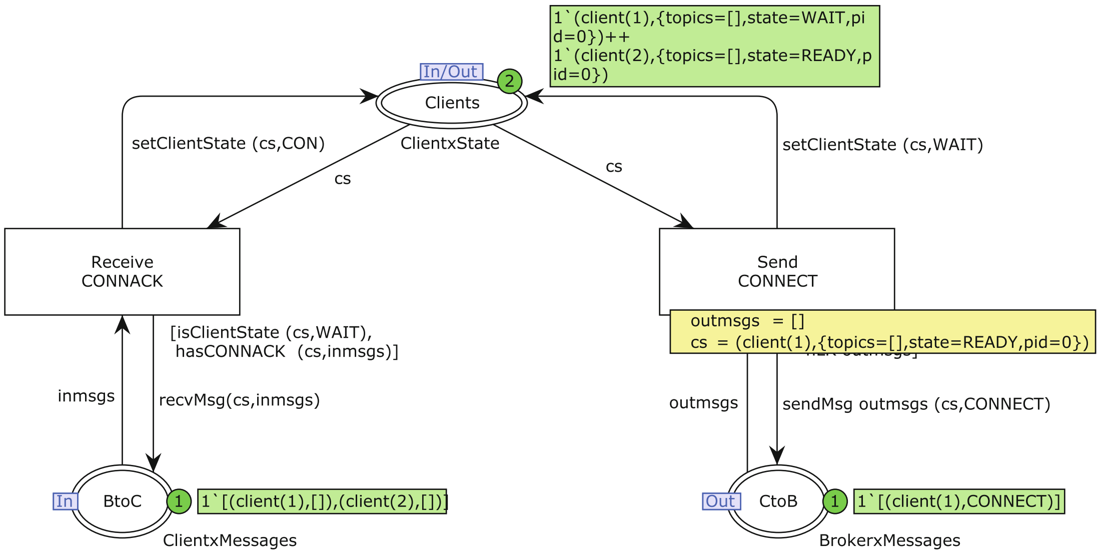

1`(client(1),{topics=[],state=WAIT,pid=0})++
1`(client(2),{topics=[],state=READY,pid=0})
In/Out
2
Clients
ClientxState
setClientState (cs,CON)
cs
cs
setClientState (cs,WAIT)
Receive CONNACK
Send CONNECT
outmsgs = []
cs = (client(1),{topics=[],state=READY,pid=0})
[isClientState (cs,WAIT), hasCONNACK (cs,inmsgs)]
inmsgs
recvMsg(cs,inmsgs)
outmsgs
sendMsg outmsgs (cs,CONNECT)
In
BtoC
1
1`[(client(1),[]),(client(2),[])]
ClientxMessages
Out
CtoB
1
1`[(client(1),CONNECT)]
BrokerxMessages

Fig. 7. CONNECT module after the sendCONNECT occurrence.

The broker will receive the CONNECT request. The broker will register the client in the place ConnectedClients and send back the acknowledgement. Figure 8 shows the situation where client(1) is connected in the broker side and the CONNACK response has been sent back to the client. The function connectClient() used on the arc from the ProcessCONNECT transition to the ConnectedClients place will record the connected client on the broker side. The last step of the connection establishment will occur again in the clients side, where the transition ReceiveCONNACK (in Fig. 7) will be enabled, meaning that the confirmation for the connection of client(1) can proceed.

3.6 Subscription Phase

Starting from the point where both clients are connected (i.e., for both clients, the state is CON as shown at the top of Fig. 9), client(2) will send a SUBSCRIBE request to topic(1) with QoS(1). The place PendingAcks represents a queue that each client maintains to store the PIDs that are waiting to be acknowledged. In this example, the message has assigned a PID = 0, and client(2) is waiting for an acknowledgement to this subscription with a PID = 0. When a client receives a SUBACK (subscribe acknowledgement) it will check that the packet identifier (0

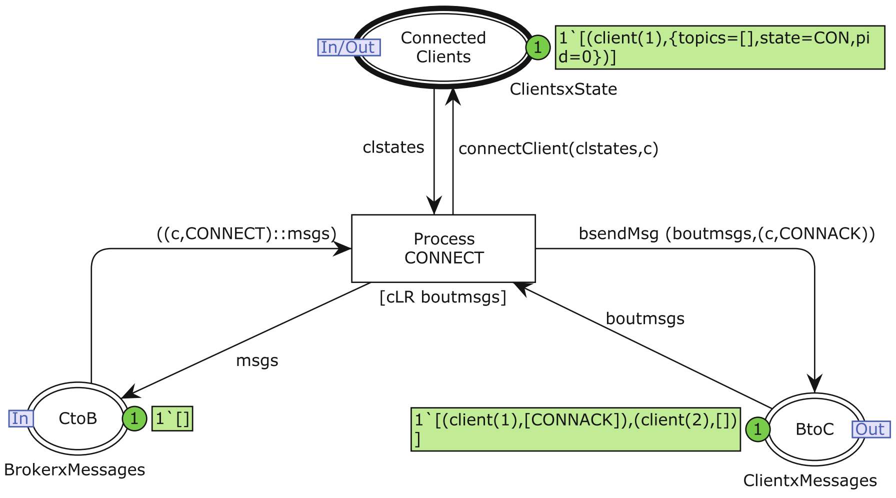


Fig. 8. ProcessCONNECT module after the ProcessCONNECT occurrence.

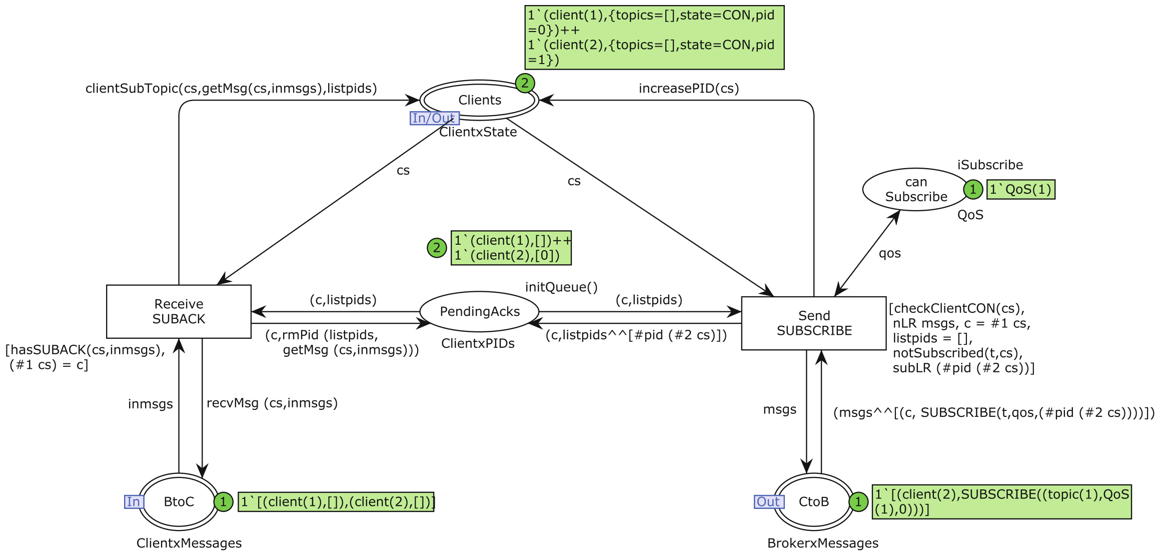

1`(client(1),{topics=[],state=CON,pid=0})++
1`(client(2),{topics=[],state=CON,pid=1})
clientSubTopic(cs,getMsg(cs,inmsgs),listpids)
2
Clients
increasePID(cs)
In/Out
ClientxState
cs
cs
iSubscribe
can Subscribe
1
1`QoS(1)
QoS
2
1`(client(1),[])++
1`(client(2),[0])
qos
initQueue()
Receive SUBACK
(c,listpids)
PendingAcks
(c,listpids)
Send SUBSCRIBE
(c,rmPid (listpids, getMsg (cs,inmsgs)))
ClientxPIDs
(c,listpids^^[#pid (#2 cs)])
[checkClientCON(cs), nLR msgs, c = #1 cs, listpids = [], notSubscribed(t,cs), subLR (#pid (#2 cs))]
[hasSUBACK(cs,inmsgs), (#1 cs) = c]
inmsgs
recvMsg (cs,inmsgs)
msgs
(msgs^^[(c, SUBSCRIBE(t,qos,(#pid (#2 cs))))])
In
BtoC
1
1`[(client(1),[]),(client(2),[])]
ClientxMessages
Out
CtoB
1
1`[(client(2),SUBSCRIBE((topic(1),QoS(1),0)))]
BrokerxMessages

Fig. 9. SUBSCRIBE module after the SUBSCRIBE occurrence.

in this case) is the same to ensure that the correct packet is being received. At the bottom right side of the Fig. 9, the message has been sent to the broker.

We show now the situation where the `SUBSCRIBE` request has been processed by the broker as represented in Fig. 10. The function `brokerSubscribeUpdate()` manages the subscription process, so if the client is subscribing to a new topic, it will be added to the client state stored in the broker. If the client is already subscribed to this topic it will update it. In the example, one can see that `client(1)` keeps the same state, but `client(2)` has appended this new topic to its list. The corresponding `SUBACK` message has been sent to `client(2)` (with the `PID` set to 0) to confirm the subscription. Next, `client(2)` will detect that the response has arrived and it will check that the packet identifiers correspond to each other.

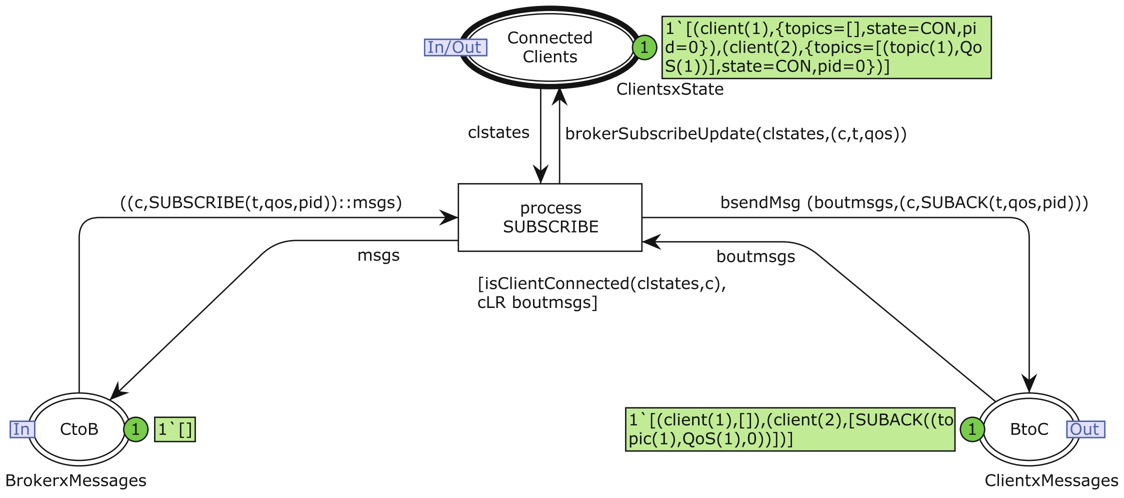

Connected Clients
In/Out
1
1`[(client(1),{topics=[],state=CON,pid=0}),(client(2),{topics=[(topic(1),QoS(1))],state=CON,pid=0})]
ClientsxState
clstates
brokerSubscribeUpdate(clstates,(c,t,qos))
((c,SUBSCRIBE(t,qos,pid))::msgs)
process SUBSCRIBE
bsendMsg (boutmsgs,(c,SUBACK(t,qos,pid)))
msgs
boutmsgs
[isClientConnected(clstates,c), cLR boutmsgs]
In
CtoB
1
1`[]
BrokerxMessages
1`[(client(1),[]),(client(2),[SUBACK((topic(1),QoS(1),0))])]
1
BtoC
Out
ClientxMessages

Fig. 10. ProcessSUBSCRIBE module after occurrence of ProcessSUBSCRIBE.

3.7 Publishing Phase

The publishing process in the considered scenario requires two steps to be completed. First a client sends a PUBLISH in a specific topic, with a specific QoS, which is received by broker. The broker will answer back with the corresponding acknowledgement, depending on the quality of service previously set. Second, the broker, that stores information for all clients, will propagate the PUBLISH sent by the client to any clients subscribing to that topic. We have modelled the clients and broker sides using different submodules depending on the quality of service that is being applied for sending and receiving. In our example, client(1) will publish in topic(1) with a QoS(1). This means that the broker will acknowledge back with a PUBACK to client(1), and will create a PUBLISH message for client(2), which is subscribed to this topic with a QoS(1). In this case, there is no downgrading for the client(2), so the publication process will be similar to step 1, i.e, client(2) will send back a PUBACK to the broker.

Figure 11 shows the situation in the model where client(1) has sent a PUBLISH with a QoS(1) for the topic(1). Similar to the subscription process, the place CtoB holds the message that the broker will receive, and the place Publishing keeps the information (PID and topic in this case) of the packet that needs to be acknowledged. The transition TimeOut models the behaviour for the re-transmission of packets. Quality of service level 1 assures that the message will be received at least once. The TimeOut transition will be enabled to re-send the message until the client has received the acknowledgement from the broker.

The Broker module models the logic for both receiver and sender behaviours. Figure 12 shows a marking corresponding to the state where the broker has processed the PUBLISH request made by client(1), and it has generated both the answer to this client and the PUBLISH message for client(2) (in this case, only one client is subscribed to the topic). The port place BPID (Broker PID), at top right of Fig. 12, will hold a packet identifier for each message that the broker re-publishes to the clients. The port place Publishing keeps information

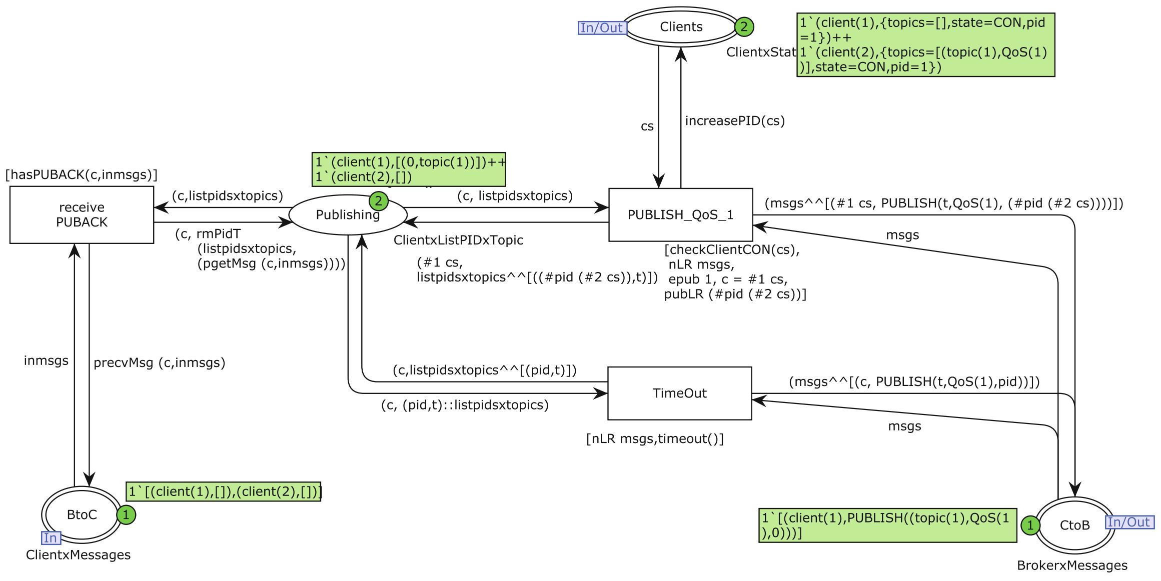

In/Out
Clients
2
ClientxStat
1`(client(1),{topics=[],state=CON,pid=1})++
1`(client(2),{topics=[(topic(1),QoS(1))],state=CON,pid=1})
cs
increasePID(cs)
[hasPUBACK(c,inmsgs)]
1`(client(1),[(0,topic(1))])++
1`(client(2),[])
receive PUBACK
(c,listpidsxtopics)
Publishing
(c, listpidsxtopics)
PUBLISH_QoS_1
(msgs^^[(#1 cs, PUBLISH(t,QoS(1), (#pid (#2 cs))))])
(c, rmPidT (listpidsxtopics, (pgetMsg (c,inmsgs))))
ClientxListPIDxTopic
msgs
(#1 cs, listpidsxtopics^^[((#pid (#2 cs)),t)])
[checkClientCON(cs), nLR msgs, epub 1, c = #1 cs, pubLR (#pid (#2 cs))]
inmsgs
precvMsg (c,inmsgs)
(c,listpidsxtopics^^[(pid,t)])
TimeOut
(msgs^^[(c, PUBLISH(t,QoS(1),pid))])
(c, (pid,t)::listpidsxtopics)
msgs
[nLR msgs,timeout()]
1`[(client(1),[]),(client(2),[])]
BtoC
1
In
ClientxMessages
1`[(client(1),PUBLISH((topic(1),QoS(1),0)))]
1
CtoB
In/Out
BrokerxMessages

Fig. 11. PUBLISH_QoS_1 module after the PUBLISH_QoS_1 occurrence.

for all the clients that will acknowledge back the publish messages transmitted by the broker. Again, a TimeOut is modelled which, in this case, creates PUBLISH messages for all the clients subscribed to the topic in question. In the BtoC place (bottom right of Fig. 12), one can see that both messages have been sent, one for the original sender `client(1)` (PUBACK packet), and one for the only receiver `client(2)` (PUBLISH packet).

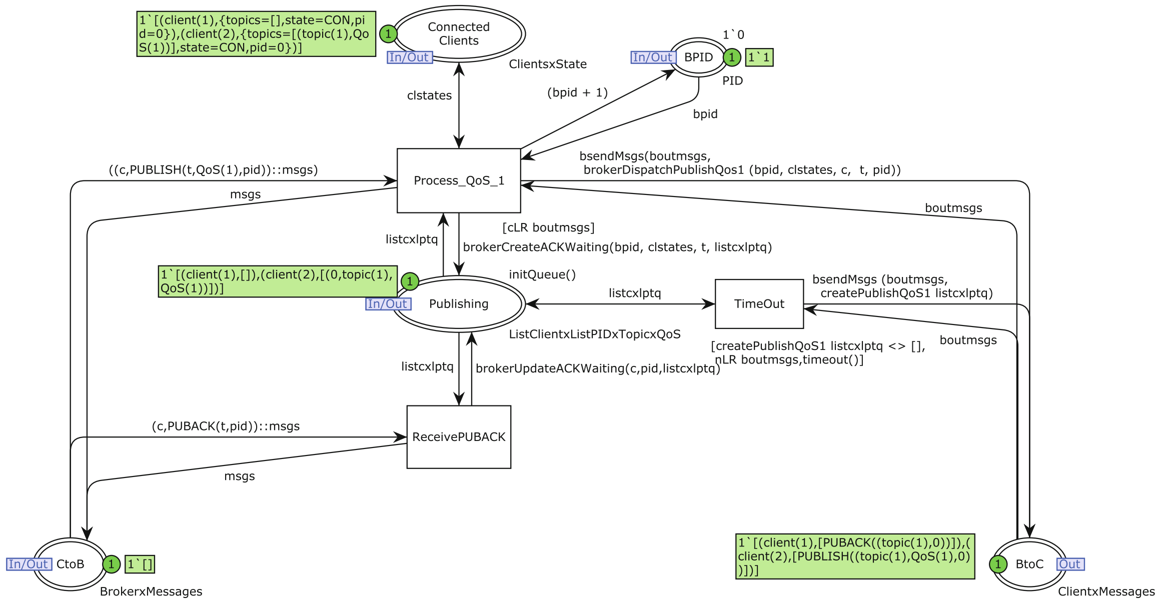

1`[(client(1),{topics=[],state=CON,pid=0}),(client(2),{topics=[(topic(1),QoS(1))],state=CON,pid=0})]
1
Connected Clients
In/Out
ClientsxState
1`0
In/Out
BPID
1
1`1
PID
clstates
(bpid + 1)
bpid
bsendMsgs(boutmsgs, brokerDispatchPublishQos1 (bpid, clstates, c, t, pid))
((c,PUBLISH(t,QoS(1),pid))::msgs)
Process_QoS_1
msgs
boutmsgs
[cLR boutmsgs]
listcxlptq
brokerCreateACKWaiting(bpid, clstates, t, listcxlptq)
1`[(client(1),[]),(client(2),[(0,topic(1),QoS(1))])]
1
initQueue()
listcxlptq
TimeOut
bsendMsgs (boutmsgs, createPublishQoS1 listcxlptq)
In/Out
Publishing
ListClientxListPIDxTopicxQoS
[createPublishQoS1 listcxlptq <> [], nLR boutmsgs,timeout()]
boutmsgs
listcxlptq
brokerUpdateACKWaiting(c,pid,listcxlptq)
(c,PUBACK(t,pid))::msgs
ReceivePUBACK
msgs
1`[(client(1),[PUBACK((topic(1),0))]),(client(2),[PUBLISH((topic(1),QoS(1),0))])]
In/Out
CtoB
1
1`[]
BrokerxMessages
1
BtoC
Out
ClientxMessages

Fig. 12. Process_QoS_1 module after the Process_QoS_1 occurrence.

To finish the process, `client(2)` will notice that there has been a message published in `topic(1)`. Since `client(2)` is subscribed to this topic with `QoS(1)`, it must send a PUBACK acknowledgement to the broker to confirm that it has received the published message. Figure 13 shows the Receive_QoS_1 submodule in the clients side. The transition Receive_QoS_1 has been fired meaning that client(2) has received the publish message from the broker, and has sent the corresponding PUBACK. When the broker detects the incoming PUBACK message, it will check if there is some confirmation pending in the Publishing place (in Fig. 12 where client(2) is waiting for a PID = 0 in topic(1) with QoS(1).

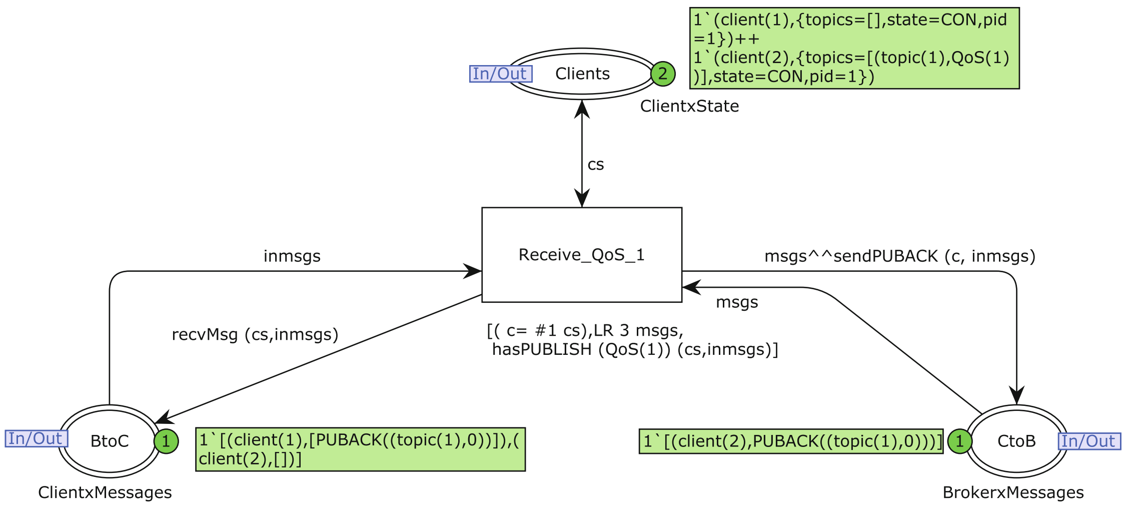

1`(client(1),{topics=[],state=CON,pid=1})++
1`(client(2),{topics=[(topic(1),QoS(1))],state=CON,pid=1})
In/Out
Clients
2
ClientxState
cs
Receive_QoS_1
inmsgs
msgs^^sendPUBACK (c, inmsgs)
msgs
recvMsg (cs,inmsgs)
[(c= #1 cs),LR 3 msgs,
hasPUBLISH (QoS(1)) (cs,inmsgs)]
In/Out
BtoC
1
1`[(client(1),[PUBACK((topic(1),0))]),(client(2),[])]
ClientxMessages
1`[(client(2),PUBACK((topic(1),0)))]
1
CtoB
In/Out
BrokerxMessages

Fig. 13. Receive module after the transition Receive_QoS_1 occurrence

3.8 Findings

In the course of constructing the CPN model based on the informal MQTT specification, we encountered several parts that were vaguely defined and which could lead developers to obtain different implementations. The most significant issues are detailed below.

- There is a gap in the specification related to the MQTT protocol being described to run over TCP/IP, or other transport protocols that provide ordered, lossless and bidirectional connections. The QoS level 0 description establishes that message loss can occur, but the specification is not clear as to whether this is related to termination of TCP connections and/or clients connecting and disconnecting from the broker.
- It is specified that the receiver (assuming the broker role) is not required to complete delivery of the application message before sending the PUBACK (for QoS1) or PUBREC or PUBCOMP (for QoS2) and the decision of handling this is up to the implementer. For instance, in the case of QoS level 2, the specification provides two alternatives with respect to forward the publish request to the subscribers: (1) The broker will forward the messages when it receives the PUBLISH from the original sender; or (2) The broker will

forward the messages after the reception of the PUBREL from the original sender. Even it is assured that either choice does not modify the behaviour of the QoS level 2, this could lead to different implementation decisions and therefore consequent interoperability problems.
- The documentation specifies that when the original sender receives the PUBACK packet (with a QoS level 1), ownership of the application message is transferred to the receiver. It is unclear how to determine that the original sender has received the PUBACK packet. The same applies for QoS level 2 and the PUBREC packet.

4 Model Validation and Verification

During development of the MQTT protocol model we used single-step and automatic simulation to test the proper operation of the model. To perform a more exhaustive validation of the model, we have conducted state space exploration of the model and verified a number of behavioural properties.

We have conducted the verification of properties using an incremental approach consisting of three steps. In the first step we include only the parts related to clients connecting and disconnecting. In the second step we add subscribe and unsubscribe, and finally in the third step we add data exchange considering the three quality of service levels in turn. At each step, we include verification of additional properties. The main motivation underlying this incremental approach is to be able to control the effect of the state explosion problem. Errors in the model will often manifest themselves in small configurations and leading to a very large state space. Hence, by incrementally adding the protocol features, we can mitigate the effect of this phenomenon. We identified several modelling errors in the course of conducting this incremental model validation based on the phases of the MQTT protocol.

In addition, we have developed a mechanism to be able to explore different scenarios and check the behavioural properties against them fully automatically. This has been done by providing the model with a set of parameterized options, which we can easily change. This feature allows us to first modify add or remove new configurations, and secondly to run them automatically. For each new modification in the parameters, we always run the six incremental executions and check the behavioural properties. Among others, one can quickly change the number of clients, the roles that such clients can perform (either subscriber, publisher, or both), switch between acyclic or cyclic version (where reconnection of clients is allowed) or enable/disable the possibility to retransmit packets (by means of timeouts).

To obtain a finite state space, we have to limit the number of clients and topics, and also bound the packet identifiers. It can be observed that there is no interaction between clients and brokers across topics as the protocol treats each topic in isolation. Executing the protocol with multiple topics is equivalent to running multiple instances of the protocol in parallel. We therefore only consider a single topic for the model validation. Initially, we consider two clients. The packet identifiers are incremented throughput the execution of the different phases of the protocol (connect, subscribe, data exchange, unsubscribe, and disconnect). This means that we cannot use a single global bound on the packet identifiers as a client could reach this bound, e.g., already during the publish phase and hence the global bound would prevent (block) a subsequent unsubscribe to take place. We therefore introduce a local upper bound on packet identifiers for each phase. This local bound expresses that the given phase may use packet identifiers up to this local bound. Note that the use of bounds does not guarantee that the client uses packet identifiers up to bound. It is the guard on the transitions sending packets from the clients that ensures that these local bounds are enforced. Finally, we enforce an upper bound on the number of messages that can be in the message queues on the places CtoB and BtoC.

Below we describe each step of the model validation and the behavioural properties verified. The properties verified in each step include the properties from the previous step. We summarise the experimental results at the end. For the actual checking of properties, we have used the state and action-oriented variant of CTL supported by the ASK-CTL library of CPN Tools.

Step 1 – Connect and Disconnect. In the first step, we consider only the part of the model related to clients connecting and disconnecting to the broker. We consider the following behavioural properties:

S1-P1-ConsistentConnect. The clients and the broker have a consistent view of the connection state. This means that if the clients side is in a connect state, then also the broker has the client recorded as connected.

S1-P2-ClientsCanConnect. For each client, there exists a reachable state in which the client is connected to the broker.

S1-P3-ConsistentTermination. In each terminal state (dead marking), clients are in a disconnect state, the broker has recorded the clients as disconnected, no clients are recorded as subscribed on both clients and broker sides, and there are no outstanding messages in the message buffers.

S1-P4-PossibleTermination. The protocol can always be terminated, i.e., a terminal state (dead marking) can always be reached.

The two properties S1-P3 and S1-P4 imply partial correctness of the protocol as it states that the protocol can always be terminated, and if it terminates, then it will be in a correct state. The state space obtained in this step is acyclic when we do not allow reconnections. This together with S1-P3 implies the stronger property of eventual correct termination. This is, however, more a property of how the model has been constructed as in a real implementation there is nothing forcing a client to disconnect.

Step 2 – Subscribe and Unsubscribe. In the second step, we add the ability for the clients to subscribe and unsubscribe (in addition to connect and disconnect from step 1). For subscribe and unsubscribe we additionally consider the following properties:

S2-P1-CanSubscribe. For each of the clients, there exists states in which both the clients and the broker sides consider the client to be subscribed.

S2-P2-ConsistentSubscription. If the broker side considers the client to be subscribed, then the clients side considers the client to be subscribed.

S2-P3-EventualSubscribed. If the client sends a subscribe message, then eventually both the clients and the broker sides will consider the client to be subscribed.

S2-P4-CanUnsubscribe. For each client there exists executions in which the client sends an unsubscribe message.

S2-P5-EventualUnsubscribed. If the client sends an unsubscribe message, then eventually both the clients and the broker sides considers the client to be unsubscribed.

It should be noted that for property S2-P2, the antecedent of the implication deliberately refers to the broker side. This is because the broker side unsubscribes the client upon reception of the unsubscribe message, whereas the client side does not remove the topic from the set of subscribed topics until the subscribe acknowledgement message is received from the broker. Hence, during unsubscribe, we may have the situation that the broker has unsubscribed the client, but the subscribe acknowledgement has not yet been received on the client side.

Step 3 – Publish and QoS Levels. In this step we also consider publication of data for each of the three quality of service levels. As we do not model the concrete data contained in the messages, we use the packet identifiers attached to the message published to identity the packets being sent and received by the clients. In order to reduce the effect of state explosion, we verify properties for each QoS level in isolation. To make it simpler to check properties related to data being sent, we record for each client the packet identifiers of messages sent. For all three service levels, we consider the following properties:

S3-P1-PublishConnect. A client only publishes a message if it is in a connected state.

S3-P2-CanPublish. For each client there exists executions in which the client publishes a message.

S3-P3-CanReceive. For each client there exists executions in which the client receives a message.

S3-P4-Publish. Any data (packet identifiers) received on the client side must also have been sent on the client side.

S3-P5-ReceiveSubscribed. A client only receives data if it is subscribed to the topic, i.e., the client side considers the client to be subscribed.

It should be noted that it is possible for a client to publish to a topic without being subscribed. The only requirement is that the client is connected to the broker. What data can correctly be received depends on the quality of service level considered. We therefore have one of the following three properties depending on the quality of service considered.

S3-P6-Publish-QoS0. The data (packet identifiers) received by the subscribing clients must be a subset of the data (packet identifiers) sent by the clients.

S3-P7-Publish-QoS1. The data sent on the client side must be a subset of the multi-set of packets received by the subscribing clients.

S3-P8-Publish-QoS2. The data received by each client is identical to the packet identifiers sent by the clients.

To check the above properties related to data received, we accumulate the packet identifiers received such that they can be compared to the packet identifiers sent. To simplify the verification of data received, we force (using priorities) both clients to be subscribed before data exchange takes places since otherwise the data that can be received depends on the time at which the clients were subscribed and unsubscribed.

Table 1 summarises the validation statistics where each configuration (scenario) is represented by a row comprised of Clients, Roles and Version. We report the size of the state space (number of states/number of arcs) and the number of dead markings (written below the state space size). We do not show the dead markings for the cyclic configurations as they are always 0. The columns S3.1, S3.2 and S3.3 correspond to the results considering QoS level 0, QoS level 1 and QoS level 2, respectively. Cells containing a hyphen represent configurations where the state space exploration and model checking did not complete within 12 h which we used as a cut-off point.

Table 1. Summary of configurations and experimental results for model validation

N° Clients	Roles	Version	State space (states/arcs) Number of dead markings				
			Step 1	Step 2	Step 3		
					S3.1	S3.2	S3.3
2	1 sub/1 pub	Acyclic	35/48 1	258/480 4	622/1074 21	1312/2616 21	3234/6394 21
	2 sub-pub		35/48 1	1849/4120 16	4282/8840 70	11462/23934 70	43791/85682 76
	1 sub/1 pub	Cyclic	24/38	271/547	1149/2265	2376/5045	5996/12267
	2 sub-pub		24/38	2954/6798	8138/17714	20362/43572	79913/159254
3	2 sub/1 pub	Acyclic	163/292 1	9529/25408 16	31765/76848 165	103176/262254 165	–
	1 sub/2 pub		163/292 1	1262/2862 4	10360/21604 90	46721/120321 90	–
	2 sub/1 pub	Cyclic	84/175	12650/35875	87450/235887	254095/679920	–
	1 sub/2 pub		84/175	1057/2662	23817/59342	101794/279871	–

5 Conclusions and Related Work

We have presented a formal CPN model based on the most recent specification of the MQTT protocol (version 3.1.1 [3]). The constructed CPN model represents a formal and executable specification of the MQTT protocol. While performing an exhaustive review of the MQTT specification to develop the model, we found several issues that might lead to not interoperable implementations. Consequently, this may add extra complexity for interoperability in the heterogeneous ecosystem that surrounds the application of a protocol such as MQTT.

The model has been built using a set of general CPN modelling patterns ensuring modular organisation of the protocol roles and protocol processing logic. Furthermore, we incorporated parameterization that makes it easy to change, among others, the number of clients and topics without having to make changes in the CPN model structure. In addition, we have applied modelling patterns related to the input and output message queues of the clients (publishers and subscribers) and brokers. These modelling patterns apply generally for modelling distributed systems that include one-to-one and one-to-many communication.

For the validation of the model, we have conducted simulation and state space exploration in order to verify an extensive list of behavioural properties and thereby validate the correctness of the model. In particular, our modelling approach makes it possible to apply an incremental verification technique where the functionality of the protocol is gradually introduced and properties are verified in each incremental step. A main advantage of the modelling patterns used for communication and message queues is that they avoid intermediate states and hence contributes to making state space exploration feasible.

There exists previous work on modelling and validation of the MQTT protocol. In [11], the authors uses the UPPAAL SMC model checker [7] to evaluate different quantitative and qualitative (safety, liveness and reachability) properties against a formal model of the MQTT protocol defined with probabilistic timed automata. Compared to their work, we have verified a larger set of behavioural properties using the incremental approach adding more operations in each step. In [13], tests are conducted over three industrial implementations of MQTT against a subset of the requirements specified in the MQTT version 3.1.1 standard using the TTCN-3 test specification language. In comparison to our work, test-based approaches do not cover all the possible executions but only randomly generated scenarios. With the exploration of state spaces, we considered all the possible cases. In [2], the authors first define a formal model of MQTT based on timed message-passing process algebra, and they conduct analysis of the three QoS levels. In contrast, our work is not limited to the publishing/subscribing process, but considers all operations of the MQTT specification.

We are planning to extend the features supported by the model in order to be able to simulate more sophisticated scenarios. For instance, we will allow the model to deal with persistence of data, so clients can receive the messages on reconnections lost suddenly in the middle of some operation. Furthermore, we plan to improve the mechanism to simulate loss of packets as an extension of the timeout system already implemented. In addition to aiding in the development of compatible MQTT implementations, the CPN MQTT model may also be used as basis for testing of MQTT implementations. As part of future work, we plan to explore model-based testing of MQTT protocol implementations following the approach presented in [16].

References

1. Adamski, M.A., Karatkevich, A., Wegrzyn, M.: Design of Embedded Control Systems, vol. 267. Springer, Boston (2005). https://doi.org/10.1007/0-387-28327-7
2. Aziz, B.: A formal model and analysis of an IoT protocol. Ad Hoc Netw. **36**, 49–57 (2016)
3. Banks, A., Gupta, R.: MQTT Version 3.1.1. OASIS Standard, 29 (2014). http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html
4. Baresi, L., Ghezzi, C., Mottola, L.: On accurate automatic verification of publish-subscribe architectures. In: Proceedings of the 29th International Conference on Software Engineering, pp. 199–208. IEEE Computer Society (2007)
5. Billington, J., Diaz, M.: Application of Petri Nets to Communication Networks: Advances in Petri Nets, vol. 1605. Springer, Heidelberg (1999). https://doi.org/10.1007/BFb0097770
6. Chen, S., Xu, H., Liu, D., Hu, B., Wang, H.: A vision of IoT: applications, challenges, and opportunities with China perspective. IEEE Internet Things J. **1**(4), 349–359 (2014)
7. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Poulsen, D.B.: Uppaal SMC tutorial. Int. J. Softw. Tools Technol. Transf. **17**(4), 397–415 (2015)
8. Desel, J., Reisig, W., Rozenberg, G. (eds.): Lectures on Concurrency and Petri Nets, Advances in Petri Nets. LNCS, vol. 3018. Springer, Heidelberg (2004). https://doi.org/10.1007/b98282
9. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.-M.: The many faces of publish/subscribe. ACM Comput. Surv. (CSUR) **35**(2), 114–131 (2003)
10. Garlan, D., Khersonsky, S., Kim, J.S.: Model checking publish-subscribe systems. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 166–180. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44829-2_11
11. Houimli, M., Kahloul, L., Benaoun, S.: Formal specification, verification and evaluation of the MQTT protocol in the Internet of Things. In: Mathematics and Information Technology, pp. 214–221. IEEE (2017)
12. Jensen, K., Kristensen, L.: Coloured Petri nets: a graphical language for modelling and validation of concurrent systems. Commun. ACM **58**(6), 61–70 (2015)
13. Mladenov, K.: Formal verification of the implementation of the MQTT protocol in IoT devices. Master thesis, University of Amsterdam (2017)
14. MQTT essentials part 3: Client, broker and connection establishment. https://www.hivemq.com/blog/mqtt-essentials-part2-publish-subscribe
15. Rodriguez, A., Kristensen, L.M., Rutle, A.: Complete CPN model of the MQTT Protocol. via Dropbox. http://www.goo.gl/6FPVUq
16. Wang, R., Kristensen, L., Meling, H., Stolz, V.: Application of model-based testing on a quorum-based distributed storage. In: Proceedings of PNSE 2017, volume 1846 of CEUR Workshop Proceedings, pp. 177–196 (2017)
17. Wortmann, F., Flüchter, K.: Internet of things. Bus. Inf. Syst. Eng. **57**(3), 221–224 (2015)
18. Zanolin, L., Ghezzi, C., Baresi, L.: An approach to model and validate publish/-subscribe architectures. Proc. SAVCBS **3**, 35–41 (2003)

PAPER **B**

VERIFICATION OF THE MQTT IOT PROTOCOL USING PROPERTY-SPECIFIC CTL SWEEP-LINE ALGORITHMS

Alejandro Rodríguez, Lars Michael Kristensen, Adrian Rutle

In Transactions on Petri Nets and Other Models of Concurrency XV, volume 12530 of Lecture Notes in Computer Science, pages 165-183, **Springer Nature**, 2021

Verification of the MQTT IoT Protocol Using Property-Specific CTL Sweep-Line Algorithms

Alejandro Rodríguez(✉), Lars Michael Kristensen, and Adrian Rutle

Department of Computer Science, Electrical Engineering, and Mathematical Sciences, Western Norway University of Applied Sciences, Bergen, Norway
{arte,lmkr,aru}@hvl.no

Abstract. MQTT is a publish-subscribe communication protocol being increasingly used for implementing internet-of-things (IoT) applications. In earlier work we have developed a formal and executable model of the MQTT protocol using Coloured Petri Nets (CPNs) and performed an initial verification of behavioural properties. The contribution of this paper is to investigate the use of the sweep-line method for verification of the MQTT CPN model in order to alleviate the effect of the state explosion problem. We formulate the behavioural properties using Computation Tree Logic (CTL) and show how to formulate a progress measure for the sweep-line method based on the main phases of the MQTT protocol. To perform the verification of properties, we provide some property-specific CTL model checking algorithms compatible with the sweep-line method.

Keywords: Coloured Petri Nets · Modelling · Verification · Communication protocols · Internet of Things

1 Introduction

The development of distributed software systems is challenging, and one of the main approaches to tackle the challenges is to build an executable model of the system prior to implementation and deployment. Coloured Petri Nets (CPNs) [13] is a formal modelling formalism convenient for specifying complex concurrent and distributed systems. CPN Tools [9,15] is a software tool that supports the construction, simulation (execution), state space analysis, and performance analysis of CPN models. One of the key functionalities of CPN Tools is the ability to perform model checking [1] of the modelled system. This means that one can generate the state space (the set of reachable states) of a system in order to verify key behavioural properties. Temporal logics [23] such as Computation Tree Logic (CTL) and Linear Temporal Logic (LTL) are widely used to express behavioural properties of systems.

MQTT [2] is a publish-subscribe messaging protocol for IoT suited for constrained application domains such as Machine-to-Machine communication

M. Koutny et al. (Eds.): ToPNoC XV, LNCS 12530, pp. 165–183, 2021.
https://doi.org/10.1007/978-3-662-63079-2_8

(M2M) and IoT contexts. MQTT is designed with the aim of being light-weight and easy to implement. In earlier work [19], we have developed a formal and executable specification of MQTT motivated by the fact that until now, the protocol has only been specified using an (ambiguous) natural language specification. MQTT contains relatively complex protocol logic for handling connections, subscriptions, and quality of service levels related to message delivery.

Our initial verification experiments were conducted using ordinary full state spaces and clearly highlighted the presence of the state explosion problem [8,22]. This was caused by the exponential growth in the number of reachable states of the system with respect to the number of clients, packets, and topics. A large part of the model checking research has aimed at developing techniques for alleviating this inherent complexity problem. This includes several different families of reduction methods such as partial-order reduction methods [7] that reduce the number of interleaving execution considered, and hash compaction [21] which provides a compact representation of states with a small probability of not covering the complete state space. Since the amount of memory is often the limiting factor in model checking, we focus on the family of methods that combat state explosion by deleting states from memory during state space exploration. Specifically, we consider the sweep-line method [12] which is based on the idea of exploiting a notion of progress exhibited by many systems. We focus on CTL because CPN Tools implements a CTL-based temporal logic called ASK-CTL [3] which enables queries taking into account both state and event information. Furthermore, CTL is able to capture the behavioural properties of interest for the MQTT protocol.

The contribution of this paper is twofold: (1) the implementation of the sweep-line method using the Standard ML (SML) language together with the ability of performing model checking of certain behavioural properties specified using tailored CTL sweep-line model checking algorithms based on [17]; and (2) the application of sweep-line based CTL model checking to our CPN model of the MQTT IoT protocol. It should be noted that there already exists work on LTL model checking using the sweep-line method [10], but several of the behavioural properties that we aim to verify for MQTT are true CTL properties, i.e., not expressible in LTL [22,24].

The rest of this paper is organised as follows. In Sect. 2 we introduce the sweep-line method and in Sect. 3 we provide the property-specific CTL model checking algorithm that we employ for the verification. Section 4 gives a brief review of the CPN model of the MQTT protocol. We describe the experiments carried out and the results obtained in Sect. 5. Finally, in Sect. 6, we sum up the conclusions and outline directions for future work. The reader is assumed to be familiar with the basic concepts of CPNs and CTL model checking techniques. This paper is based upon the workshop paper [20] and the conference paper [17].

2 The Sweep-Line State Space Exploration Method

The sweep-line method [4] is aimed at systems for which it is possible to define a measure of progress based on the states of the system. A progress measure maps each state of the system into a *progress value* and is in most cases specific for the system under consideration. In this paper, we consider the version of the sweep-line algorithm for *monotonic progress measures*. The key property of a monotonic progress measure is that for any given state s, all states reachable from s have a progress value which is greater than or equal to the progress value of s. This means that a monotonic progress measure preserves the reachability relation. Having defined a progress measure of the system makes it possible to organise the state space into *layers* such that states that share the same progress value belong to the same layer.

The basic idea of the sweep-line method is to explore the state space in a least-progress-first order, one layer at a time, such that once all states in a given layer have been processed, they are removed from memory and the exploration proceeds to the next layer [12]. In conventional state space exploration, the states are kept in memory to recognise already visited states. However, a monotonic progress measure guarantees that states which have a progress value that is strictly less than the minimal progress value of those states for which successors have not yet been calculated can never be reached again. It is therefore safe to delete such states from memory which significantly reduces the memory usage during the state space exploration.

The progress exploited by the sweep-line method and formalised in the form of a progress measure is defined below in Definition 1 where S denotes the set of system states, $s_0 \in S$ denote the initial state, $s \rightarrow^* s'$ denotes that $s' \in S$ is reachable from $s \in S$ via some number of transitions, and $reach(s_0)$ the set of states reachable from the initial state.

Definition 1 (Monotonic Progress Measure). A **monotonic progress measure** is a tuple $\mathcal{P} = (O, \sqsubseteq, \Psi)$ such that O is a set of **progress values**, $\sqsubseteq$ is a total order on O, and $\Psi : S \rightarrow O$ is a **progress mapping** such that $\forall s, s' \in reach(s_0) : s \rightarrow^* s' \Rightarrow \Psi(s) \sqsubseteq \Psi(s')$. □

A progress measure is non-monotonic when there is at least one *regress edge*, i.e., an edge where the source state has a larger progress value than the destination state. A generalised version of the sweep-line method that can handle non-monotonic progress measures and regress edges also exists [14], but is not the focus of our work. It was already proved [12] that the sweep-line method guarantees full coverage of the state space, and in the case of a monotonic progress measure it terminates after having explored each reachable state once. In the case of a non-monotonic progress measures, termination is still guaranteed but some states may be explored multiple times.

Algorithm 1 based on [12] specifies the sweep-line algorithm for monotonic progress measures. The algorithm starts with a hash table of visited states and a priority queue on progress values containing the states that are still to be processed. Both are initialized at the beginning with the initial state s_0 (lines 2-3). The progress value for the current (initial) layer ψ_c is also initialized in line 4. Then, the algorithm executes a loop (lines 5-28) which ends when all the reachable states have been processed. For each iteration, we select one of the

```
   Data:
   Nodes ▷ Hash table of visited states currently stored.
   Unprocessed ▷ Priority queue of unprocessed states.
   Layer ▷ List of states processed in the current layer.
   $\psi_c$ ▷ Progress value for current layer.
   $\Phi$ ▷ Property to be verified.
   Result: True if the property is satisfied, false otherwise.
 1 begin
 2     Nodes.insert($s_0$)
 3     Unprocessed.insert($s_0$)
 4     $\psi_c \longleftarrow \psi(s_0)$
 5     while $\neg$(Unprocessed.isEmpty()) do
           /* node with lowest progress measure                    */
 6         s $\longleftarrow$ Unprocessed.getMinElement()
 7         if $\psi_c \sqsubset \psi(s)$ then
 8             if $\neg$ (checkProperty(Layer, $\Phi$)) then
 9                 return false
10             end
11             forall $s' \in$ Layer do
12                 Nodes.delete($s'$)
13             end
14             Layer $\longleftarrow \emptyset$
               /* Update progress measure for current layer        */
15             $\psi_c \longleftarrow \psi(s)$
16         end
17         Layer.insert(s)
           /* For every successor state of s                        */
18         forall (t, $s'$) such that $s \xrightarrow{t} s'$ do
19             if $\neg$(Nodes.contains($s'$)) then
20                 Nodes.insert($s'$)
21                 if ($\psi(s) \sqsupset \psi(s')$) then
22                     RaiseException('Regress edge found')
23                 else
24                     Unprocessed.insert($s'$)
25                 end
26             end
27         end
28     end
29     return true
30 end
```

Algorithm 1: Sweep-line algorithm for monotonic progress measures

states with the lowest progress value among the unprocessed states (line 6). The condition in line 7 checks if the progress value of the layer is strictly less than the progress value of the selected state; if so, we are about to move into the next layer. This is the point where we invoke the property-specific CTL model checking algorithm for the property Φ using the CHECKPROPERTY procedure at line 8. If the CHECKPROPERTY determines that the property is violated, then we return false and the algorithm stops. The implementation of CHECKPROPERTY is the subject of the next section. In line 18, we use $s \xrightarrow{t} s'$ to denote that the transition t is enabled in state s, and that the occurrence of t in s leads to the state s'. If the property is never violated the algorithm returns true at the end of the execution (line 29).

3 CTL Property Checking Algorithms

CTL [5] is an important branching temporal logic that is sufficiently expressive for the formulation of an important set of behavioural system properties. Even though a large set of properties can be specified using the semantics of CTL, there are some restrictions when applying them with the sweep-line method algorithm. The challenge of combining CTL model checking with the sweep-line method is that conventional algorithms for CTL model checking propagate information backwards from a state to its predecessors [6]. This follows the opposite workflow than the forward progress-first exploration that the sweep-line method performs.

In this paper, we do not consider the full CTL, but only formulas of the $AG\{EF, AF\}$-fragment that can be obtained from the following grammar, where p as an atomic state proposition and ϕ is called a *state predicate*:

$$\begin{aligned}
\Phi &::= \mathbf{AG}\,\psi \mid \psi \\
\psi &::= \mathbf{EF}\,\phi \mid \mathbf{AF}\,\phi \mid \phi \\
\phi &::= p \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi
\end{aligned}$$

The formulas expressing behavioural properties to be verified are interpreted over the paths of the state space as informally explained below:

Property - AG ψ "Invariantly", which holds if ψ holds in all states that are reachable from the current state.

Property - EF ϕ "Holds potentially" or "possibly", which holds if it is possible to find a state reachable from the current state where ϕ holds.

Property - AF ϕ "Holds eventually" which holds if from the current state, a state satisfying ϕ is always eventually reached.

Property - AG EF ϕ "Always possible", which holds if from any state reachable from the current state, a state satisfying ϕ can always be reached.

Property - AG AF ϕ "Always eventually", which holds if from any state reachable from the current state, a state satisfying ϕ is always eventually reached.

We say that a formula (property) Φ holds if Φ holds in the initial state s_0. To model check the **AF EF** and **AG AF** properties, we exploit the set of *strongly connected components (SCC).* A strongly connected component of a directed graph is a maximal subgraph determined by nodes that are mutually reachable. A strongly connected component is *terminal* if no states in the component has outgoing edges to states in other components. It should be noted that when checking the **AG AF** and **AF** properties we implicitly add a self-loop to any terminal states, i.e. (deadlocked) states without enabled transitions.

Because of the monotonicity of the progress measure, each strongly connected component only contains nodes belonging to the same layer and is hence always contained in a single layer. This is formally stated in the proposition below.

Proposition 1. *Let $\mathcal{P} = (O, \sqsubseteq, \psi)$ a monotonic progress measure, SCC be the set of strongly connected components, and let $scc \in SCC$ be a strongly connected component. Then: $\forall s, s' \in scc : \psi(s) = \psi(s')$.*

Proof. Assume that there exists an $scc \in SCC$ and states $s, s' \in scc$ such that $\psi(s) \neq \psi(s')$. Hence either $\psi(s) \not\sqsubseteq \psi(s')$ or $\psi(s') \not\sqsubseteq \psi(s)$. Since s and s' are in the same *scc*, then they are mutually reachable and therefore there must exist a pair of states (s_i, s_j) on the path from either s to s' or s' to s such that $\psi(s_i) \not\sqsubseteq \psi(s_j)$. This contradicts the fact that the progress measure is monotonic.

Based on this, we can compute the strongly connected components for a given layer immediately before we delete the nodes in the current layer and move to the next one. The algorithm checks the property depending on the form of the property as outlined below.

Property - AG ϕ. We check that every node within the layer satisfies ϕ. If ϕ does not hold in one of them, we return false and abort the exploration.

Property - EF ϕ. If at least one state is encountered that satisfies ϕ, then true is returned and the execution finishes. Thus, false will be returned if at the end of the exploration not a single state satisfying ϕ has been found.

Property - AG EF ϕ. For this property, we first compute the *SCC* of the given Layer. The property will not be satisfied and therefore the procedure will finish the execution returning false, if any *scc* among the *SCC* of Layer is terminal and ϕ does not hold in any of the states contained in *scc*.

Property - AG AF ϕ. For this property, we first compute the *SCC* of the given Layer. We then remove the states that satisfy ϕ. If the resulting set of nodes has a cycle, then the property is violated and therefore the execution immediately finishes returning false.

Property - AF ϕ. This property can be checked in a similar fashion as **AG AF** ϕ with the modification that we can truncate the search at *SCC* where all cycles include a state satisfying ϕ.

The two first properties can easily be checked by just inspecting each state encountered during the sweep-line state space exploration. For verification of the two other properties, we invoke the procedure CHECKPROPERTY at the moment where the algorithm is about the leave the current layer and move into the next ones. We do not detail the checking of **AF** ϕ as it is very similar to **AG AF** ϕ as explained above.

A consequence of Proposition 1 is that SCC can be computed by considering one layer at a time. Furthermore, Theorem 1 ensures that the sweep-line method covers all reachable states which means that we will encounter all strongly connected components at some stage. The remaining step consist of linking the inspection of SCC to the model checking of the **AG EF** and **AG AF** properties. This is done in the proposition below which formalises the requirements informally introduced above.

Proposition 2. *Let SCC be the set of strongly connected components of M, $SCC_T \subseteq SCC$ the set of terminal strongly connected components, and let ϕ be a state predicate. Then:*

1. $\mathbf{AG\,EF}\,\phi$ *is satisfied* $\Leftrightarrow \forall scc \in SCC_T\ \exists s \in scc : \phi(s)$
2. $\mathbf{AG\,AF}\,\phi$ *is satisfied* $\Leftrightarrow \forall scc \in SCC : scc \setminus \{s \in scc : \phi(s)\}$ *is acyclic*

Proof. First we prove 1. Assume that $\mathbf{AG\,EF}\,\phi$ holds and there exists a terminal scc named scc_t such that no states in scc_t satisfy ϕ. Since all states belong to some scc, then we can find a path from the initial state to a state s in scc_t. Since scc_t is terminal and do not contain states satisfying ϕ, then we can no longer reach states that satisfies ϕ from s. Hence, $\mathbf{AG\,EF}\,\phi$ cannot hold. Assume that each terminal scc contains a state satisfying ϕ and let s be any reachable state. Since we cannot have cycles that spans multiple SCC and all states belong to some scc, there must exists a path from the scc to which s belongs to a state s' in some terminal scc. Within this terminal scc, all states are mutually reachable and by our assumption at least one state in there satisfies ϕ. Hence, $\mathbf{AG\,EF}\,\phi$ holds.

Next we prove 2. Assume that $\mathbf{AG\,AF}\,\phi$ holds and there exists a scc such that when all states satisfying ϕ are removed from scc we still have a cycle consisting of states in scc. In that case, we can find a path $s_0, s_1 \ldots s$ leading to a state s on this cycle, and we can then extend this to an infinite path by repeating the states on the cycle to which s belong. Since no state on the cycle satisfy ϕ, then $\mathbf{AG\,AF}\,\phi$ cannot hold. Hence, we cannot have such cycles. Assume now that each strongly connected component becomes acyclic when removing states satisfying ϕ. Since all cycles belongs to some strongly connected component, then we cannot have cycles where no states satisfy ϕ. Thus, from any states on an infinite path we must eventually encounter a state satisfying ϕ which means that $\mathbf{AG\,AF}\,\phi$ holds.

Based on Proposition 2 we can now specify the CHECKPROPERTY procedure which is given in Algorithm 2. The procedure first computes the SCC of the given layer $\mathcal{L}$. Here any algorithm for computing SCC can be used, and we do not specify this further. Based on the SCC and Proposition 2, the procedure then checks whether the property being investigated is violated in which case false is returned and the entire algorithm terminates. At the end of the algorithm (line 18), true is returned in case the property was never violated.

```
begin
    $SCC \leftarrow ComputeSCC(Layer)$
    if $\Phi \equiv \mathbf{AG}\,\mathbf{EF}\,\phi$ then
        forall $scc \in SCC$ do
            if $isTerminal(scc) \wedge \forall s \in scc : \neg\phi(s)$ then
                return $false$
            end
        end
    end
    if $\Phi \equiv \mathbf{AG}\,\mathbf{AF}\,\phi$ then
        forall $scc \in SCC$ do
            $V \leftarrow scc \setminus \{s \in scc \mid \phi(s)\}$
            if $hasCycle(V)$ then
                return $false$
            end
        end
    end
    return $true$
end
```

Algorithm 2: Checking strongly connected components of current layer

We have not specified the details of the isTerminal and hasCycle procedures. The isTerminal procedure can be implemented by checking that all successors of nodes in the *scc* are contained in the *scc*. The hasCycle procedure can be implemented by, e.g., a depth-first search of the nodes in V.

The completeness of the basic sweep-line algorithm and Proposition 1 ensures that all strongly connected components will eventually have been computed and inspected in Algorithm 2. Furthermore, Algorithm 2 is a direct implementation of the two properties stated in Proposition 2. We therefore have the following theorem concerning the correctness of our algorithm:

Theorem 1. *Let $\mathcal{P} = (O, \sqsubseteq, \psi)$ be a monotonic progress measure, and let $\Phi \equiv \mathbf{AG}\,\mathbf{EF}\,\phi$ or $\Phi \equiv \mathbf{AG}\,\mathbf{AF}\,\phi$. Then Algorithm 1 terminates and Φ is satisfied if and only if the algorithm returns true.*

In Algorithm 2 we have separated the computation of *SCC* from the checking of the *SCC*. As an optimisation it is possible to integrate the checking of the properties of a *scc* into the *scc* computation algorithm. This could make it possible to check the *SCC* as they are encountered by the *scc*-algorithm. As a further optimisation it is also possible to compute the *SCC* as the layer is being explored and not at the end of exploring a layer. However, for reason of clarity, we have decided to separate the two steps in the formulation of the algorithm.

As the continuation of the work presented in [17], we have implemented Algorithm 1 using the Standard ML language, and integrated it into CPN Tools. This allows us not only to analyse states spaces of models constructed using CPN Tools taking advantage of the sweep-line method, but also to verify the aforementioned behavioural properties. We have also optimised the algorithm, so every time a property is violated or we know that it cannot be further satisfied, the execution stops to save time.

4 The CPN MQTT Model

Our aim is to use the property-specific sweep-line model checking algorithms for CTL from the previous section to verify the key behavioural properties of the CPN model we have developed of the MQTT protocol [19].

MQTT applies topic-based filtering of messages with a topic being part of each published message. An MQTT client can subscribe to a topic to receive messages, publish on a topic, and clients can subscribe to as many topics as they are interested in. As described in [18], an MQTT client can operate as a publisher or as a subscriber, and we use the term client to generally refer to a publisher or a subscriber. The broker [18] is the core of any publish/subscribe protocol and is responsible for keeping track of subscriptions, receiving and filtering messages, deciding to which clients they will be dispatched, and sending them to all subscribed clients. The MQTT protocol delivers application messages according to the three Quality of Service (QoS) levels defined in [2], which are motivated by the typically needs that IoT applications may have in terms of reliable delivery of messages.

4.1 Interaction Overview

MQTT defines five main operations: connect, subscribe, publish, unsubscribe and disconnect. Such operations, except the connect which must be performed a priori by each of the clients who want to participate in the communication, are mutually independent and can be triggered in parallel by the clients and processed by the broker. We have developed the CPN model following modelling patterns that ensure modularity, and thereby encapsulation of both the protocol logic and the behaviour of such operations.

In order to show how the clients and the broker interact, we describe the different actions that clients may carry out by considering an example. Figure 1 shows a sequence diagram for a scenario where two clients connect, perform subscribe, publish and unsubscribe, and finally disconnect from the broker. The protocol interaction is as follows:

1. Client 1 and Client 2 request a connection to the Broker.
2. The Broker sends back a connection acknowledgement (CONNACK) to confirm the establishment of the connection.
3. Client 2 subscribes to topic 1 with a QoS level 1, and the Broker confirms the subscription with a subscribe acknowledgement message.
4. Client 1 publishes on topic 1 with a QoS level 1. The Broker responds with a corresponding publish acknowledgement (PUBACK).
5. The Broker transmits the publish message to Client 2 which is subscribed to the topic.

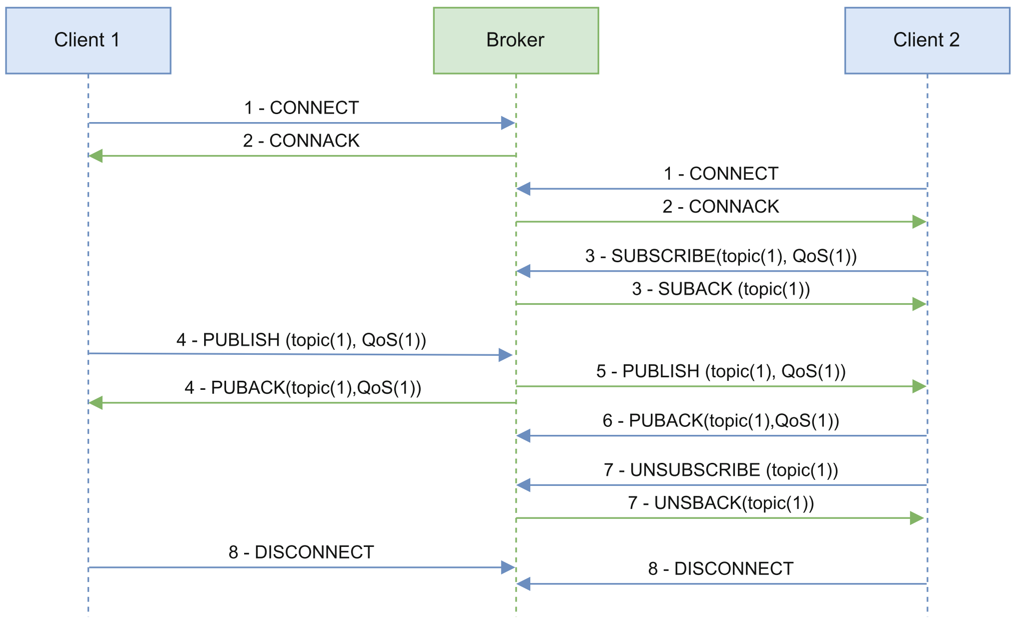

Client 1
Broker
Client 2
1 - CONNECT
2 - CONNACK
1 - CONNECT
2 - CONNACK
3 - SUBSCRIBE(topic(1), QoS(1))
3 - SUBACK (topic(1))
4 - PUBLISH (topic(1), QoS(1))
5 - PUBLISH (topic(1), QoS(1))
4 - PUBACK(topic(1),QoS(1))
6 - PUBACK(topic(1),QoS(1))
7 - UNSUBSCRIBE (topic(1))
7 - UNSBACK(topic(1))
8 - DISCONNECT
8 - DISCONNECT

Fig. 1. Message sequence diagram illustrating the MQTT phases.

6. Client 2 gets the published message, and sends a publish acknowledgement back as a confirmation to the Broker that it has received the message.
7. Client 2 unsubscribes to topic 1, and the Broker responds with an unsubscribe acknowledgement.
8. Client 1 and Client 2 disconnect.

4.2 CPN Model Overview

We now briefly show and discuss the model and its main elements that are important for the understanding of the work carried out. We refer the reader to [19] for a detailed description of the MQTT protocol and the MQTT CPN model. The complete CPN model of the MQTT protocol consists of twenty four modules organised into six hierarchical levels.

The model is organised following a modelling pattern that ensures modularity and therefore, encapsulation of the protocol logic and behaviour of such operations. This offers advantages both for readability and understandability of the model and also, for making it easier to detect and fix errors during the incremental verification. For instance, this has allowed us to make a clear separation of the different QoS functional logic without having any negative complexity impact on the model. Note that the verification is incremental in the sense that we start with a core functionality of the protocol, and then we incrementally add more operations until we have the complete functionality included. This implies that we incrementally verify properties associated to each set of the operations.

Figure 2 shows the top-level module of the CPN MQTT model which consists of two *substitution transitions* (drawn as rectangles with double-lined borders) representing the Clients and the Broker roles of MQTT. Substitution transitions constitute the basic syntactical structuring mechanism of CPNs and

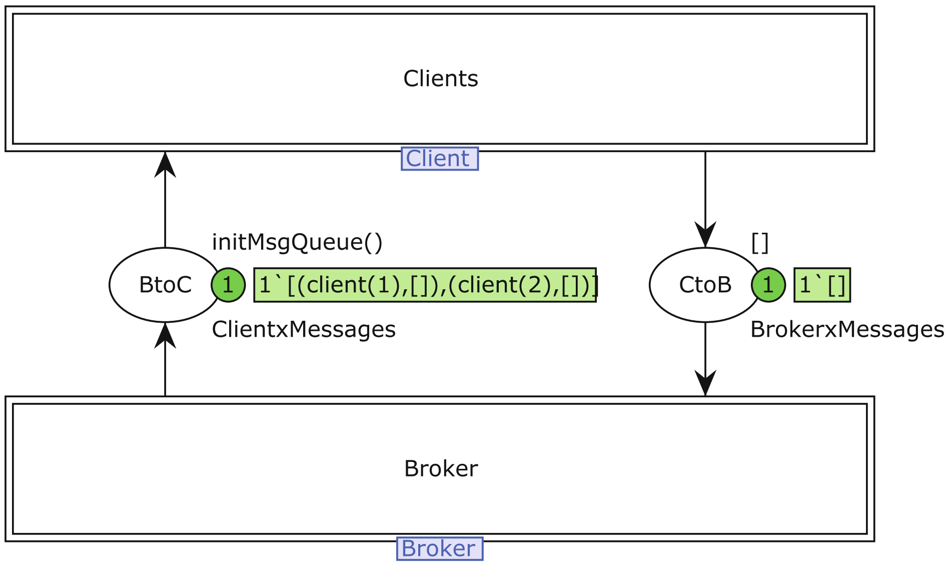


Fig. 2. The top-level module of the MQTT CPN model.

each of the substitution transitions has an associated *module* that models the detailed behaviour of the clients and the broker, respectively. The name of the (sub)module associated with a substitution transition is written in the rectangular tag positioned next to the transition.

The two substitution transitions in Fig. 2 are connected via directed arcs to the two places CtoB and BtoC. The clients and the broker interact by producing and consuming tokens on the places. The places CtoB and BtoC are designed to behave as queues. The queue mechanism offers some advantages that the MQTT specification implicitly indicates. The purpose of this is to ensure the ordered message distribution as assumed from the transport service on top of which MQTT operates.

4.3 Client and Broker State Modelling

The colour sets defined for modelling the client state are shown in Fig. 3. The ClientProcessing submodule in Fig. 4 models all the operations that a client can carry out. Clients can behave as senders and receivers, and the five substitution transitions `CONNECT`, `PUBLISH`, `SUBSCRIBE`, `UNSUBSCRIBE` and `DISCONNECT` have been constructed to capture both behaviours.

The place Clients (top-left place in Fig. 4) uses a token for each client to store its respective state during the communication. The `State` colour set is an enumeration type containing the values `READY` (for the initial state), `WAIT` (when the client is waiting to be connected), `CON` (when the client is connected), and `DISC` (for when the client has disconnected). The states of the clients are represented by the `ClientxState` colour set which is a product of `Client` and `ClientState`. The colour set `ClientState` is used to represent the state of a client and consists of a list of `TopicxQoS`, a `State`, and a `PID`. Using this, a client stores the topics it is subscribed to, and the quality of service level of

```
colset State = with READY | DISC | CON | WAIT;

colset TopicxQoS     = product Topic * QoS;
colset ListTopicxQoS = list TopicxQoS;

colset Client = index client with 1..C;
colset ClientState  = record topics : ListTopicxQoS *
state  : State *
pid    : PID;

colset ClientxState = product Client * ClientState;
```

Fig. 3. Colour set definitions used for modelling client state.

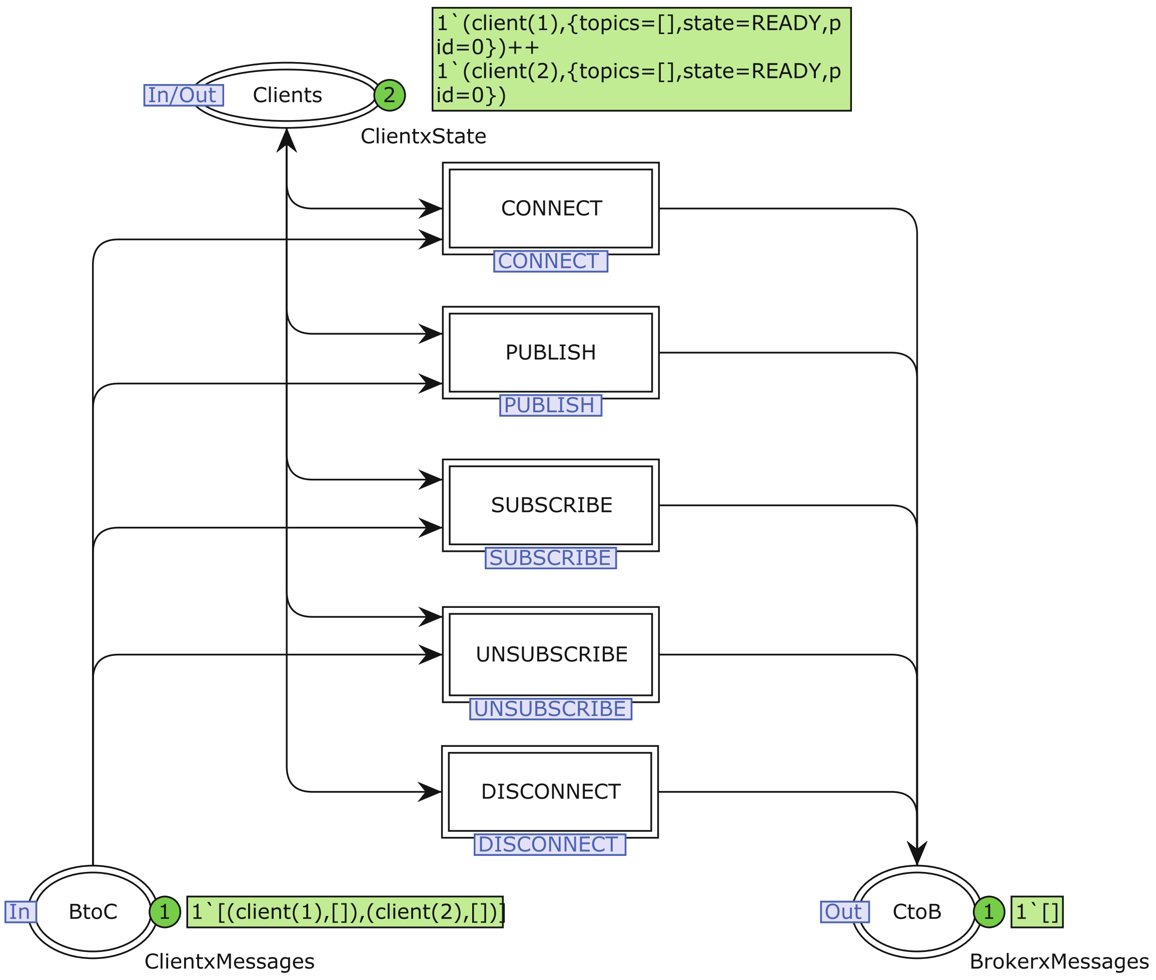


Fig. 4. ClientProcessing submodule.

each subscription. The colour set PID is used for modelling the packet identifiers which play a central role in the MQTT protocol logic.

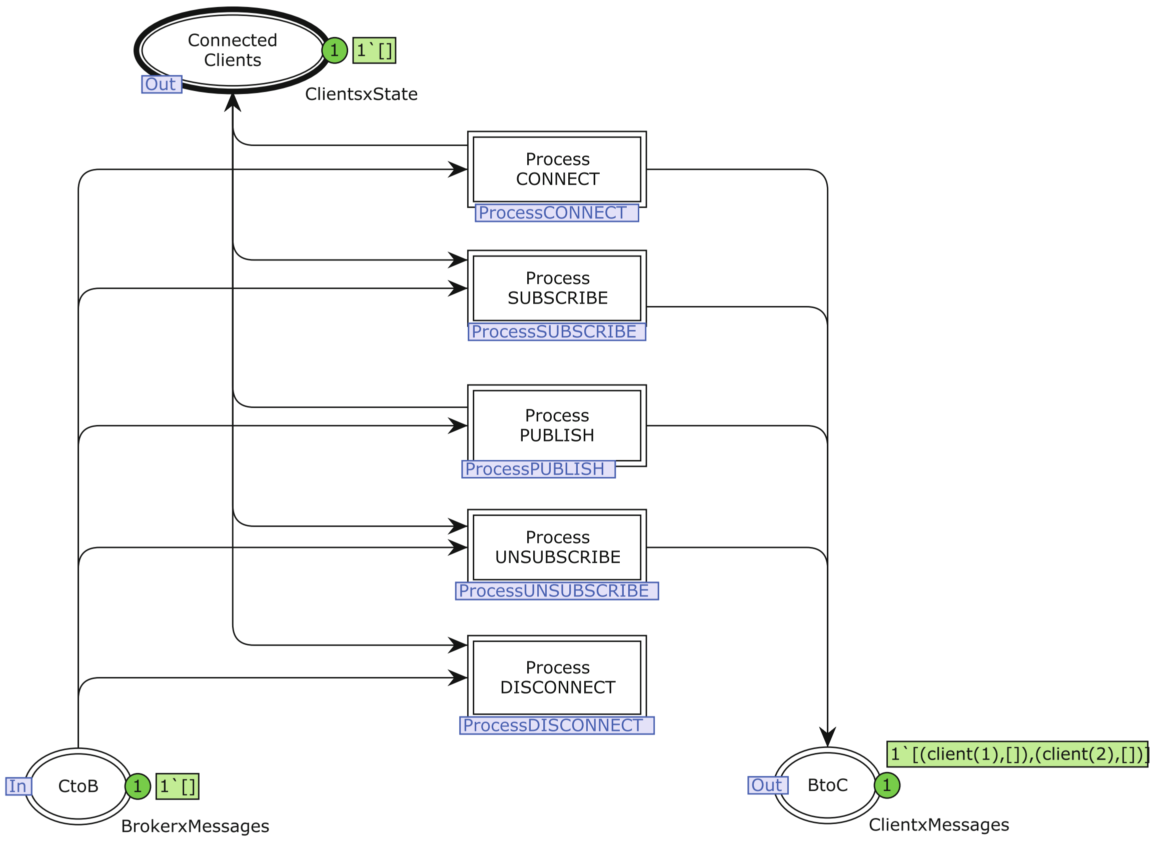


Fig. 5. The BrokerProcessing module.

We have structured the broker similarly as we have done for clients. This can be seen from Fig. 5 which shows the BrokerProcessing submodule. The ConnectedClients place keeps the information of all clients as perceived by the broker. This place is designed as a central storage, and it is used by the broker to distribute the messages over the network. The broker behaviour is different from that of the clients, since it will have to manage all the requests and generate responses for several clients at the same time.

5 Model Checking and Experimental Results

In this section we show how we have performed sweep-line based model checking of the CPN MQTT model and present the results from the experiments.

5.1 Progress Measure

The first aspect to consider is how to define the progress measure of the model. Since the model runs in an acyclic configuration there is a final state where all the clients are disconnected and we take advantage of the `PID` as a way to keep track of the evolution of the message interchange. We have therefore defined the progress measure as a combination of the different states the clients can go through in conjunction with the `PIDs`. In the experiments, we consider two clients, so the initial state is made up of two clients in the READY state and PID = 0 and the final state is reached when both clients are in a DISC state and the PID = 3.

Our definition of this progress measure over the possible combinations splits our state space into 100 layers. We have also experimented with other progress measures specifications, for instance, just taking into account the states or only the PIDs which for each such separated choice produces a total of 16 layers. In our experience, there is a trade-off between the granularity and the size of each layer, and it is up to the analyst to decide depending on the concrete resources. Since the progress measure is defined such that the progress values are integers, we have for the states assigned 1 for READY, 2 for WAIT, 3 for CON and 4 for DISC, and 1 for PID = 0, 2 for PID = 1, 3 for PID = 2 and 4 for PID = 4. It is important to note that the clients cannot backtrack to a previous state nor to a lower PID. For instance, if client 1 reaches the CON state, it can never be again in the WAIT state. As we need to keep a global notion of progress, we compute it using the following equation with c_1 and c_2 being client 1 and client 2, respectively and where B is a base:

$$\psi_{\mathrm{c}} = B^3 * \mathit{state}(c_1) + B^2 * \mathit{pid}(c_1) + B^1 * \mathit{state}(c_2) + B^0 * \mathit{pid}(c_2)$$

Essentially, we interpret the states and the PIDs of the two clients as a number where B is required to be larger than the number of states of each client. In our experiments, we have used $B = 10$, i.e., the decimal numbering system. With this, we can obtain a progress value for each possibility (between 1111 and 4444) and respecting the monotonic ordering of non-regress.

As we have implemented the model in a modular and parameterized fashion, we are able to control several elements, for instance, the number of clients, the operations those clients can perform (e.g., connect and subscribe), and the size of the queues for handling messages. Note that, in order to obtain a finite state space, we have to limit the number of clients and topics, and also bound the packet identifiers. The packet identifiers are incremented throughout the execution of the different phases of the protocol, i.e., the connect, subscribe, data exchange, unsubscribe, and disconnect phases. This means that we cannot use a single global bound on the packet identifiers as a client could reach this bound, e.g., already during the publish phase and hence the global bound would prevent (block) a subsequent unsubscribe to take place. We therefore introduce a local upper bound on packet identifiers for each phase. This local bound expresses that the given phase may use packet identifiers up to this local bound. In the next subsection, we present the results of, first, running the state space using the sweep-line algorithm, and second, verifying certain behavioural properties.

5.2 Incremental Verification and Properties

We have designed a system to run six incremental executions which gives us more control to detect errors during the validation of the model and the verification of the properties. The six different scenarios are wrapped within three different steps. In the first step we include only the parts related to clients connecting and disconnecting. In the second step we add subscribe and unsubscribe, and finally in the third step we add data exchange considering the three quality of service levels in turn. At each step, we include verification of additional properties. Below we briefly discuss the three steps and the properties verified at each step. Note that properties that reason about clients are verified for each individual client. In other words, the properties make sure that every client involved satisfies the property being verified.

Step 1. Connect and Disconnect. In this first step we consider only the part of the model related to clients connecting and disconnecting to the broker.

S1-P1-ConsistentConnect. The clients and the broker have a consistent view of the connection state.
S1-P2-ClientsCanConnect. There exists a reachable state in which each client is connected to the broker.
S1-P3-ConsistentTermination. Each terminal state (dead marking) has a consistent and desired behaviour.
S1-P4-PossibleTermination. The protocol can always be terminated, i.e., a terminal state (dead marking) can always be reached.

Step 2. Subscribe and Unsubscribe. In this step, we add the ability for the clients to subscribe and unsubscribe (in addition to connect/disconnect from step 1).

S2-P1-CanSubscribe. There exists states in which both the clients and the broker sides consider each client to be subscribed.
S2-P2-ConsistentSubscription. In every state there is a consistent subscription in both clients and broker sides.
S2-P3-PossiblySubscribed. If the client sends a subscribe message, then eventually both the clients and the broker sides will consider the client to be subscribed.
S2-P4-CanUnsubscribe. For each client there exists executions in which the client sends an unsubscribe message.
S2-P5-EventuallyUnsubscribed. If the client sends an unsubscribe message, then eventually that both the clients and the broker sides consider the client to be unsubscribed.

Step 3. Publish and QoS levels. We add the ability for the clients to publish and receive messages in addition to the rest of the properties of Steps 1 and 2.

S3-P1-PublishConnect. Each client can publish if it is in a connected state.
S3-P2-CanPublish. There exists an execution in which each client publishes a message.
S3-P3-CanReceive. For each client there exists an execution in which each client receives a message.
S3-P4-ReceiveSubscribed. A client only receives data if it is subscribed to the topic, i.e., the client side considers the client to be subscribed.

Table 1 shows the representation of the properties in CTL. Note that the verified properties have the forms described in Sect. 3. We have marked in Table 1 some properties with "*". The property S2-P3 has been computed as if it were an *EF* property (the same applies to S2-P5). However, this does not completely verify the property since it only checks that it is possible to find a state where the client is subscribed. What we really want to check is that we can reach a state where the client sends a subscribe message, and eventually after that the client is subscribed in the broker side. The implementation of such properties of the form $AG(\Phi \Rightarrow AF(\Psi))$ is part of our future work.

5.3 Experimental Results

Table 2 summarises the statistics as a result of running the six scenarios, using both approaches, the traditional CPN state space exploration and the sweep-line method approach, and verifying the properties aforementioned. The States and Arcs columns give the number of states and edges, respectively, in the state space. The Peak column lists the peak number of states stored in memory (i.e., the number of states in the largest layer). The Rel. Mem. Reduction column indicates the reduction of memory as the result of using the sweep-line method, compared to the total number of states (stored in memory by the tradition approach). For instance, in row number 5 in Table 2, we have a reduction in memory consumed of 84.17%, which means that the number of states we have in memory corresponds to the 15.83% of the total amount of states we would store using the traditional approach. The TV-Time column amounts the time that took for the traditional procedure to verify the properties. The SLV-Time column details the time needed to verify the properties using the sweep-line approach. Finally, the column Rel. Time Increment gives the relative additional

Table 1. CTL properties verified.

Property	CTL formula	Description
S1-P1	AGΦ	Φ: Consistent connection
S1-P2	EFΦ	Φ: Each client is connected to the broker
S1-P3	AG(¬ DM ∨ Φ)	DM: Dead marking | Φ: desired dead marking
S1-P4	AGEF DM	DM: Dead marking (checked in S1-P3 that it is desired)
S2-P1	EFΦ	Φ: Each client can subscribe
S2-P2	AGΦ	Φ: Each client is consistently subscribed
S2-P3*	EFΦ	Explanation above
S2-P4	EFΦ	Φ: Each client can unsubscribe
S2-P5*	EFΦ	Explanation above
S3-P1	AG (Φ ⇒ Ψ)	Φ: Client connected | Ψ: Client can publish
S3-P2	EFΦ	Φ: Each client can send a publish
S3-P3	EFΦ	Φ: Each client can receive a publish
S3-P4	AG (Φ ⇒ Ψ)	Φ: Client receives a publish | Ψ: Client is subscribed

Table 2. Results on the six incremental executions using both approaches.

Configuration	States	Arcs	Peak	Rel. Mem. Reduction	TV-Time	SLV-Time	Rel. Time Increment
1. Conn-Disconn	35	48	9	74.29%	0.00 s	0.00 s	0%
2. 1 + Subscribe	507	1,054	180	64.50%	0.156 s	0.219 s	79%
3. 2 + Unsubscribe	1,849	4,120	300	83.78%	1.328 s	2.171 s	63.48%
4. 3 + Pub QoS 0	4,282	8,840	711	83.4%	4.453 s	4.983 s	11.9%
5. 3 + Pub QoS 1	11,462	23,934	1,815	84.17%	20.172 s	28.531 s	41.44%
6. 3 + Pub QoS 2	43,791	85,682	7,037	83.93%	168.113 s	250.708 s	49.13%

time that was necessary for the sweep-line method to proceed, compared to the traditional approach.

The two approaches provided the same results during the evaluation of the properties, keeping the consistency of the verification process. Even though the sweep-line is more time consuming, the memory usage was successfully reduced even in the worst case scenario. The highest relative time consumption is located in the third row with an increase of 63.48%. However, this should not be taken completely as reference since the calculation with such a low number of states and arcs is very sensitive to also the time that takes to compute the state space and the SCC.

6 Conclusions and Future Work

We have presented the application of the sweep-line method for verifying an elaborate set of behavioral properties of the MQTT protocol. The application of the sweep-line method relied on a set of on-the-fly algorithms for model checking selected CTL behavioral properties. We have compared the application of the sweep-line method with the application of standard CTL model checking in CPN Tools demonstrating a substantial reduction in memory usage at the expense of a modest increase in execution time. The consistency between the results obtained using conventional CTL model checking and the results obtained with the implementation of our property-specific CTL model checking algorithms for the sweep-line method serves as a validation of our new approach.

We see several possible directions for future work based on the results and experiments presented in this paper. We plan to investigate a more complete set of scenarios where different configurations are considered. This includes the number of clients, different progress measures, distinct queue sizes, and the possibility of retransmitting packets. This is going to be relevant to make other analysis and study, first, how the number and size of the strongly connected components affects the sweep-line method and second, how the reduction factor grows with the value of the parameter. Related to this, there are also several possibilities for improving the implementation of the property-specific CTL model checking algorithms that we employ.

CTL model checking with the sweep-line method has until now been an open research problem, and the algorithms presented represents a first step towards addressing this. The extension of our approach to cover a larger subset of CTL properties is an important direction of future work. An example is the *S2-P3-EventualSubscribed* property discussed in Sect. 5. Properties on this form can be explored in a two-steps fashion way, where first the property in the left-hand side of the implication is accomplished, and then a second instance of the state space is explored, checking whether the property in the right-hand side is satisfied or not. The work presented in [16] on using tailored model checking algorithms for different CTL properties could serve as a starting point. A key challenge is to identity a subset of CTL compatible with the least-progress-first exploration order of the sweep-line method. In the context of symbolic model checking using binary-decision diagrams (BDDs), forward CTL model checking algorithms have been developed [11]. However, the sweep-line method is not compatible with the use of BDDs. The reason is that deleting states from a BDD (as required by the sweep-line method) may cause the memory usage for storing the BDD to increase. This counteracts the idea of how the sweep-line method alleviates the state explosion problem.

A more open direction of future work is to develop CTL model checking techniques that can be used for non-monotonic progress measures - and not only monotonic progress measures as presented in this paper. We see potential improvements in being capable of including non-monotonic progress measures. It would significantly expand the class of models that can be analysed, for instance, we could also run the algorithm in the cyclic version of the CPN MQTT model.

References

1. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press, Cambridge (2008)
2. Banks, A., Gupta, R.: MQTT Version 3.1.1. OASIS Stand. **29**, 89 (2014). http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html
3. Cheng, A., Christensen, S., Mortensen, K.H.: Model checking coloured petri nets - exploiting strongly connected components. DAIMI Rep. Ser. **26**, 519 (1997)
4. Christensen, S., Kristensen, L.M., Mailund, T.: A sweep-line method for state space exploration. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 450–464. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45319-9_31
5. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982). https://doi.org/10.1007/BFb0025774
6. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. (TOPLAS) **8**(2), 244–263 (1986)
7. Clarke, E.M., Grumberg, O., Minea, M., Peled, D.: State space reduction using partial order techniques. Int. J. Softw. Tools Technol. Transf. **2**(3), 279–287 (1999)

8. Clarke, E.M., Klieber, W., Nováček, M., Zuliani, P.: Model checking and the state explosion problem. In: Meyer, B., Nordio, M. (eds.) LASER 2011. LNCS, vol. 7682, pp. 1–30. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35746-6_1
9. CPN tools. http://cpntools.org/
10. Evangelista, S., Kristensen, L.M.: Hybrid on-the-fly LTL model checking with the sweep-line method. In: Haddad, S., Pomello, L. (eds.) PETRI NETS 2012. LNCS, vol. 7347, pp. 248–267. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31131-4_14
11. Iwashita, H., Nakata, T., Hirose, F.: CTL model checking based on forward state traversal. In: Proceedings of International Conference on Computer Aided Design, pp. 82–87. IEEE Computer Society (1996)
12. Jensen, K., Kristensen, L., Mailund, T.: The sweep-line state space exploration method. Theor. Comput. Sci. **429**, 169–179 (2012)
13. Jensen, K., Kristensen, L.M., Wells, L.: Coloured petri nets and CPN tools for modelling and validation of concurrent systems. Int. J. Softw. Tools Technol. Transf. **9**(3), 213–254 (2007)
14. Kristensen, L.M., Mailund, T.: A generalised sweep-line method for safety properties. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 549–567. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45614-7_31
15. Kristensen, L.M., Christensen, S.: Implementing coloured petri nets using a functional programming language. Higher-order Symbolic Comput. **17**(3), 207–243 (2004)
16. Liebke, T., Wolf, K.: Taking some burden off an explicit CTL model checker. In: Donatelli, S., Haar, S. (eds.) PETRI NETS 2019. LNCS, vol. 11522, pp. 321–341. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21571-2_18
17. Lilleskare, A., Kristensen, L.M., Høyland, S.-O.: CTL model checking with the sweep-line state space exploration method. In: Proceedings of Norwegian Informatics Conference (NIK) (2017)
18. MQTT essentials part 3: Client, broker and connection establishment. https://www.hivemq.com/blog/mqtt-essentials-part2-publish-subscribe
19. Rodríguez, A., Kristensen, L.M., Rutle, A.: Formal modelling and incremental verification of the MQTT IoT protocol. In: Koutny, M., Pomello, L., Kristensen, L.M. (eds.) Transactions on Petri Nets and Other Models of Concurrency XIV. LNCS, vol. 11790, pp. 126–145. Springer, Heidelberg (2019). https://doi.org/10.1007/978-3-662-60651-3_5
20. Rodriguez, A., Kristensen, L.M., Rutle, A.: On CTL model checking of the MQTT IoT protocol using the sweep-line method. In: Petri Nets and Software Engineering. International Workshop, PNSE 19, Aachen, Germany, June 24, 2019, volume 2424 of CEUR Workshop Proceedings, pp. 57–72 (2019)
21. Stern, U., Dill, D.L.: Improved probabilistic verification by hash compaction. In: Camurati, P.E., Eveking, H. (eds.) CHARME 1995. LNCS, vol. 987, pp. 206–224. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60385-9_13
22. Valmari, A.: The state explosion problem. In: Advanced Course on Petri Nets, pp. 429–528. Springer (1996)
23. Van Leeuwen, J., Leeuwen, J.: Handbook of Theoretical Computer Science, vol. 1. Mit Press, Elsevier (1990)
24. Vardi, M.Y.: Branching vs. Linear time: final showdown. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 1–22. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45319-9_1

PAPER C

EXECUTING MULTILEVEL DOMAIN-SPECIFIC MODELS IN MAUDE

Alejandro Rodríguez, Francisco Durán, Adrian Rutle, Lars Michael Kristensen

In Journal of Object Technology Volume 18, number 2 (July 2019), pages 4:1-21, 2019.

Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Executing Multilevel Domain-Specific Models in Maude

Alejandro Rodríguez[a] Francisco Durán[b] Adrian Rutle[a]
Lars Michael Kristensen[a]

a. Western Norway University of Applied Sciences, Bergen, Norway

b. Universidad de Málaga, Málaga, Spain

Abstract Multilevel modelling (MLM) tackles the limitation in the number of abstraction levels present in traditional modelling approaches within the model-driven software engineering (MDSE) field. One way to specify the behaviour description of MLMs is by means of multilevel model transformations. In this paper, we propose an approach to achieve reusability and flexibility in specifying and executing multilevel model transformations. For this purpose, we rely on code-generation and the efficient rewriting logic mechanisms that Maude provides. As a proof of concept, we have developed an infrastructure which combines our MLM tool MultEcore, that facilitates definition of MLM hierarchies and transformations, with Maude, which performs the execution of the transformations on these hierarchies.

Keywords Multilevel modelling; Model transformations; Rewriting logic

1 Introduction

MDSE tackles the increasing complexity of software by utilizing abstractions and modelling techniques, and treats models as first-class entities in all phases of software development. MDSE has proven to be a successful approach in terms of gaining quality and efficiency [WHR14, MGS+13]. Most traditional MDSE approaches are based on the Object Management Group (OMG) 4-layer architecture, such as the Eclipse Modelling Framework (EMF) [SBMP08] and the Unified Modelling Language (UML) [UML]. These approaches follow a two-level hierarchy in which only two levels of abstraction are available for the modeller; i.e., models and their instances. Compelling to use these two-level (meta)modelling approaches may introduce several challenges, for instance, convolution and an increase in the complexity of models [LGC14, LG18]. It also has a direct impact in the specification of Domain-Specific Modelling Languages (DSML), since the domain expert might be forced to fit several abstraction layers into the only two levels which are supported by the traditional approaches [AK08]. Furthermore, capturing all the concepts in the same level makes it more difficult to define the metamodel and to fix the potential inconsistencies created in the artefacts conforming to (or depending on) this metamodel.

MLM has proven to be a successful approach in areas such as software architecture and enterprise/process modelling domains [LGC14, AK17, AKdL18]. Having a hierarchical organization of the metamodels defined to precisely capture the desired environment facilitates the possible extensions and modifications that might come in the future, not only in the existing levels, but also for adding/removing levels. MLM provides separation of concerns and therefore prevent the pollution of models where specialization of concepts would be done in the same level. This also leads to a better modularization and facilitates extendibility. Being able to add new metalevels makes extensions/modifications independent on other models. Further benefits of MLM and a detailed comparison between MLM and two-level traditional approaches can be found in [LG18].

Understanding the behaviour of a model is key to comprehend the behaviour of the underlying system that is being abstracted. In MDSE, model transformations are one of the possible means to specify behaviour. Although there are several approaches proposed for the definition and simulation of behavioural models based on reusable model transformations (e.g., [dLV02, Ren03, RDV09]), these rely on traditional two-level modelling hierarchies. Furthermore, modelling the behaviour through multilevel model transformations [AGM15] and performing execution in MLM has not been widely explored yet. Multilevel Coupled Model Transformations (MCMTs) have already been proposed [MRS$^+$18b, MWR$^+$19] to achieve reusable multilevel model transformations for the definition of behaviour. In this paper, we have improved the MCMTs by making them more reusable and flexible, extended them with the notion of cardinality, and implemented a first prototype for the execution of the rules.

In this paper, we propose an infrastructure for the execution of MLM hierarchies. This infrastructure is built on top of previous work for specification of structure and behaviour of MLM hierarchies in MultEcore [MRS16, MWR$^+$19, MRS$^+$18b]. MultEcore is a set of Eclipse plugins aimed to combine the best from traditional two-level modelling – the mature tool ecosystem (integration with EMF) and familiarity – with the flexibility of MLM. It supports the main features that characterize MLM such as potency, multiple typing and unlimited level of abstractions.

We rely on Maude for the execution/simulation of MLM hierarchies [CDE$^+$07]. Maude is a high-level language and a high-performance interpreter and compiler in the OBJ algebraic specification family [GM13]. It supports rewriting logic and programming of systems. Among the functionalities that Maude provides, we exploit the ability to specify object-based systems which allows us to transform both the multilevel hierarchy and the MCMTs from MultEcore to Maude. This transformation provides the complete Maude specification (a rewrite logic theory) that can be directly executed by the rewriting logic engine. Execution in Maude means to apply the rewrite rules that gives the next states of our model. Ultimately, we can conduct reachability analysis (by means of strategies [EMOMV07]) and model checking. Maude supports model checking on the generated state space as it implements a Linear Temporal Logic (LTL) [BK08] model checker.

Paper outline: We present the prototype infrastructure in Sect. 2. Section 3 introduces the MLM background and the running example which we use for the rest of the paper. In Sect. 4 we describe how MCMTs work and display the rules we define for the MLM hierarchy example presented in Sect. 3. Section 5 discusses how Maude can be used to execute the MLM hierarchy and how the translation between MultEcore and Maude has been achieved. In Section 6 we discuss related work. Finally, Section 7 concludes the paper and outlines directions for future work.

2 The infrastructure

In this section, we present the overall architecture (see Fig. 1) of the infrastructure which we have developed for the execution of MLM hierarchies. The left-hand side of Fig. 1 shows the MultEcore part, where we can specify the MCMT rules (top), the multilevel hierarchy (middle) and the possible specification of behavioural properties that we want to check or enforce during the execution. In [MRS+18b] the so-called supplementary hierarchies are used to define property specification languages like Linear Temporal Logic (LTL) and to specify behavioural properties. We can directly translate these properties to Maude since it implements an LTL model checker. The Transformer: MultEcore $\leftrightarrow$ Maude takes care of the automatic transformation. This can be viewed as a bidirectional transformation [Ste07, CFH+09] between the model spaces in MultEcore and Maude:

MultEcore $\rightarrow$ Maude: once the modeller decides which specific language is going to be simulated, the transformer takes both the models that define the language (the concrete hierarchy branch) and the multilevel model transformation rules, and creates the Maude specification. Such a specification corresponds to a functional Maude file that can be executed directly.

Maude $\rightarrow$ MultEcore: the states that Maude provides (new versions of the model) are given by means of an XML file. This file is interpreted by the transformer which can directly propagate the new state(s) to the multilevel hierarchy in MultEcore.

The right-hand side of Fig. 1 shows the Maude perspective. Once we have generated the specification with the Transformer, we are able to execute the model using Maude's

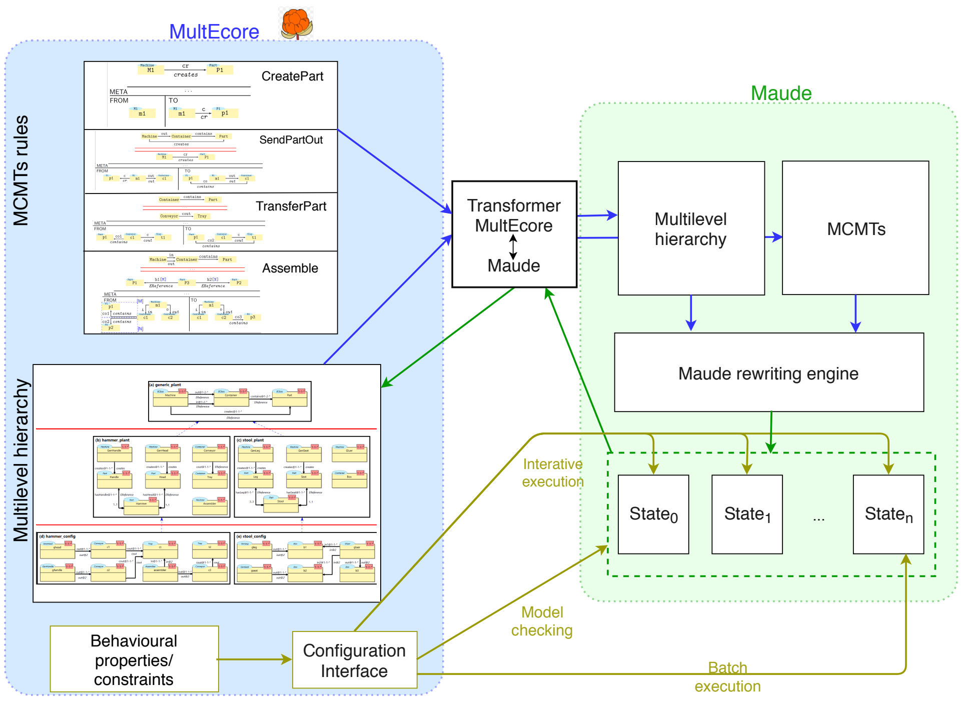


Figure 1 – Infrastructure for the execution of multilevel models

rewriting engine. As Maude allows several kinds of rewriting procedures depending on the strategy chosen, we might want to perform either an interactive execution (i.e., step-by-step, where the modeller can take control of the next states that can be given), or batch execution to directly get a final state.

In our prototype, we fully implement the capability to specify multilevel hierarchies and MCMTs, the bidirectional transformer, and the execution of the specified configurations (see [Dep] to access the infrastructure). This encompasses all the features shown in the figure except for the Configuration Interface which is aimed to offer the modeller a user-friendly interface for controlling aspects related to execution and verification.

3 DSML Structure - Multilevel Modelling

In this section, we discuss how we achieve the definition of the structural dimension of DSMLs by means of MLM. MLM is based on the idea of deep instantiation and eliminating the restriction in the number of times a model element can be instantiated. In this context, MLM techniques match well with the creation of DSMLs, especially when we focus on behavioural languages since behaviour is usually defined at the metamodel level while it is executed at least two levels below; i.e., at the instance level [dLG10, MWR+19].

Usually, when we are defining the structure of a domain-specific language, we mentally "sketch" this as a hierarchical composition. It is therefore natural to have a way to literally translate this mental representation into a model. An example of a multilevel hierarchy (originally from [RDV09]) describing a DSML for Product Line Systems (PLS) is shown in Fig. 2. This hierarchy (which is specified using MultEcore) contains three levels of abstractions (four if we include the reserved level 0 that corresponds to *Ecore* in EMF, and five if we take into account the extension we make in Sect. 4.2). Note that each model in the hierarchy is a directed multi-graph and we establish typing relations in the vertical dimension which are formalised as *graph homomorphisms* [EEPT06]. The complete formalization as well as other MLM examples and an evaluation of MultEcore are depicted in [Mac19]. Further examples of multilevel models with four or more levels can be checked out in [RDLGN15].

The example displays a hierarchical distribution with the generic_plant model at the top (Fig. 2a). In this model, the abstract concepts related to the manufacturing of objects are defined. Machine is aimed for any gear that can create, modify or combine objects, which are represented by the concept Part. Both concepts are linked by the creates relation. A Container can store parts, and this connection is captured by the relation contains. All machines may have containers where they can take parts from or where they can drop the manufactured ones. These two relations are identified with the in and out edges, respectively. The annotations in the rectangles at the right top corners of the nodes, and after the names in the arrows (separated by '@') specify the potencies. *Potency* is used on elements as a means of restricting the levels at which this element may be used to type other elements. In the case of MultEcore, a potency specification includes three values: the first two specify the first and the last levels where one can directly instantiate an element (min and max), and the third value specifies the number of times the element can be indirectly re-instantiated (depth).

The second level contains two models which are defined for two specific environments: one for creating hammers and one for manufacturing stools (hammer_plant in Fig. 2b and stool_plant in Fig. 2c, respectively). One can see that both branches share similarities. The languages (branches) must belong to the same family in order to make (horizontal) reusability possible. The hammer_plant contains the concepts related to the manufacturing of Hammers which are created by combining one Handle and one Head. This can be seen from the multiplicities 1..1 in the relations hasHandle and hasHead. These two relations have as type EReference (from Ecore [SBPM09]) since no relation is defined between parts in the top level model (generic_plant). This is because the concept of assembling parts is too specific to be located in generic_plant. At this level, we can also find the machines GenHandle and GenHead that create the parts, the Assembler, and the containers Conveyor and Tray that move and store the parts, respectively. It is due to the nature of PLSs that the stool_plant (Fig. 2c) branch in the MLM hierarchy is structured similar to the one in hammer_plant. In this case, we have machines GenLeg and GenSeat to generate Leg and Seat, respectively, and a Gluer that puts together three legs and one seat to make a Stool.

The two models defined at the bottom of the hierarchy, in Fig. 2d and Fig. 2e, represent specific configurations for hammers (hammer_config) and stools (stool_config) productions, respectively. They contain specific instances of the concepts defined in the levels above and they are used to specify concrete product lines configurations, in which parts get transferred from generator machines to machines that combine them.

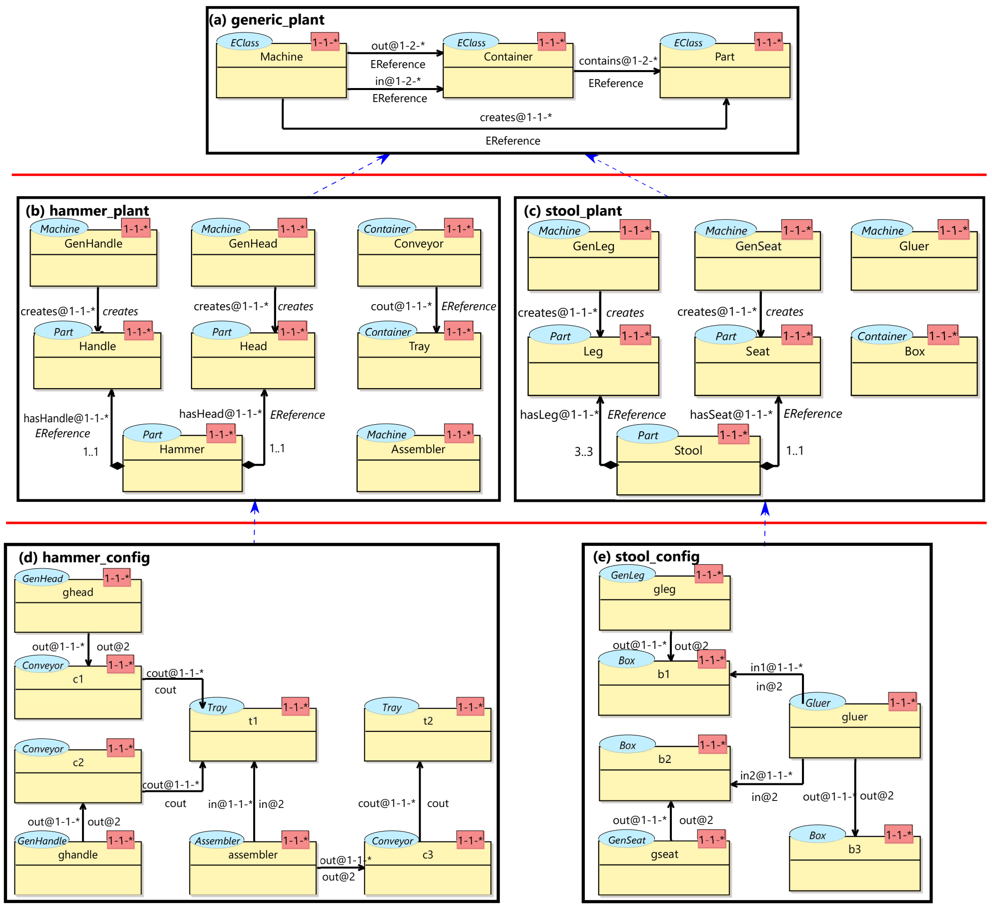


Figure 2 – Full hierarchy for the PLS case study

One could argue that this hierarchy can be managed with traditional two-level approaches using specialization and generalization (i.e., using subclassing and inheritance relations, respectively). The traditional 4-layer architecture of OMG would force us to a design with several concepts in the same model, since this architecture leaves only one level for user models. The top level M3 is reserved for MOF; M2 for metamodels, e.g., UML class diagram or UML object diagram; M1 is designated for user-models; M0 has a "representation" relation to M1, which associates elements of M1 to real world objects, i.e., there is no "instance-of" relationship to the M1 level above. Hence, we would fit the levels generic_plant, hammer_plant and hammer_config into one model at M1 level. Furthermore, the typing relations between model elements in these different levels would have to be maintained manually, i.e., we would need elements like *MachineInstance* and *MachineType*, *ContainerInstance* and *ContainerType*, etc.

MLM provides the flexibility needed to avoid the use of anti-patterns (e.g., type-object pattern is described in [LGC14, LG18]) when fitting several layers of abstractions into one single level. This anti-pattern appears when both the concept and the metaconcept are defined in the same level, leading to convolution. Since the focus and the contribution of this paper is oriented to the flexible definition of the behaviour and the execution/simulation of the models, we do not enter into details of all the concepts related to the definition and construction of MLM hierarchies; we refer to [dLGC15, AK18, Küh18a, Küh18b, MWR+19] for the details.

4 DSML behaviour - MCMTs

Transformation rules can be used to represent actions that may happen in the system. Conventional in-place model transformations (MTs) are rule-based modifications of a source model (specified in the left-hand side of the rule) resulting in a new state of the model (determined by the right-hand side). While the left-hand side takes as input (a part of) a model and it can be understood as the pattern we want to find in our original model, the right-hand side describes the target state of the system we want to acquire in our model. There is a match when what we specify in the left-hand side is found in our source model. The behaviour is the implicit transition from the left-hand side to the right-hand side.

MCMTs have been proposed as a mean to overcome the issues of both the traditional two-level transformation rules and the multilevel model transformations [MWR+19]. While the former lacks the ability to capture generalities, the later is too loose to be precise enough (case distinctions). In this section we show how the behaviour of a multilevel DSML can be described by using MCMTs.

4.1 PLS behaviour definition

The actions illustrated in this section describe a possible behaviour in the PLS environment. These actions detail how to create parts, move them through the different machines and assemble them into new parts. A rule *CreatePart* can be specified as shown in Fig. 3. It represents the process in which a machine creates a part. The META block allows us to locate types in any level of the hierarchy that can be used in FROM and TO blocks.

However, the actual power of the META comes from the fact that it facilitates the definition of an entire multilevel pattern. The rule *CreatePart* is sufficient to generate instances of Head and Handle for the hammer branch of Fig. 2 and instances of Seat and Leg for the stool branch of Fig. 2. The variable P1 matches to any of the aforementioned parts, both in hammer_plant and stool_plant models, and the variable M1 matches any of the creator machines: GenHead, GenHandle, GenSeat or GenLeg. However, the key feature is that this rule can only match the generators of parts, since we require M1 in the META block to have a creates relation to P1. Then a correct match of the rule comes when an element, coupled together with its type, fits an instance of M1 that has a relation of type creates to an instance of P1. For example, GenHead in Fig. 2b, fits M1, since GenHead has a creates relation to Head. Hence, m1 can be matched to ghead (defined at the left in Fig. 2d) when applying the rule, in order to create a new part (p1), which would be an instance of Head.

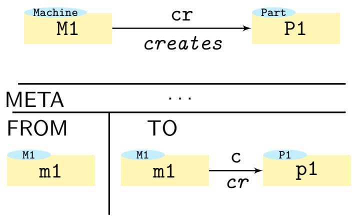


Figure 3 – Rule *CreatePart*: The execution gives a state where a machine has created a part

Compared to the original idea of MCMTs [MWR+19] (the levels specified in a rule had to be consecutive by default) we have removed the strictness in the levels to provide a more flexible definition. There might be several levels in between the blocks FROM/TO and the upper level. This is represented by the three dots in Fig. 3.

Another rule called *SendPartOut* shown in Fig. 4 is the action defined for moving a created part from its generator into the output container. It shows two levels specified in the META block (separated by the upper double line). Similarly as in the *CreatePart* rule, the three dots in between the specified meta levels enhance the flexibility of the rule that can be applied in several cases without modifying it (this will be shown later in this section). Also, it leads to a more natural way of defining that a type is defined at some level above, without the need of saying explicitly in which level. At the top level, we mirror part of generic_plant, defining elements like out and contains, that are used directly as types in the FROM and TO blocks. These elements are defined as constants, meaning that the name of the pattern element must match an element with the same name in the typing chain. The use of constants allows us to be more restrictive when matching, and significantly reduces the amount of matches that we obtain. On the other hand, we allow the type on the variables to be *transitive* (i.e., indirect typing). For instance P1, which has the variable Part for the type, will match any node which indirectly has Part as type, or ultimately will match to Part if no indirect one is found. Fig. 5 displays *TransferPart* rule which moves a part from a Conveyor to a Tray. It models the action where c1 (of type Conveyor), that holds a part p1 and which is connected to t1 with Tray as type (described in the FROM block), moves such a part to t1 (specified in the FROM block).

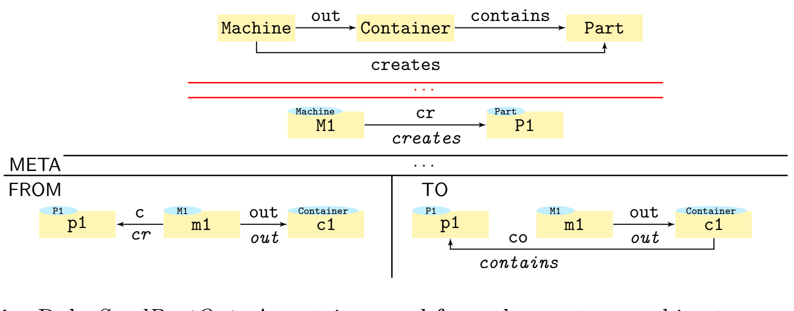


Figure 4 – Rule *SendPartOut*: A part is moved from the creator machine to a container

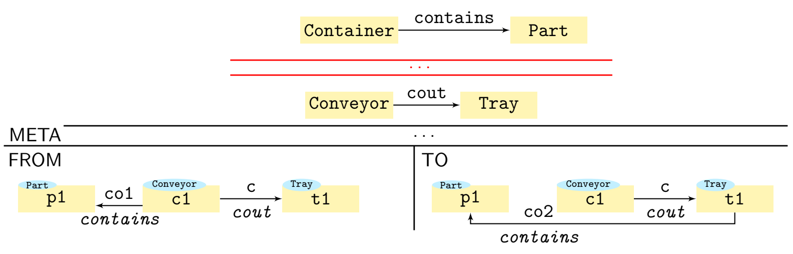


Figure 5 – Rule *TransferPart*: The execution provides a model state where a part is transferred from a conveyor to a tray

The *Assemble* rule creates new products by combining the component parts. It assembles two parts into a different part (see Fig 6). It requires, for the resulting part p3, to consist of, or be built from parts p1 and p2. Having three variables for the different parts, allows us to make an explicit distinction between them even though all of them are instances of Part. Variables [M] and [N] in the intermediate level on the h1 and h2 relations, represent the cardinality that have to be matched in order to apply the rule. A part consisting of other parts might also need a specific number of instances to be built from. In Fig. 2b we can see that a Hammer is composed of 1 Handle and 1 Head (this in fact can be understood as the default case). However, in Fig. 2c, a Stool needs 3 Legs in order to be assembled. As the multiplicity has been explicitly specified in the second META level of the rule, and we have established those same variables for the multiplicities in the FROM block, then the rule will take that into consideration during the matching process. When this process takes action, M and N will be bound to 3 and 1 for stools, respectively. Then, these numbers will be used to check whether that amount of parts exist in the FROM block. For the match to succeed, three legs and one seat (and the respective relations with the container), need to be found. Thus, this is syntactic sugar to represent that in the model it is necessary to explicitly find this number of instances for the match to occur. The way we define and use these multiplicities is inspired by the concept of *cardinality* described in [SCGdL11]. Fig. 7 shows the unfolded version of the *Assemble* rule. As M and N have been bound to 3 and 1, respectively, the pattern shown in the figure needs to be found for a successful match.

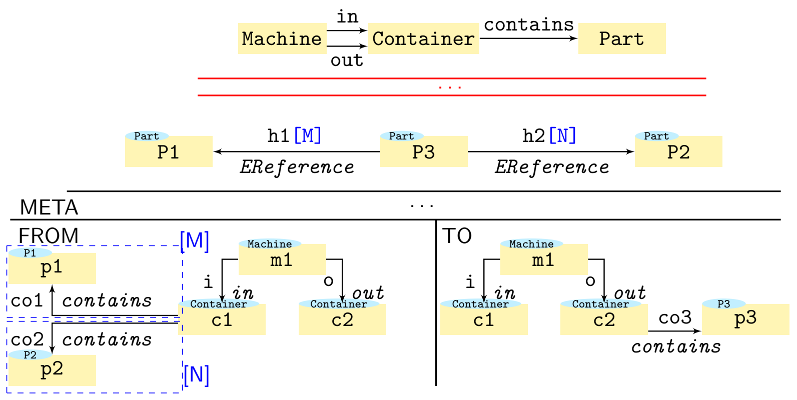


Figure 6 – Rule *Assemble*: The execution gives a state where a machine takes several parts and assemble them in a new one

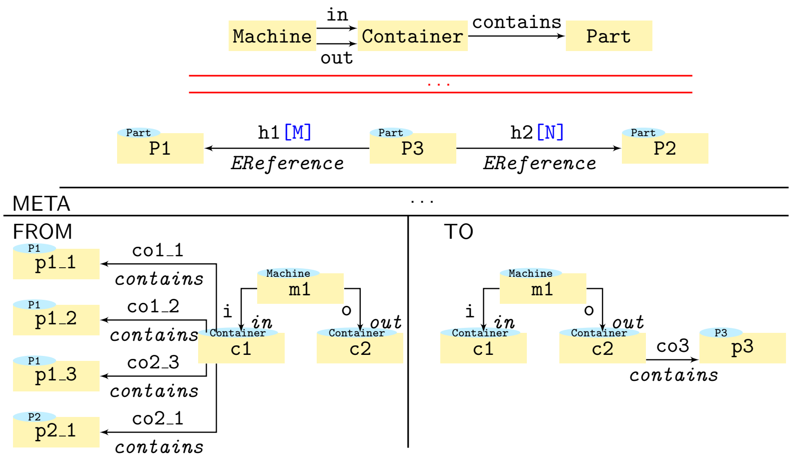


Figure 7 – Unfolded version of *Assemble* rule. The match takes into account the multiplicities specified, and searches for three p1 and one p2

4.2 Horizontal and vertical flexibility

In the previous section we have defined the model transformation rules that provide the behaviour to the multilevel hierarchy. Horizontal flexibility is indirectly inferred since these rules can be directly applied to both branches shown in Fig. 2. For example, *CreatePart* rule can be applied to create either a Head or a Handle (for hammer branch) or to create a Leg or a Seat (for stool branch).

In this section, we demonstrate how MCMT rules are still applicable when modifying an existing multilevel hierarchy (vertical flexibility) and how we can make restrictions in the rules to confine the typing flexibility.

Let us suppose that ACME factories have some specific type of hammers that are created by a handle and a green head. This can be introduced as a new level in between Fig. 2b and Fig. 2d, that captures the ability to create green heads, called special_head_config. This new level is depicted in Fig. 8. The two nodes, SpecialGenHead and GreenHead and the edge creates are now instances of GenHead, GreenHead and creates, respectively, which are defined in the level hammer_plant (Fig. 2b).

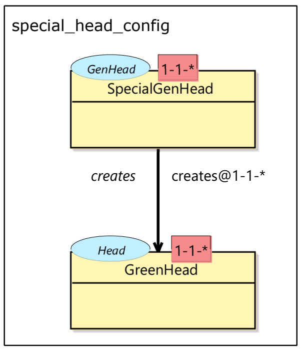


Figure 8 – New specified level for creating green heads

We are now able both to define a generator for regular heads and also a generator for green heads, in the level shown in Fig. 2d. As this depends on the concrete scenario, we might construct different configurations which can include any combination of the two generator of heads aforementioned, and the rules should be agnostic to those possibilities. Fig. 9 shows the two possible matches depending on the machine we define at the instantiation level (i.e., at the lowest level). At the left side of the dashed double vertical line we can see the *CreatePart* rule, already shown in Fig. 3.

At the right side we show a hierarchy consisting of three levels (divided by horizontal lines). These levels comprise those elements of the PLS hierarchy involved in the generation of heads (i.e., a *generator* that *creates* a *head*). They represent

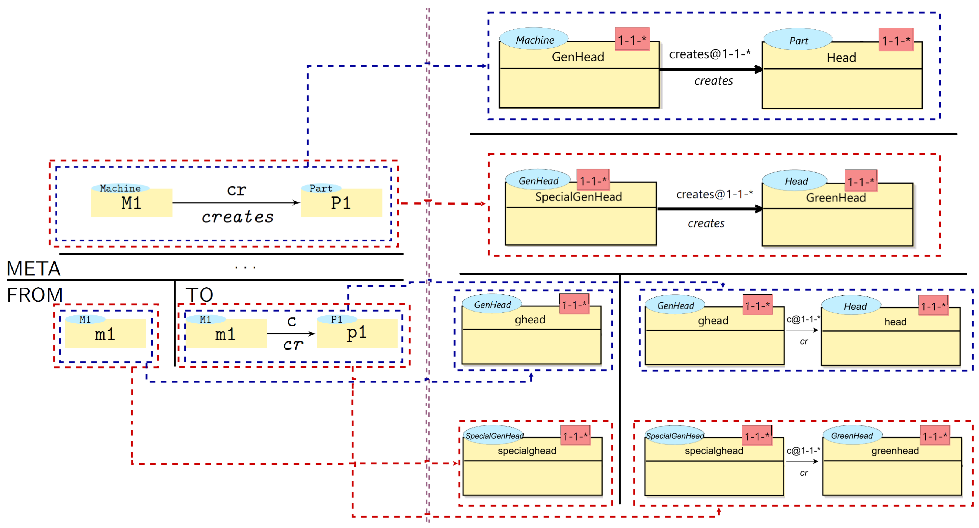


Figure 9 – Vertical reusability of rule *CreatePart*

hammer_plant (Fig. 2b), *special_head_config* (Fig. 8) and *hammer_config* (Fig. 2d) levels, respectively. Note that the lowest level shown in this hierarchy is divided by a vertical black line, which represents the same logic as the FROM/TO pattern. This level is composed by two instances, one represents the match for the creation of a regular head (at the top) and the other corresponds to the match of a specified generator of green heads (at the bottom).

The dashed blue lines represent the match of the rule in case we define our configuration as using regular head generator (ghead), while the dashed red lines represent the match of the rule for a scenario where we have a green head generator (specialghead). Moreover, the right hand side of the hierarchy at the instantiation level (bottom-right side of Fig. 9) shows the state where the *CreatePart* rule has been fired. As one can observe, the rule has not been modified at all, but the flexibility provided allows both matchings depending on the scenario specified.

The default flexibility opens for several possible matchings. For instance, a normal head could be created by a special head generator. Another possibility is, in the *Assemble* rule, that a hammer can be manufactured from a green head and a normal handle. Since considering these matches as valid is up to the modeller, we provide functionality for allowing/disallowing them. We can restrict the *CreatePart* rule using a matching strategy where the nearest type is selected (specialization priority) and still leave open the matches for *Assemble*.

One might consider the need of restricting the indirect typing which is allowed by-default since this flexible assumption (*the type can be found at any number of jumps of any length*) might not be desired in all situations. To disallow that, we can use $t@n$ $(n \mid n \in \mathbb{N})$ over a type t. First, this disables the indirect typing (so we must find the type in just one jump upwards) and second, it forces the type to be at n levels above the one where the match for t has been found.

5 Formal specification and execution with Maude

As explained in Section 1, the MultEcore tool allows us to define multilevel hierarchies and MCMTs to describe their behavior. MultEcore relies on Maude for the simulation and formal analysis of the specified MLM systems.

Maude [CDE+07] is a specification language based on rewriting logic [Mes92], a logic of change that can naturally deal with states and non deterministic concurrent computations. A rewrite logic theory is a tuple $(\Sigma;E;R)$, where Σ is called signature and specifies the type structure (sorts, subsorts, etc.) and E is the collection of equations and memberships declared in the functional module. Therefore, $(\Sigma;E)$ is an equational theory that specifies the system states as elements of the initial algebra $\mathcal{T}_{(\Sigma;\mathrm{E})}$, and R is a set of rewrite rules that describe the one-step possible concurrent transitions in the system. Rewrite specifications thus described are executable, since they satisfy some restrictions such as termination and confluence of the equational subspecfication and coherence of equations and rules. Indeed, Maude provides support for rewriting modulo associativity, commutativity and identity, which perfectly captures the evolution of models made up of objects linked by references as in graph grammar. In summary, Maude provides, among others, the next useful features [CDE+02]:

Formal specification. The Maude specification of multilevel hierarchies and MCMTs represents a formal semantics in rewriting logic. Since these specifications are executable, they can be used for simulating/executing our models. The automatic bidirectional transformation MultEcore $\leftrightarrow$ Maude allows the execution of MLM models from the MultEcore tool. Indeed, Maude's flexibility and customization capabilities have allowed us to represent MLM models and MCMTs in Maude using a syntax very similar to the MultEcore syntax. This has led to a straightforward transformation between MultEcore and Maude.

Execution of the specification. The Maude specification obtained from MLM hierarchies and corresponding MCMTs using the above transformation are executable, and therefore can be used to simulate them in Maude. The versatile rewriting engine provides a lot of functionalities to customize the way we go trough the execution steps. As we will see below, we can simulate our systems letting Maude choose the path to follow, or we can specify a concrete path by means of execution strategies.

Formal environment. Once the rewriting logic specification of the MLM hierarchies and their MCMTs is available in Maude, we can use the formal tools in its formal environment to analyze the systems thus described. For example, we can check properties as confluence or termination of our specifications, but also perform reachability analysis, model checking or theorem proving on them.

5.1 Multilevel hierarchies in Maude

In the Maude language, object-oriented systems can be specified by object-oriented modules in which classes and subclasses are declared, with the usual support for inheritance, dynamic binding, etc. A class is declared with syntax class $C \mid a_1$: S_1,. . . , a_n: S_n, where C is the name of the class, a_i are attribute identifiers, and S_i are the sorts of the corresponding attributes. The objects of a class C are record-like structures of the form $< O : C \mid a_1$: v_1, . . . , a_n: $v_n >$, where O is the identifier of the object and v_i are the current values of its attributes.

```
1  class Model | name : Name, om : Name, elts : Configuration, rels : Configuration .
2  class Element | name : Oid, type : Oid .
3  class Node | .
4  class Relation | source : Oid, target : Oid, min-mult : Nat, max-mult : Nat* .
5  subclasses Node Relation < Element .
```

Figure 10 – Maude structure of a multilevel hierarchy

In a concurrent object-oriented system, the concurrent state, which is called a *configuration*, has the structure of a multiset made up of objects and messages that evolves by concurrent rewriting using rules that describe the effects of the communication events of objects and messages. The system presented in this paper evolves as the result of applying the rewrite rules on collections of objects.

A multilevel hierarchy is represented in Maude as a structure of sort System of the form $MLM\{model_1\ model_2\ \ldots model_n\}$, where MLM is the name of the multilevel hierarchy and each $model_i$ is an object of class Model that represents a model in the hierarchy. Note that when the transformation from MultEcore to Maude is to be performed, the modeller have to decide the branch that is it going to be executed. For instance, for the PLS hierarchy used in this paper, the modeller would decide between hammer or stool branch. Fig. 10 illustrates the specification of a multilevel hierarchy in Maude. A model is represented as an object of class Model, which has attributes representing its name, its ontological metamodel om, and the nodes (elts) and relations (rels) that are part of it (line 1). As mentioned in the previous paragraph, Configuration is a predefined sort in Maude implemented to deal with object-based systems. Instances of classes Node (line 3) and Relation (line 4) represent, respectively, nodes and relations. Both are subclasses (line 5) of a class Element (line 2) of elements with a name and a type. In addition to the attributes inherited from Element, class Relation has attributes for the source and target of a relation, and its multiplicity range (min-mult, max-mult).

The sort Name allows us to define how our objects are going to be identified. For instance, the identifier of a model is represented as $level(x)$, for x either 0 or a natural number. Level 0 is always reserved for *Ecore* and n the lowest level in the hierarchy. Identifiers are required to be unique. The transformation assigns these names automatically when generated. As we will see in the next section, objects generated in transformation rules are also given unique fresh names.

Given these declarations, Fig. 11 illustrates how models are represented. Specifically, it shows the Maude term that represents the generic_plant level (corresponding to Fig. 2a). Note that this is just one of the models in the MLM hierarchy. In this case it has assigned level(1) (Line 1). Then we have the name of the model ("generic-plant", Line 2) and its metamodel at the level right above represented by om : "Ecore" (Line 3). It contains two sets, elts (Line 4) and rels (Line 8), for capturing the nodes and the relations, respectively. For instance, the relation specified in Line 11, with identifier oid(1,5), represents the *in* relation between a machine and a container: its name is id(1,"in"), its type is id(0, "EReference"), and it links two nodes, id(1, "Machine") as source (in Line 5) and id(1, "Container") as target (in Line 6). As expected, all the information available in MultEcore is encoded in the Maude representation.

5.2 The MCMT rules in Maude

In Maude, a distributed system is axiomatized by an equational theory describing its states as an algebraic data type and a collection of conditional rewrite rules specifying its *behaviour.* Rewrite rules are written crl $[l]$: t => t' if C, with l the rule label, t and t' terms, and C a guard or condition. Rules describe the local, concurrent transitions that are possible in the system, i.e., when a part of the system state fits the pattern t, then it can be replaced by the corresponding instantiation of t'. The guard C acts as a blocking precondition: a conditional rule can only be fired if its condition is satisfied. Rules may be given without label or condition.

We can directly translate MCMT rules to conditional rewrite rules in Maude. We illustrate this representation of rules with the *CreatePart* rule in Fig. 3. Fig. 12 shows its Maude counterpart. The left-hand side of the rule (Lines 2-17) encodes the META and FROM blocks of the rule. In this case, in the META section there is one model level(L), and in the FROM one model level(J). Notice that we are not specifying a concrete level, but we use variables that will be bound when the rewriting engine matches the rule to our MLM concrete hierarchy. It is in the conditions where we can constraint the behaviour of the rule by defining predicates or conditional expressions, such as L < J. The counter specified in Line 16 is an auxiliary object that keeps a counter so that we can create fresh new identifiers for the elements created in the right-hand side. The rest of the left-hand side is fairly straightforward as it can directly be inferred from the *CreatePart* rule displayed in Fig. 3). Atts..., Elts..., O..., etc. are just variables we define to capture those attributes we do not explicitly specify.

The right-hand side of the rule (Lines 18-29) shows how the objects in the left-hand side will be modified when the rule is applied; the ellipses are only to save space. Model level(L) (the META) is left unmodified. We display in detail the level(J) model, which corresponds to the TO block shown in Fig. 3. As we can see in Lines 22 and 24-25, there are two new elements. The first one corresponds to the new part, which will have identifier oid(J, s N), name id(J, s s s N), and type P1 (note the reference to the correspondent object in model level(L)). Notice that new identifiers and names are generated by using the counter object. The s operator is a Maude predefined operator that calculates the successor of a number. For instance, if J and N get bound to 3 and 100, we would get as results oid(3, 101) and id(3,103), respectively. A new relation is also created, with source m1 (Line 12) and target the new part id(J, s s s N).

In addition to the condition on models level(L) and level(J), other conditions are also

```
< level(1) : Model |
    name : "generic-plant",
    om   : "Ecore",
    elts : (
      < oid(1,1): Node | name : id(1, "Machine"), type : id(0, "EClass") >
      < oid(1,2): Node | name : id(1, "Container"),type : id(0, "EClass") >
      < oid(1,3): Node | name : id(1, "Part"), type : id(0, "EClass") >),
    rels : (
      < oid(1,4): Relation | name : id(1, "out"), type : id(0, "EReference"),
          source : id(1, "Machine"), target : id(1, "Container") >
      < oid(1,5): Relation | name : id(1, "in"), type : id(0, "EReference"),
          source : id(1, "Machine"), target : id(1, "Container") >
      < oid(1,6): Relation | name : id(1,"contains"), type : id(0,"EReference"),
          source : id(1, "Container"), target : id(1, "Part") >
      < oid(1,7): Relation | name : id(1,"creates"), type : id(0,"EReference"),
          source : id(1, "Machine"), target : id(1, "Part") >) >
```

Figure 11 – Maude specification for generic_plant model

given as a conjunction of predicates. In this case, * references of variables are handled by the predicate *. This predicate is necessary to provide a type with the *transitive* dimension mentioned in Sect. 4.1. We call it * to be consistent with the original idea (degree of genericness $*t$) presented in [MRS+18b]. Multiplicities, potencies, and other facilities in the rules are handled similarly.

5.3 Execution and results

Given MLM hierarchies and MCMT rules specified in Maude as shown in the previous section, we have several options for executing it. Given an initial ground MLM hierarchy instantiation from which to start the execution, the Maude rewrite commands can attempt the consecutive application of the rules in our specification. Maude provides two different rewriting commands, for which we can specify a maximum number of rewriting steps to take, implementing two different strategies: *rewrite* follows a top-down rule-fair strategy and *frewrite* follows a depth-first position-fair strategy.

In addition, Maude also provides commands for the controlled execution of our rules. Maude facilitates a rich strategy language with which we can specify our own strategies. For example, we can perform a batch execution, just by specifying step by step, which rules are to be applied, and, if desired, the objects on which it should happen, by providing a partial substitution for the instantiation.

Let us show a very simple example of the use of the *srewrite* command (abbreviated *srew*), which allows us to apply a concrete strategy to a given term (our initial state will

```
crl [CreatePart] :
   { < level(L) : Model |
          name : M,
          elts : (< O01 : Node  | name : M1, type : *Machine, A01 >
                  < O02 : Node  | name : P1, type : *Part, A02 >
                  Elts),
          rels : (< O03 : Relation | name : cr, type : *creates, source : M1, target : P1,A03>
                  Rels),
          Atts >
    < level(J) : Model |
          name : M',
          elts : (< O04 : Node | name : m1, type : *M1, A04 >
                  Elts'),
          rels : Rels',
          Atts' >
      < counter : Counter | value : N >
      Conf }
 => { < level(L) : Model | ... >
      < level(J) : Model |
          name : M',
          elts : (< O04 : Node | name : m1, type : *M1, A04 >
                  < oid(J, s N) : Node | name : id(J, s s s N), type : P1 >
                  Elts'),
          rels : (< oid(J, N) : Relation | name : id(J, s s N), type : cr,
                    source : m1, target : id(J, s s s N), min-mult : 1, max-mult : 1 >
                  Rels'),
          Atts' >
      < counter : Counter | value : s s s s N >
      Conf }
 if L < J
 /\ *(*M1, level(sd(J,1)), M1, ...)
 /\ *(*Machine, level(sd(L,1)), id(1, "Machine"), ...)
 /\ *(*Part, level(sd(L,1)), id(1, "Part"), ...)
 /\ *(*creates, level(sd(L,1)), id(1, "creates"), ...) .
```

Figure 12 – Maude representation of the *CreatePart* rule (note the ellipses)

be the Maude representation of either hammer_config (Fig. 2d) or stool_config (Fig. 2e)) where we specify the rules (and optionally some constraints within them) ordering. Let us assume that we want to make a complete iteration over the hammer_config model (for a smoother explanation, in this scenario we do not consider having the specialghead as instance of SpecialGenHead displayed in Fig. 8 and described in Sect. 4). We would need then to create a head, a handle, and eventually we would get assembled a new hammer. Fig. 13 shows the defined strategy we can execute to test if such a final state is reached.

```
1  srew PLS using CreatePart ;
2                 CreatePart ;
3                 SendPartOut ;
4                 SendPartOut ;
5                 TransferPart ;
6                 TransferPart ;
7                 Assemble ;
8                 TransferPart .
```

Figure 13 – Strategy for manufacturing a Hammer in hammer_config configuration

PLS in Line 1 corresponds to the initial term (the complete hierarchy). As the strategy is written, each application of the CreatePart rule (Lines 1 and 2) can create either a Handle or a Head. This is not a problem as the rewriting engine provides all the solution, and then it discards the non valid ones when applying the Assemble rule (a solution right before the execution of Assemble might have produced 2 Handles). However, we can constrain the applications of the CreatePart rule by explicitly providing a partial substitution:

```
CreatePart[P1 <- id(2, "Handle")] ;
CreatePart[P1 <- id(2, "Head")] ;
```

With this, we are binding P1 to generate first a Handle, then a Head. In both cases, we would end up having the same solution.

The rest of the rules are applied sequentially. Taking the model in Fig. 2d as reference, we would create a handle and a head, and move them to the conveyors c1 and c2, respectively, using the rule *SendPartOut*. Then we move both parts to the tray t1 with the rule *TransferPart* and the parts are assembled into a hammer using the rule *Assemble*. Finally, the hammer is moved from the conveyor c3 to the tray t2 using again the rule *TransferPart*. Once we get a solution model by running the rules, Maude generates an XML file which is transformed back into MultEcore. Fig. 14 shows how the solution would look in the graphical view of MultEcore.

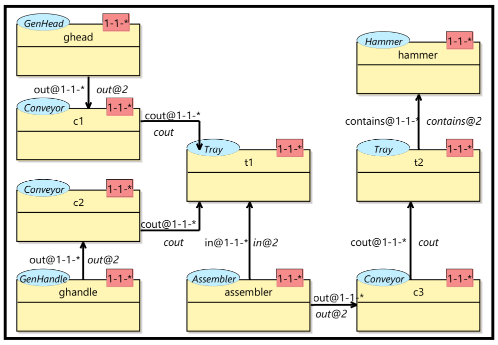


Figure 14 – hammer_config with a Hammer

6 Related work

There exist several tools and technologies that support dynamic execution of models through model transformation in a graphical manner. A Tool for Multi-Paradigm Modeling (AToMPM) [SVM+13], the Foundational UML (fUML) [Sub11] and the Executable Meta-Object Facility (xMOF) [MLWK13], are some examples that grant such capability. AToMPM is an open-source framework for designing DSML environments, performing model transformations, manipulating and managing models which runs entirely over the web. The fUML is an executable subset of UML that can be used to define, in an operational style, the structural and behavioural semantics of systems. However, due to its exclusive focus on UML, it cannot be applied to arbitrary domain-specific languages. The Action Language for Foundational UML (ALF) [Sei14], which is built on top of fUML, provides functionality for executing UML models in a textual way. They both are intended to work together, resulting in a more complete framework that provides both graphic and programmatic (when a high degree of details is required) facets. The xMOF is a metamodelling language that integrates Ecore with the behavioural part of fUML. It is aimed at developing executable DSMLs that can be simulated using the fUML virtual machine. In [BEK$^+$06], the authors present an approach for the definition of in-place transformations in EMF. All the approaches mentioned above are based on traditional two-level approaches which disallow the multilevel capabilities presented in this paper.

ConceptBase [JGJ$^+$95] is a tool that implements the object model of a Datalog-based variant of Telos [MBJK90]. It supports subtyping chains and unrestricted class-instance relationships. However, it does not make a clear organization of elements in hierarchical models. Furthermore, it does not support key features of MLM as flexible depth (supported by our approach via *Potency* concept). The MOMENT-QVT tool [BCR06] is a model transformation engine that provides partial support for the QVT relations language [RVA06]. QVT (Query/View/Transformation) is a standard set of languages for model transformation defined by the OMG. In [AGT12], authors present an approach to transform from a multilevel setting to a two-level configuration (and the other way around) using the ATL Transformation Language (ATL) [JABK08], which is not designed to work within a multilevel context. However, our approach makes it possible to directly define the behaviour of our multilevel hierarchy (by using MCMTs). Since we can directly translate and use MLM hierarchies in Maude, a transformation to a two-level setting to be able to rely on a model transformation engine (like ATL or QVT [Kur07]) is not necessary.

In [RGdLV08], the authors show how Maude is used to represent a subset of the PLS example used in this paper. The subset corresponds to the left-hand branch of the multilevel hierarchy shown in Fig. 2. In their work, they encode both the PLS metamodel and an instance of it, to later be able to simulate it and perform formal analysis and model checking. Changes in either the metamodel or the model would need to be done manually in the Maude implementation. Our approach hides the Maude implementation so the user can make modifications directly in the graphical editor which are in turn translated automatically to Maude.

7 Conclusions and future work

In this paper, we have described how flexible and reusable model transformations (by means of the MCMTs) can be applied in the context of MLM. In addition to a theoretical foundation, we have developed a prototype of an infrastructure to connect our MLM tool MultEcore with Maude in order to execute/simulate the constructed models. We have showcased the flexibility and reusability of the MCMT rules, first, in the vertical aspect by adding an extra level into an MLM hierarchy, and second, in the horizontal aspect by using the same rules for two branches of the hierarchy. Furthermore, two important new features have been successfully applied. First, the default restriction forcing the levels in the rules to be consecutive has been lifted, providing vertical flexibility. Second, multiplicities are now supported in the MCMTs, enriching the syntax and enhancing the reusability of the rules.

We see several directions for future work. The infrastructure that connects MultEcore with Maude has been constructed as a proof of concept, and we are working on considerable extensions. To generalise from the examples in this paper, we will design an experiment in which we pick several mainstream behavioural models, refactor them to MLM hierarchies using [LG18], adapt them to MultEcore using the rearchitecter tool presented in [MRS18a], and then execute them using the presented infrastructure.

We have developed a first version of the multiplicities (cardinalities) in MCMTs. However, we intend to further extend this feature for more complex cases with potential nested definitions. We plan also to provide MultEcore functionalities to control the Maude execution directly from the editor. Moreover, we want to give the user the control to make executions customizable so that step-by-step or batch simulations might be performed. Another task is to provide the MultEcore-Maude transformation engine with more comprehension so that it becomes more fault tolerant. We currently offer the user the possibility to define both the multilevel hierarchy and the behavioural rules. We also want to work on improving the part of the infrastructure for the definition of behavioural properties to later verify them with Maude, in a user-friendly manner. Furthermore, we are currently working on better ways to specify rule orchestration and prioritization to improve the definition and application of strategies in a more generic, reusable and user-friendly way.

References

[AGM15] Colin Atkinson, Ralph Gerbig, and Noah Metzger. On the execution of deep models. In *EXE@ MoDELS*, pages 28–33, 2015. URL: `http://ceur-ws.org/Vol-1560/paper5.pdf`.

[AGT12] Colin Atkinson, Ralph Gerbig, and Christian Tunjic. Towards multi-level aware model transformations. In *International Conference on Theory and Practice of Model Transformations*, pages 208–223. Springer, 2012.

[AK08] Colin Atkinson and Thomas Kühne. Reducing accidental complexity in domain models. *Software & Systems Modeling*, 7(3):345–359, 2008.

[AK17] Colin Atkinson and Thomas Kühne. On evaluating multi-level modeling. In *MoDELS*, 2017.

[AK18] Colin Atkinson and Thomas Kühne. Deep instantiation. In *Encyclopedia of Database Systems, Second Edition.* Elsevier, 2018. `doi:10.1007/978-1-4614-8265-9\_80608`.

[AKdL18] Colin Atkinson, Thomas Kühne, and Juan de Lara. Editorial to the theme issue on multi-level modeling. *Softw. Syst. Model.*, 17(1):163–165, February 2018. `doi:10.1007/s10270-016-0565-6`.

[BCR06] Artur Boronat, José Á. Carsí, and Isidro Ramos. Algebraic specification of a model transformation engine. In *International Conference on Fundamental Approaches to Software Engineering*, pages 262–277. Springer, 2006.

[BEK$^+$06] Enrico Biermann, Karsten Ehrig, Christian Köhler, Günter Kuhns, Gabriele Taentzer, and Eduard Weiss. Graphical definition of in-place transformations in the eclipse modeling framework. In *MoDELS*, pages 425–439. Springer, 2006.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* MIT press, 2008.

[CDE$^+$02] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martı-Oliet, José Meseguer, and José F Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[CDE$^+$07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All about Maude a high-performance logical framework: how to specify, program and verify systems in rewriting logic.* Springer-Verlag, 2007.

[CFH$^+$09] Krzysztof Czarnecki, J Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *International Conference on Theory and Practice of Model Transformations*, pages 260–283. Springer, 2009.

[Dep] HVL Computer Science Department. MultEcore Maude Website. URL: `https://ict.hvl.no/multecore-maude/`.

[dLG10] Juan de Lara and Esther Guerra. Generic meta-modelling with concepts, templates and mixin layers. In *MoDELS*, pages 16–30, 2010. `doi:10.1007/978-3-642-16145-2\_2`.

[dLGC15] Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. Model-driven engineering with domain-specific meta-modelling languages. *Software and System Modeling*, 14(1):429–459, 2015. `doi:10.1007/s10270-013-0367-z`.

[dLV02] Juan de Lara and Hans Vangheluwe. Atom 3: A tool for multi-formalism and meta-modelling. In *International Conference on Fundamental Approaches to Software Engineering*, pages 174–188. Springer, 2002.

[EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation.* Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006. URL: `https://doi.org/10.1007/3-540-31188-2`, `doi:10.1007/3-540-31188-2`.

[EMOMV07] Steven Eker, Narciso Martí-Oliet, José Meseguer, and Alberto Verdejo. Deduction, strategies, and rewriting. *Electronic Notes in Theoretical Computer Science*, 174(11):3–25, 2007.

[GM13] Joseph A Goguen and Grant Malcolm. *Software Engineering with OBJ: algebraic specification in action*, volume 2. Springer Science & Business Media, 2013.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of computer programming*, 72(1-2):31–39, 2008.

[JGJ$^+$95] Matthias Jarke, Rainer Gallersdörfer, Manfred A Jeusfeld, Martin Staudt, and Stefan Eherer. Conceptbase—a deductive object base for meta data management. *Journal of Intelligent Information Systems*, 4(2):167–192, 1995.

[Küh18a] Thomas Kühne. Exploring potency. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 2–12, 2018. `doi:10.1145/3239372.3239411`.

[Küh18b] Thomas Kühne. A story of levels. In *Proceedings of MULTI Workshop: co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems*, pages 673–682, 2018. URL: `http://ceur-ws.org/Vol-2245/multi_paper_5.pdf`.

[Kur07] Ivan Kurtev. State of the art of QVT: A model transformation language standard. In *International Symp. on Applications of Graph Transformations with Industrial Relevance*, pages 377–393. Springer, 2007.

[LG18] Juan de Lara and Esther Guerra. Refactoring multi-level models. *ACM Trans. Softw. Eng. Methodol.*, 27(4):17:1–17:56, November 2018. `doi:10.1145/3280985`.

[LGC14] Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. When and how to use multilevel modelling. *ACM Trans. on Software Engineering and Methodology (TOSEM)*, 24(2):12, 2014.

[Mac19] Fernando Macías. *Multilevel modelling and domain-specific languages.* PhD dissertation, University of Oslo, Norway, 2019.

[MBJK90] John Mylopoulos, Alex Borgida, Matthias Jarke, and Manolis Koubarakis. Telos: Representing knowledge about information systems. *ACM Transactions on Information Systems (TOIS)*, 8(4):325–362, 1990.

[Mes92] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[MGS$^+$13] Parastoo Mohagheghi, Wasif Gilani, Alin Stefanescu, Miguel A. Fernández, Bjørn Nordmoen, and Mathias Fritzsche. Where does model-driven engineering help? Experiences from three industrial cases. *Software & Systems Modeling*, 12(3):619–639, 2013.

[MLWK13] Tanja Mayerhofer, Philip Langer, Manuel Wimmer, and Gerti Kappel. xMOF: Executable DSMLs based on fUML. In *SLE*, pages 56–75. Springer, 2013.

[MRS16] Fernando Macías, Adrian Rutle, and Volker Stolz. Multecore: Combining the best of fixed-level and multilevel metamodelling. In *MULTI@ MoDELS*, pages 66–75, 2016.

[MRS18a] Fernando Macías, Adrian Rutle, and Volker Stolz. A tool for the convergence of multilevel modelling approaches. In *MULTI@ MoDELS*, 2018.

[MRS$^+$18b] Fernando Macías, Adrian Rutle, Volker Stolz, Roberto Rodriguez-Echeverria, and Uwe Wolter. An approach to flexible multilevel modelling. *Enterprise Modelling and Information Systems Architectures*, 13:10:1–10:35, 2018.

[MWR$^+$19] Fernando Macías, Uwe Wolter, Adrian Rutle, Francisco Durán, and Roberto Rodriguez-Echeverria. Multilevel Coupled Model Transformations for Precise and Reusable Definition of Model Behaviour. *Journal of Logical and Algebraic Methods in Programming*, 2019. doi:10.1016/j.jlamp.2018.12.005.

[RDLGN15] Alessandro Rossini, Juan De Lara, Esther Guerra, and Nikolay Nikolov. A comparison of two-level and multi-level modelling for cloud-based applications. In *ECMFA*, pages 18–32. Springer, 2015.

[RDV09] Jose E. Rivera, Francisco Durán, and Antonio Vallecillo. A graphical approach for modeling time-dependent behavior of DSLs. In *Visual Languages and Human-Centric Computing, 2009. VL/HCC 2009. IEEE Symposium on*, pages 51–55. IEEE, 2009.

[Ren03] Arend Rensink. The groove simulator: A tool for state space generation. In *International Workshop on Applications of Graph Transformations with Industrial Relevance*, pages 479–485. Springer, 2003.

[RGdLV08] José Eduardo Rivera, Esther Guerra, Juan de Lara, and Antonio Vallecillo. Analyzing rule-based behavioral semantics of visual modeling languages with maude. In *International Conference on Software Language Engineering*, pages 54–73. Springer, 2008.

[RVA06] Sreedhar Reddy, R Venkatesh, and Zahid Ansari. A relational approach to model transformation using qvt relations. *TATA Research Development and Design Centre*, pages 1–15, 2006.

[SBMP08] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[SBPM09] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

[SCGdL11] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. Generic model transformations: Write once, reuse everywhere. In *ICMT*, pages 62–77, 2011. doi:10.1007/978-3-642-21732-6_5.

[Sei14] Ed Seidewitz. UML with meaning: executable modeling in foundational UML and the Alf action language. In *HILT*, pages 61–68. ACM, 2014.

[Ste07] Perdita Stevens. A landscape of bidirectional model transformations. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 408–424. Springer, 2007.

[Sub11] OMG Semantics Of A Foundational Subset. For executable UML models (fUML), version 1.0, 2011.

[SVM$^+$13] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Huseyin Ergin. AToMPM: A web-based modeling environment. In *MoDELS*, pages 21–25, 2013.

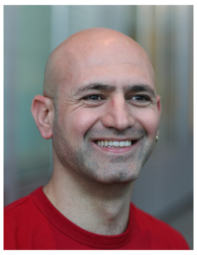
[UML] UML. http://www.uml.org/.

[WHR14] Jon Whittle, John Hutchinson, and Mark Rouncefield. The state of practice in model-driven engineering. *IEEE software*, 31(3):79–85, 2014.

About the authors

Alejandro Rodríguez is a PhD student at the Western Norway University of Applied Sciences. He is currently researching in Model-driven software engineering, multilevel modelling and coloured Petri net fields. He is part of the Software Engineering, Sensor Networks and Engineering Computing department. Contact him at `arte@hvl.no`

Francisco Durán is Full Professor at the Department of Computer Science of the University of Málaga, Spain. He received his Ph.D. degree in Computer Science from the University of Málaga in 1999, after several years as an International Fellow at SRI International, CA. He is one of the developers of the Maude system, and his research interests deal with the application of formal methods to software engineering, including topics such as cloud systems, model-driven engineering, component-based software development, open distributed programming, reflection and meta-programming, and software composition.



Adrian Rutle is Associate Professor at the Western Norway University of Applied Sciences, Norway. His research focuses on the application of theoretical results from the field of model-driven software engineering. His work has recently focused on modelling and simulation for smart robotics, MLM, patient workflows and their verification. His main expertise is the development of formal modelling frameworks for domain-specific modelling languages, graph-based logic for reasoning about static and dynamic properties of models, and the use of model transformations for the definition of semantics of modelling languages.

Lars Michael Kristensen received the PhD in computer science from University of Aarhus, and is currently professor in software engineering at Western Norway University of Applied Sciences. He has published more than 70 papers in strictly referred journal and conferences, is member of the Editorial Board of the TopNoC Springer journal, and is a member of the steering committee for the International Petri Nets conference. He is co-author of the most recent textbook on Coloured Petri Net and CPN Tools which is one of the most widely used software tools for modelling and validation of concurrent systems.

Acknowledgments Francisco Durán was partly funded by the project PGC2018-094905-B-I00 (Spanish MINECO/FEDER), and by Univ. Málaga, Andalucía Tech.

PAPER D

COMPOSITION OF MULTILEVEL DOMAIN-SPECIFIC MODELLING LANGUAGES

Alejandro Rodríguez, Fernando Macías, Francisco Durán, Adrian Rutle and Uwe Wolter

In Journal of Logical and Algebraic Methods in Programming, **Elsevier Ltd**, 2020. (Submitted)

Composition of Multilevel Domain-Specific Modelling Languages

Alejandro Rodríguez[a,*], Fernando Macías[d], Francisco Durán[c], Adrian Rutle[a], Uwe Wolter[b]

[a] *Western Norway University of Applied Sciences, Bergen, Norway*
[b] *University of Bergen, Bergen, Norway*
[c] *ITIS Software, University of Málaga, Málaga, Spain*
[d] *IMDEA Software Institute, Madrid, Spain*

Abstract

Multilevel Modelling (MLM) approaches make it possible for designers and modellers to work with an unlimited number of abstraction levels to specify their domain-specific modelling languages (DSMLs). To fully exploit MLM techniques, we need powerful model composition operators. Indeed, the composition of DSMLs is becoming increasingly relevant to the modelling community either because some DSMLs may share commonalities that we want to make reusable, or because we want to facilitate interoperability between DSMLs. In this paper, we propose a composition mechanism for structure and behaviour of multilevel modelling hierarchies. Our approach facilitates the inclusion of additional features while keeping a clear separation of concerns that enhances modularity. We provide a formal semantics of the constructions based on category theory and graph transformations and show their use in practice on a case study.

Keywords: Model-driven software engineering, Domain-specific modelling languages, Multilevel Modelling, Composition, Category theory, Graph theory, Graph transformations

1. Introduction

Multilevel Modelling is a prominent research area where models and their specifications can be organised into several levels of abstraction [1, 2]. Although there exist several approaches for MLM (see [3, 4, 5, 6] for some of them), they all share the idea of not limiting the number of levels that designers can use to specify their modelling languages. This restriction is present in traditional Model-Driven Software Engineering (MDSE) approaches which are

*Corresponding author
Email addresses: `arte@hvl.no` (Alejandro Rodríguez), `fernando.macias@imdea.org` (Fernando Macías), `duran@lcc.uma.es` (Francisco Durán), `Adrian.Rutle@hvl.no` (Adrian Rutle), `uwe.wolter@uib.no` (Uwe Wolter)

based on the Object Management Group (OMG) 4-layer architecture such as the Unified Modelling Language (UML) [7] and the Eclipse Modelling Framework (EMF) [8, 9]. Like traditional MDSE approaches, MLM uses abstractions and modelling techniques to tackle the continually increasing complexity of software by considering models as first-class entities throughout the software engineering life cycle. Despite the success of MDSE approaches in terms of quality and effectiveness gains [10], modellers can only make use of two levels of abstraction to specify their systems: one for (meta)models and one for their instances. Model designers might find this limitation too restrictive. Moreover, these limitations may lead to complications like model convolution, accidental complexity and mixing concepts belonging to different domains (see, e.g., [11, 12, 13] for discussions on this).

One of the most successful applications of MDSE is in the construction of (industrial) DSMLs [9]. DSMLs are modelling languages tailored to specific areas which are meant to be easily understood and used by domain experts. Thus, such challenges become more prevalent in the case of defining DSMLs, since variations on general purpose languages (i.e., to specify different refinements oriented to the different domains) would require further specialisations on the metamodels.

The MLM community has demonstrated that MLM is a successful approach in areas such as process modelling and software architecture domains [11, 14, 15]. Furthermore, MLM techniques are excellent for the creation of DSMLs, especially when focusing on behavioural languages, since behaviour is usually defined at the metamodel level while it is executed, at least, two levels below at the instance level [16, 17].

Although DSMLs are conceived to describe and abstract different concrete domains, we may find many similarities between existing DSMLs. In fact, the research community in software language engineering has proposed the notion of *Language Product Lines Engineering* (LPLE) with the goal of constructing software product lines where the products are languages [18]. The key aspect of their approach is the definition of *language features* that encapsulate a set of language constructs representing certain DSML functionalities. Usually, one can detect that some DSMLs share certain commonalities coming from similar modelling patterns that can be abstracted and reused across several other languages. Interoperability and reusability can therefore be achieved by advocating modularisation and composition techniques.

We have observed that several DSMLs can benefit from each other by composing them, resulting into a more complete system specification. To cope with this, we present an alternative approach to handle composition based on multiple typing which we compare with the standard way of facing composition through a *merge* operator. Traditionally, frameworks had to craft, in a tedious, ad-hoc and (usually) non reusable way, their own composition operators. Further research in this direction had raised more standard and widely accepted composition mechanisms, such as the merge operator or through direct linking among modules [18, 19]. Taking advantage of MLM and inspired by the concept of *language feature*, we present in this paper mechanisms based on our MLM approach and multiple typing to foster composition by defining the abstract syntax and the behavioural description in a modular way, i.e., by adding/removing dimensions to a selected model or a model transformation rule. We compare our construction with the merge operator and put into practice our constructs to achieve composition by applying them to a case study where we consider a multilevel DSML for processes management and a DSML that abstracts human being notions.

The rest of the paper is organised as follows. Section 2 describes our approach for Multilevel Modelling regarding structure (Section 2.1) and operational semantics (Section 2.2). Section 3 presents our composition mechanism. After motivating this mechanism in Section 3.1, we compare it to the usual merge operator and present its categorical semantics in Section 3.2. We apply in Section 4 the formal constructions presented in Section 3.2 to a case study where we demonstrate how the composition of two different languages can be successfully managed. In Section 5, we discuss related work, and finally conclude the paper and outline directions for future work in Section 6.

2. Background: Multilevel Modelling

MLM is a recognised research area with clear advantages in several scenarios [20]. It provides the flexibility needed to avoid the use of anti-patterns, e.g., the type-object pattern described in [11, 21] when fitting several layers of abstraction into one single level. This anti-pattern appears when both the concept and the metaconcept have to be defined in the same level, leading to convolution. However, there exist several challenges within the MLM community that hamper its wide-range adoption, such as a lack of recognised standards and fundamental concepts of the paradigm, that have led to a proliferation of different multilevel tools [22, 23] without a clear consensus and focus.

The MultEcore approach for MLM combines two-level and multilevel modelling approaches and takes the best from each world with the goal of bringing standards into MLM solutions [16, 24]. Its main goal is to facilitate the specification of multilevel hierarchies which are both generic and precise [25, 24]. These ideas are reflected in the MultEcore tool. The tool enables multilevel modelling in the Eclipse Modelling Framework (EMF), allowing us to reuse the existing EMF tools and plugins [26, 27]. MultEcore provides facilities to the modeller to define both the structure and the behaviour of multilevel hierarchies.

MultEcore is designed as a set of Eclipse plugins, giving access to its mature tool ecosystem (integration with EMF) and incorporating the flexibility of MLM. In the MultEcore approach [16], the abstract syntax is provided by MLM models and the behaviour by the so-called Multilevel Coupled Model Transformations (MCMTs) [16, 25]. Using the MultEcore tool, modellers can (i) define MLM models using the model graphical editor, (ii) define MCMTs using its rule editor, and (iii) execute specific models. The execution of MultEcore models rely on a transformation of these models into Maude [28] specifications [29]. To provide a formal description of our framework and the aforementioned features, we rely on graph transformations and corresponding parts of category theory.

2.1. Multilevel Modelling in MultEcore - Structure

The MultEcore multilevel modelling approach is based on a flexible typing mechanism based on graphs. We present in this section a summary of the formalisation in [30] on which we base the semantics of our composition construction in Section 3.2. In this formalisation, models are represented as graphs, since they are a natural way of abstracting concepts and the relations among them. Each model in our approach is identified by a name and represented as directed multigraph. Graphs are defined as follows.

Definition 1 (Graph)**.** *A Graph $G = (G^N, G^A, sc^G, tg^G)$ consists of a set of nodes G^N, a set of arrows G^A and two maps $sc^G : G^A \to G^N$ and $tg^G : G^A \to G^N$ that assign to each arrow its source and target node, respectively. These two maps must be total for the graph to be considered valid. We use the notations $x \xrightarrow{f} y$ or $f : x \to y$ to indicate that $sc^G(f) = x$ and $tg^G(f) = y$.*

Intuitively, graphs consist of nodes and arrows. A node represents a class, and an arrow represents a relation between two classes. Hence, an arrow always connects two nodes in the same graph, and any two nodes can be connected by an arbitrary number of arrows. Relations between graphs, like typing and matching, are defined by means of *graph homomorphisms.*

Definition 2 (Graph Homomorphism)**.** *A homomorphism $\varphi : G \to H$ between graphs is given by two maps $\varphi^N : G^N \to H^N$ and $\varphi^A : G^A \to H^A$ such that $sc^G;\varphi^N = \varphi^A;sc^H$ and $tg^G;\varphi^N = \varphi^A;tg^H$. Note that we use the symbol $\_;\_$ to denote composition in diagrammatic order.*

We use the terms graph and model indistinctly. Models are distributed in *multilevel modelling hierarchies.* By a multilevel modelling hierarchy we understand a tree-shaped hierarchy of models with a single root one typically depicted at the top of the hierarchy tree. Thus, hierarchies enclose a set of models which are connected via typing relations.

Figure 1 displays a simple multilevel hierarchy containing three levels of abstraction (four if we include the reserved *Ecore* model placed at the top in level 0, Figure 1(a)). Note that each graph, except the one at the top has exactly one parent graph in the hierarchy. Then, at Level 1, we branch into two paths. The models *generic-model-1* and *generic-model-2* (Figures 1(b) and 1(c), respectively) contain three nodes and one relation each. As shown in the figure, the type of a node is indicated in an ellipse at its top left side, e.g., EClass is the type of A, B, and C in model *generic-model-1*, as well as of D, E, and F in model *generic-model-2.* The type of an arrow is written near the arrow in italic font type, e.g., EReference under G in model *generic-model-1*, and under H in model *generic-model-2.* As we see below, typing relations are graph homomorphisms. However, we use these two individual typing graphical representations to express types without filling up the hierarchy graphical representations with arrows.

A hierarchy has $n+1$ abstraction levels, where n is the maximal path length in the hierarchy tree. Levels are indexed with increasing natural numbers starting from the uppermost one, with index 0. Each graph in the hierarchy is placed at

4

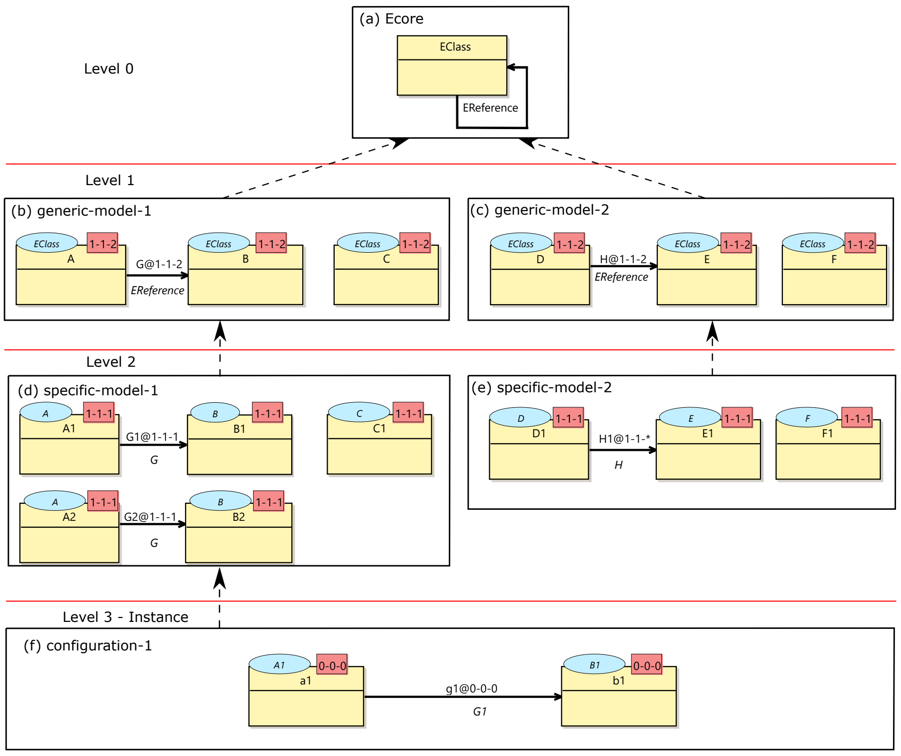


Figure 1: Multilevel hierarchy for a conceptual example

some level i, where i is the length of the path from that graph to the topmost one. To be flexible concerning abstraction levels and to support a smooth evolution of modelling descriptions, we allow certain positions in a hierarchy to be empty, i.e., filled by an empty graph. We use the notation G_i to indicate that a graph is placed at level i. For implementation reasons, we use Ecore [8] as root graph at level 0 in all example hierarchies, since Ecore is based on the concept of graph which makes it powerful enough to represent the structure of software models.

We use levels as an organisational tool, where the main rationale for locating elements in a particular level is grouping them by how abstract they are, and how reusable and useful they can be in that particular level. Thus, we encourage the *level cohesion* principle [31], that is, we recommend to organise elements that are semantically close (by means of potency and level organisation). On the contrary, we do not promote the *level segregation* principle, which establishes that level organisational semantics should be unique, i.e., aligned to one particular organisational scheme, such as *classification* or *generalisation*. We use, however, a more broad *abstraction* semantics. Furthermore, the MultEcore

5

tool checks correct potency and typing safeness.[1]

In Figure 1, red horizontal lines are used to indicate the separation between two consecutive levels, and upwards dashed arrows represent sequences of graphs that constitute *typing chains* G_i, G_{i-1}, …, G_1, G_0.

For flexibility reasons, we allow typing to jump over abstraction levels, i.e., an element in graph G_i may have no type in G_{i-1} but only in one (or more) of the graphs in G_{i-2}, …, G_1, G_0. Moreover, two different elements in the same graph may be typed by elements located in different graphs along the typing chain. To formalise this kind of flexible typing, we use *partial graph homomorphisms.*

Definition 3 (Partial Graph Homomorphism)**.** *A* ***partial graph homomorphism*** $\varphi : G \rightarrow\!\!\!\!\!\circ\; H$ *is given by a subgraph* $D(\varphi) \sqsubseteq G$*, called the* ***domain of definition*** *of* φ*, and a graph homomorphism* $\varphi : D(\varphi) \to H$*.*

To express transitivity of typing and later also compatibility of typing, we need as well the composition of partial graph homomorphisms as a partial order between partial graph homomorphisms.

Definition 4 (Composition of partial graph homomorphisms)**.** *The* ***composition*** $\varphi;\psi : G \rightarrow\!\!\!\!\!\circ\; K$ *of two partial graph homomorphisms* $\varphi : G \rightarrow\!\!\!\!\!\circ\; H$ *and* $\psi : H \rightarrow\!\!\!\!\!\circ\; K$ *is defined as follows:*

- $D(\varphi;\psi) := \varphi^{-1}(D(\psi))$*, i.e., for all nodes* $e \in G^N$ *we have* $e \in D(\varphi;\psi)^N$ *iff* $e \in D(\varphi)^N$ *and* $\varphi^N(e)$ *in* $D(\psi)^N$*, and for all arrows* $f \in G^A$ *we have* $f \in D(\varphi;\psi)^A$ *iff* $f \in D(\varphi)^A$ *and* $\varphi^A(f) \in D(\psi)^A$*.*
- $(\varphi;\psi)^N(e) := \psi^N(\varphi^N(e))$ *for all* $e \in D(\varphi;\psi)^N$ *and* $(\varphi;\psi)^A(f) := \psi^A(\varphi^A(f))$ *for all* $f \in D(\varphi;\psi)^A$*.*

More abstractly, the composition of two partial graph homomorphisms is defined by the following commutative diagram of total graph homomorphisms. (Keep in mind that inverse images are just special pullbacks.)

$$\begin{tikzcd}
 & & D(\varphi;\psi) \arrow[dl, hook', "\sqsubseteq"'] \arrow[dr, "\varphi_{|\psi}"] \arrow[ddrr, bend left, "\varphi;\psi"] & & \\
 & D(\varphi) \arrow[dl, hook', "\sqsubseteq"'] \arrow[dr, "\varphi"] & PB & D(\psi) \arrow[dl, hook', "\sqsubseteq"'] \arrow[dr, "\psi"] & \\
G & & H & & K
\end{tikzcd}$$

Note that $D(\varphi;\psi) = D(\varphi)$ *if* φ *is total, i.e.,* $H = D(\varphi)$*.*

Definition 5 (Order between partial graph homomorphisms)**.** *For any two parallel partial graph homomorphisms* $\varphi, \phi : G \rightarrow\!\!\!\!\!\circ\; H$ *we have* $\varphi \preceq \phi$ *if, and only if,* $D(\varphi) \sqsubseteq D(\phi)$ *and, moreover,* $\sqsubseteq;\phi = \varphi$ *for the corresponding total graph homomorphisms* $\varphi : D(\varphi) \to H$ *and* $\phi : D(\phi) \to H$*.*

[1]Typing relations cannot be circular, reversed or inconsistent neither vertically, i.e., within the same hierarchy, nor horizontally, i.e., if we consider more than one hierarchy.

Typing chains appear in multilevel hierarchies as sequences of graphs from a certain graph in the hierarchy all the way up to the top of the hierarchy. They are formally defined in Definition 6.

Definition 6 (Typing Chain $\mathcal{G}$)**.** *A typing chain $\mathcal{G} = (\overline{G}, n, \tau^G)$ is given by a natural number n, a sequence $\overline{G} = [G_n, G_{n-1}, \ldots, G_1, G_0]$ of graphs of length $n+1$ and a family $\tau^G = (\tau^G_{j,i} : G_j \mathrel{-\!\!\circ\!\!\rightarrow} G_i \mid n \geq j > i \geq 0)$ of partial graph homomorphisms, called* ***typing morphisms****, satisfying the following properties:*

- ***Total:*** *All the morphisms $\tau^G_{j,0} : G_j \to G_0$ with $n \geq j \geq 1$ are total.*
- ***Transitive:*** *For all $n \geq k > j > i \geq 0$ we have $\tau_{k,j}; \tau_{j,i} \preceq \tau_{k,i}$.*
- ***Connex:*** *For all $n \geq k > j > i \geq 0$ we have $D(\tau^G_{k,j}) \cap D(\tau^G_{k,i}) \sqsubseteq D(\tau^G_{k,j}; \tau^G_{j,i})$ and, moreover, $\tau^G_{k,j}; \tau^G_{j,i}$ and $\tau_{k,i}$ coincide on $D(\tau^G_{k,j}) \cap D(\tau^G_{k,i})$.*

Totality, transitivity and connexity ensure that for any element e in any graph G_i in a typing chain there exists a unique index m_e, with $i > m_e \geq 0$, such that e is in the domain of the typing morphism τ^G_{i,m_e} but not in the domain of any typing morphism $\tau^G_{i,j}$ with $i > j > m_e$.

Definition 7 (Individual Direct Type)**.** *For any e in a graph G_i in a typing chain $\mathcal{G} = (\overline{G}, n, \tau^G)$, with $n \geq i \geq 1$, we call $ty(e) := \tau^G_{i,m_e}(e)$ its individual direct type. We say also that e is a direct instance of $ty(e)$.*

By $df(e) = i - m_e$ we denote the difference between i and the level where $ty(e)$ is located. Usually, this difference is 1, which means that the type of e is placed at the level right above it. For convenience, we use the following abbreviations:

$$\begin{array}{lll} ty^2(e) = ty(ty(e)) & ty^3(e) = ty(ty(ty(e))) & \ldots \\ df^2(e) = df(e) + df(ty(e)) & df^3(e) = df^2(e) + df(ty^2(e)) & \ldots \end{array}$$

From a general point of view, we obtain for any e in G_i a sequence of typing assignments of length $1 \leq s_e \leq$ i with $(i - df^{s_e}(e)) = 0$. The number s_e of steps depends individually on the item e. We call any of the elements $ty(e)$, $ty^2(e)$, $ty^3(e)$, ... a *transitive type* of e. The requirement that the domains of definition of typing morphisms are subgraphs ensures that for any arrow $x \xrightarrow{f} y$ in any graph G_i the non-dangling condition is satisfied: The source and the target of the direct type $ty(f) \in G_{m_f}$ of f are transitive types of x and y, respectively. Finally, we want to mention that any sequence $[G_n, G_{n-1}, \ldots, G_1, G_0]$ of graphs such that any e in any graph G_i with $n \geq i \geq 1$ has a unique individual direct type $ty(e)$ in one of the graphs $G_{i-1}, \ldots, G_1, G_0$ gives rise to a typing chain, according to Definition 6, as long as the non-dangling condition for arrows is satisfied (compare [30]).

Level 2 in Figure 1 contains instances of models described in Level 1 (called *specific-model-1* and *specific-model-2*). The nodes and references in the models depicted in Figures 1(d) and 1(e) are typed by elements defined, in this case, at

7

Level 1, e.g., for A1 node and G1 relation the types are A and G, respectively. At the bottom of the hierarchy (Figure 1(f)), we have (at Level 3) the *Instance* level where model *configuration-1* is displayed. Note that, even though there exists one typing chain per model (except for *Ecore*), we only focus on the typing chain computed from the bottommost level (Instance level). Notice also that in the hierarchy shown in Figure 1, the typing chain is represented by upwards dashed arrows from the instance level given by the left-hand branch of the hierarchy.

The last concept introduced in Figure 1 is *potency*, displayed as three numbers in a red box at the top right of every node, and concatenated to the name after "@" for every reference. Potencies are used on elements as a means of restricting the levels at which these elements may be used to type other elements. Thanks to potencies on elements we can define the degree of flexibility / restrictiveness we want to allow on the elements of our multilevel hierarchy. These three values are used to constrain the instantiation of elements so that the flexibility of our approach can be controlled in order to use concepts in a sensible manner. The first two values, *start* and *end*, specify the range of levels below, relative to the current one, where the element can be directly instantiated. In the example hierarchy in Figure 1, these two values are always 1, meaning that the element can only be instantiated in the level right below. A potency value of $2 - 4 - X$, for instance, would mean that an element can be directly instantiated two, three and four levels below the one where the element is defined. The third value, *depth*, is used to control the maximum number of times that the element can be transitively instantiated, regardless of the levels where this happens. That is, the amount of times an instance of that element can be re-instantiated.

In the example in Figure 1, all elements at level 1 have a depth of 2, meaning that they can be directly instantiated, and these instances can be instantiated themselves again (i.e. two times at most). This value is therefore dependent on the value of the type, and the depth of an element must always be strictly less than the depth of its type. For this reason, all elements in level 2 have a depth value of 1, and their instances of 0, meaning that they cannot be further instantiated. For elements in level 3, the instance level, the first two values also become 0, since there are no further levels below where these elements could be instantiated. In other words, the potency $0 - 0 - 0$ is used to enforce that elements at the bottom level (3) are used purely as instances, which cannot be refined further into levels below it. In general, the default potency for elements is $1 - 1 - *$ ($*$ meaning unbounded), and the potencies for all elements in the top level (Ecore) is $0 - * - *$ in order to allow, exceptionally, self-typing and to keep all instantiation initially unconstrained.

2.2. Multilevel Modelling in MultEcore - Operational semantics

Transformation rules can be used to represent actions that may happen in the system. Conventional in-place model transformations (MTs) are rule-based modifications of a source model (specified in the left-hand side of the rule) resulting in a new state of such a model (determined by its right-hand side). The left-hand side takes as input (a part of) a model and it can be understood as

8

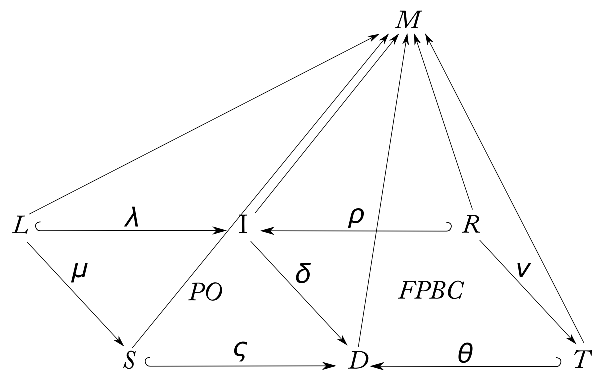


Figure 2: Conventional two-level MT rule

the pattern we want to find in our original model. The right-hand side describes the transformation we want to perform on our model and thereby the next state of the system.

Since we use graphs to formalise models, we employ graph transformation rules to express the operational semantics of multilevel models. A graph transformation rule is defined by a left L and a right R pattern. These patterns are graphs which are mapped to each other via graph morphisms λ, ρ from or to a third graph I, such that L, R, I constitute either a span ($L \longleftarrow I \longrightarrow R$) or a co-span ($L \longrightarrow I \longleftarrow R$), respectively [32, 33]. These graph morphisms are typically homomorphisms, and more specifically inclusions. Then in the span version, the graph I is the intersection of L and R, while it is the union in the co-span version. In our approach, In this paper, we use the co-span version of graph transformation rules since the graph I can be used to collect the whole context between L and R, as well as due to advantages related to the properties of the constructions used in the application of these rules [33] (see also below).

Figure 2 depicts the application of a graph transformation rule. To apply a rule ($L \hookrightarrow I \hookleftarrow R$) to a source graph S, a match μ of the left pattern in S has to be found, i.e., a graph homomorphism $\mu : L \longrightarrow S$. Then, using a pushout construction (PO), followed by a final pullback complement construction (FPBC), a target graph T will be produced [33].

We use MTs to provide definitions of behaviour by means of so-called Multilevel Coupled Model Transformations (MCMTs) [16]. MCMTs have been proposed as a means to take traditional two-level transformations rules (Figure 2) into the multilevel model world, with the right balance between precision and flexibility (see [16] for details). That is, MCMTs allow us to exploit multilevel modelling capabilities within the context of MTs. In this paper, we focus on the use of MCMTs to describe the operational semantics of DSMLs. MCMTs can also be used with other purposes, for instance, MCMTs have been used to check the structural correctness of models in [34, 27].

Figure 3 shows a simple example of an MCMT rule (called *Add and Connect*) that models the creation of a new node and a relation between the existing node and the new one.

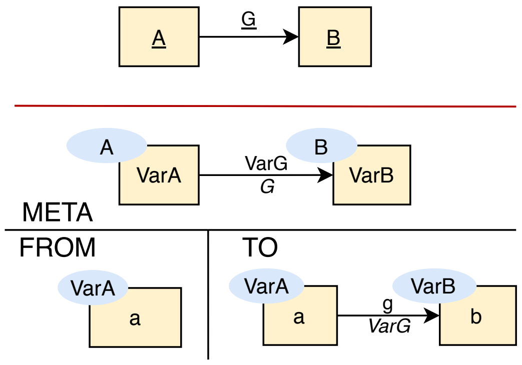


Figure 3: Rule *Add and Connect*: The execution of this rule gives a new state on the model where a new node is created and connected to the first one

The FROM and TO blocks describe the left pattern and the right pattern of the rule, respectively. The META block depicts a typing chain allowing us to locate types in any level of the chain that can be used as individual types for the items in the FROM and TO block, respectively. Notice that this is quite powerful, as META facilitates the definition of an entire multilevel pattern. At the top level of Figure 3, we mirror parts of *generic-model-1*, defining elements like A, B and G as constants. We differentiate constants as their names are underlined and their types are not specified via the ellipse above (for nodes) or the italic text (for references). The use of constants constrains the matching process, significantly reducing the amount of matches. The rule can be applied to models (instances) typed by the left-hand typing chain of Figure 1 (i.e., *specific-model-1, generic-model-1, Ecore*).

Note, that the horizontal lines do not enforce consecutiveness between the levels specified in the rule with respect to the hierarchy. This leads to a more natural way of defining that a type is defined at some level above, without explicitly stating in which level. In fact, this also promotes flexibility in case of future modifications of the number of branches (horizontal dimension) and the depth (vertical dimension) of hierarchies. Consider for the horizontal dimension, for instance, in the example in Figure 1, adding a new model called *specific-model-1'*, branching at level 2, as instance of *generic-model-1*. For the vertical dimension, consider for example introducing a new level between levels 2 and 3 to create a more refined model (called, e.g., *more-specific-model-1*). The key aspect is that none of these extensions would require the modification of other models in the hierarchy, nor the rule depicted in Figure 3, while the MCMT would still be valid. This flexibility is achieved as we allow the types on the variables to be transitive types. For instance, VarA (placed at the second level of the META), typed by the variable A, would match any node which indirectly has A as type, or ultimately will match to A if no indirect one is found. A correct match of the rule comes when an element, coupled together with its type, fits an instance of VarA (e.g., a located in the FROM part).

Given the current state of the hierarchy in Figure 1, any instances of elements matching the pattern VarA would be candidates to perform the transformation. This in turn makes it possible to apply the rule to either instances of A1 or to instances of A2 (these elements are defined in model *specific-model-1* at Figure 1).

The general structure of an MCMT and its application is displayed in Figure 4. The figure can be visualised as two flat trees, each of them defined by typing chains and connected to each other by matching morphisms.

$$\begin{tikzcd}
 & MM_0 \arrow[rrr, "\beta_0"] \arrow[rrrd, phantom, "="] & & & TG_0 & \\
 & MM_1 \arrow[u, "\circ" marking] \arrow[rrrd, "\beta_1"] \arrow[rrrdd, phantom, "="] & & & \cdots \arrow[u, "\circ" marking] & \\
 & \cdots \arrow[u, "\circ" marking] & & & TG_i \arrow[u, "\circ" marking] & \\
 & MM_n \arrow[u, "\circ" marking] \arrow[rrrd, "\beta_n"] & & & \cdots \arrow[u, "\circ" marking] & \\
L \arrow[r, hook, "\lambda"] \arrow[ur, "\circ" marking] \arrow[rrrd, "\mu"'] \arrow[uuuur, gray, bend left, "\circ" marking] \arrow[uuur, gray, bend left, "\circ" marking] & I \arrow[u, "\circ" marking] \arrow[rrrd, "\delta"] \arrow[uuuu, gray, bend left, "\circ" marking] \arrow[uuu, gray, bend right, "\circ" marking] \arrow[rrd, phantom, "PO"] & R \arrow[l, hook', "\rho"'] \arrow[ul, "\circ" marking] \arrow[rrrd, "\nu"] \arrow[uuuul, gray, bend right, "\circ" marking] \arrow[uuul, gray, bend right, "\circ" marking] \arrow[rrd, phantom, "FPBC"] & & TG_m \arrow[u, "\circ" marking] & \\
 & & & S \arrow[r, hook, "\varsigma"] \arrow[ur, "\circ" marking] \arrow[uuuuur, gray, bend left, "\circ" marking] \arrow[uuur, gray, bend left, "\circ" marking] & D \arrow[u, "\circ" marking] \arrow[uuuuu, gray, bend left, "\circ" marking] \arrow[uuu, gray, bend right, "\circ" marking] & T \arrow[l, hook', "\theta"'] \arrow[ul, "\circ" marking] \arrow[uuuuul, gray, bend right, "\circ" marking] \arrow[uuul, gray, bend right, "\circ" marking]
\end{tikzcd}$$

Figure 4: Formal construction for MCMT

The tree on the left contains the pattern that the user defines in the rule. It consists of the left and right parts of the rule (FROM and TO, respectively), represented as L and R in the diagram, and the interface I that is the union of both L and R, being λ and ρ inclusion graph homomorphisms.

These three graphs are typed by elements in the same typing chain $\mathcal{MM} = (\overline{MM}, n, \tau^{MM})$, defined in the META block, which is depicted as a sequence of metamodels MM_i, for $0 \leq i \leq n$, that ends with the root of the chain MM_0 (Ecore in our case). The multilevel typing of the graphs L, I, R is given by families of typing morphisms.

Definition 8 (Multilevel Typed Graph)**.** *A* ***multilevel typed graph*** (H, σ^H) *is a graph H with a* ***multilevel typing*** $\sigma^H : H \Rightarrow \mathcal{G}$ *of H over a typing chain $\mathcal{G}$ = $(\overline{G}, n, \tau^G)$ given by a family $\sigma^H = (\sigma^H : H \multimap\!\!\rightarrow G_i \mid n \geq i \geq 0)$ of partial graph homomorphisms.*

So, the rule is given by multilevel typed graphs $(L, \sigma^L : L \Rightarrow \mathcal{MM})$, $(R, \sigma^R : R \Rightarrow \mathcal{MM})$, $(I, \sigma^I : I \Rightarrow \mathcal{MM})$ with $I = L \cup R$ such that σ^L and σ^R coincide on the intersection $L \cap R$ and σ^I is constructed as the union of σ^L and σ^R.

For the rule to be applied, we have to find a match (a graph homomorphism μ) of the pattern graph L into an instance graph S at the bottom of the current application hierarchy. The choice of S determines a sequence $[S, TG_m, TG_{m-1}, \dots, TG_1, TG_0]$ of graphs from S up to the top of the hierarchy. The sequence $[TG_m, TG_{m-1}, \dots, TG_1, TG_0]$ of graphs constitutes a typing chain $\mathcal{TG} = (\overline{TG}, m, \tau^{TG})$ and the family of typing morphisms from S into TG_i, $m \geq i \geq 0$ turns S into a multilevel typed graph $(S, \sigma^S : S \Rightarrow \mathcal{TG})$. The match $\mu : L \to S$ has, however, to satisfy some application conditions: There has to be a match of the typing chain $\mathcal{MM}$ into the typing chain $\mathcal{TG}$ that is compatible with the multilevel typings σ^L, σ^S and the match μ. Matches of typing chains are described by a very flexible concept of morphisms between typing chains.

Definition 9. *A **typing chain morphism** $(\phi, f) : \mathcal{G} \to \mathcal{H}$ between two typing chains $\mathcal{G} = (\overline{G}, n, \tau^G)$ and $\mathcal{H} = (\overline{H}, m, \tau^H)$ with $n \leq m$ is given by*

- *a function $f : [n] \to [m]$, where $[n] = \{0, 1, 2, \dots, n\}$, such that $f(0) = 0$ and $j > i$ implies $f(j) > f(i)$ for all $i, j \in [n]$, and*
- *a family of total graph homomorphisms $\phi = (\phi_i : G_i \to H_{f(i)} \mid i \in [n])$ with*

$$\tau^G_{j,i}; \phi_i \preceq \phi_j; \tau^H_{f(j),f(i)} \quad \textit{for all } n \geq j > i \geq 0. \tag{1}$$

*A typing chain morphism $(\phi, f) : \mathcal{G} \to \mathcal{H}$ is called **closed** if, and only if, $\tau^G_{j,i}; \phi_i = \phi_j; \tau^H_{f(j),f(i)}$ for all $n \geq j > i \geq 0$.*

There are three flexibility features we want to underline: (1) Jumps of typing can be arbitrarily stretched in the sense, that the difference $f(j) - f(i)$ can be bigger than the difference $j - i$. (2) We require, in general, only that typing is preserved, i.e., if an element $\mathtt{e}$ in G_j has a transitive type in G_i then the image $\phi_j(\mathtt{e})$ in $H_{f(j)}$ is required to have a transitive type in $H_{f(i)}$. For closed typing chain morphisms, we require, however, that typing is also reflected, i.e., if the image $\phi_j(\mathtt{e})$ in $H_{f(j)}$ has a transitive type in $H_{f(i)}$ it is required that $\mathtt{e}$ has a transitive type in G_i. (3) The granularity of typing does not need to be preserved, i.e., if an element $\mathtt{e}$ in G_j has a direct (!) type in G_i then the image $\phi_j(\mathtt{e})$ in $H_{f(j)}$ needs only to have a transitive type in $H_{f(i)}$.

The graph homomorphisms $\beta_n, \dots, \beta_1, \beta_0$ and the assignments $0 \mapsto 0, 1 \mapsto i, \dots, n \mapsto m$ in Figure 4 depict the required typing chain morphism (match) $(\beta, f) : \mathcal{MM} \to \mathcal{TG}$. To describe type compatibility of matches and the result of an MCMT application we need to have the composition of typing chain morphisms at hand.

Definition 10 (Composition of typing chain morphisms)**.** *The composition $(\phi, f); (\psi, g) : \mathcal{G} \to \mathcal{K}$ of two typing chain morphisms $(\phi, f) : \mathcal{G} \to \mathcal{H}$, $(\psi, g) :$*

$\mathcal{H} \to \mathcal{K}$ between typing chains $\mathcal{G} = (\overline{G}, n, \tau^G)$, $\mathcal{H} = (\overline{H}, m, \tau^H)$, $\mathcal{K} = (\overline{K}, l, \tau^K)$ with $n \leq m \leq l$ is defined by

$$(\phi, f); (\psi, g) := (\phi; \psi_{\downarrow f}, f; g)$$

where $\psi_{\downarrow f} := (\psi_{f(i)} : H_{f(i)} \to K_{g(f(i))} \mid i \in [n])$ and thus

$$\phi; \psi_{\downarrow f} := (\phi_i; \psi_{f(i)} : G_i \to K_{g(f(i))} \mid i \in [n]).$$

Chain denotes the category of typing chains and typing chain morphisms.

It turns out that multilevel typings are not appropriate to formulate adequate compatibility conditions for matches. Therefore, we describe multilevel typing by means of inclusion chains and typing chain morphisms.

Lemma 1 (Inclusion chain)**.** *For any graph H we can extend any sequence $\overline{H} = [H_n, H_{n-1}, \ldots, H_1, H_0]$ of subgraphs of H, with $H_0 = H$, to a typing chain $\mathcal{H} = (\overline{H}, n, \tau^H)$ where for all $n \geq j > i \geq 0$ the corresponding* ***partial inclusion graph homomorphism*** *$\tau^H_{j,i} : H_j \multimap H_i$ is given by $D(\tau^H_{j,i}) := H_j \cap H_i$ and the span of total inclusion graph homomorphisms*

$$\begin{tikzcd} H_j & D(\tau^H_{j,i}) = H_j \cap H_i \arrow[l, hook', "\sqsubseteq"'] \arrow[r, hook, "\tau^H_{j,i}"] & H_i \end{tikzcd}$$

By means of Lemma 1, we can represent now the four given multilevel typings $\sigma^L : L \Rightarrow \mathcal{MM}$, $\sigma^I : I \Rightarrow \mathcal{MM}$, $\sigma^R : R \Rightarrow \mathcal{MM}$, and $\sigma^S : S \Rightarrow \mathcal{TG}$, equivalently, by four corresponding inclusion chains (see Figures 5 and 6)

- $\mathcal{L} = (\overline{L}, n, \tau^L)$ with $L_i := D(\sigma^L_i)$ for all $i \in [n]$ and thus $L_0 = L$,
- $\mathcal{I} = (\overline{I}, n, \tau^I)$ with $I_i := D(\sigma^I_i)$ for all $i \in [n]$ and thus $I_0 = I$,
- $\mathcal{R} = (\overline{R}, n, \tau^R)$ with $R_i := D(\sigma^R_i)$ for all $i \in [n]$ and thus $R_0 = R$ and
- $\mathcal{S} = (\overline{S}, m, \tau^S)$ with $S_j := D(\sigma^S_j)$ for all $j \in [m]$ and thus $S_0 = S$,

together with four typing chain morphisms

- $(\sigma^L, id_{[n]}) : \mathcal{L} \to \mathcal{MM}$ with $\sigma^L = (\sigma^L_i : L_i \to MM_i \mid i \in [n])$,
- $(\sigma^I, id_{[n]}) : \mathcal{I} \to \mathcal{MM}$ with $\sigma^I = (\sigma^I_i : I_i \to MM_i \mid i \in [n])$,
- $(\sigma^R, id_{[n]}) : \mathcal{R} \to \mathcal{MM}$ with $\sigma^R = (\sigma^R_i : R_i \to MM_i \mid i \in [n])$, and
- $(\sigma^S, id_{[m]}) : \mathcal{S} \to \mathcal{TG}$ with $\sigma^S = (\sigma^S_j : S_j \to TG_j \mid j \in [m])$.

By construction, we have $I_i = L_i \cup R_i$ for all $i \in [n]$ thus the family of inclusion graph homomorphisms $\lambda_i : L_i \hookrightarrow I_i$, $i \in [n]$ establishes a closed typing chain morphism $(\lambda, id_{[n]}) : \mathcal{L} \to \mathcal{I}$ while the family of inclusion graph homomorphisms $\rho_i : R_i \hookrightarrow I_i$, $i \in [n]$ establishes a closed typing chain morphism $(\rho, id_{[n]}) : \mathcal{R} \to \mathcal{I}$. Finally, the construction of $\mathcal{I}$ ensures type compatibility of the rule:

$$(\lambda, id_{[n]}); (\sigma^I, id_{[n]}) = (\sigma^L, id_{[n]}) \quad \text{and} \quad (\rho, id_{[n]}); (\sigma^I, id_{[n]}) = (\sigma^R, id_{[n]}) \quad (2)$$

13

$$\begin{tikzcd}
 & & \mathcal{MM} \arrow[dr, "{(\beta, f)}"] & \\
 & & & \mathcal{TG} \\
\mathcal{L} \arrow[urr, "{(\sigma^L, id_{[n]})}"] \arrow[rr, "{(\lambda, id_{[n]})}"] \arrow[dr, "{(\mu, f)}"'] & & \mathcal{I} \arrow[uu, "{(\sigma^I, id_{[n]})}"'] \arrow[dr, "{(\delta, f)}"] & \\
 & \mathcal{S} \arrow[uurr, "{(\sigma^S, id_{[m]})}"] \arrow[rr, "{(\varsigma, id_{[m]})}"'] & & \mathcal{D} \arrow[uu, dotted, "{(\sigma^D, id_{[m]})}"']
\end{tikzcd}$$

Figure 5: Pushout step

Type compatibility of the matches $\mu : L \to S$ and $(\beta, f) : \mathcal{MM} \to \mathcal{TG}$ means that $\mu : L \to S$ restricts for each $i \in [n]$ to a map $\mu_i : L_i \to S_{f(i)}$ such that this family of graph homomorphisms establishes a typing chain morphism $(\mu, f) : \mathcal{L} \to \mathcal{S}$ satisfying the equation

$$(\sigma^L, id_{[n]}); (\beta, f) = (\mu, f); (\sigma^S, id_{[m]}). \tag{3}$$

The type compatibility requirements for rules and matches ensure that the pushout for graphs, at the bottom of Figure 4, gives rise to a pushout for the corresponding inclusion chains at the bottom of Figure 5: For each $n \geq i > 0$ we set $D_{f(i)} := S_{f(i)} \cup \delta(I_i)$ thus the co-span $S \overset{\varsigma}{\hookrightarrow} D \overset{\delta}{\leftarrow} I$ restricts to a co-span $S_{f(i)} \overset{\varsigma_{f(i)}}{\hookrightarrow} D_{f(i)} \overset{\delta_i}{\leftarrow} I_i$. This co-span can be proven to be a pushout of the span $S_{f(i)} \overset{\mu_i}{\leftarrow} L_i \overset{\lambda_i}{\to} I_i$. To get a complete inclusion chain $\mathcal{D}$ of length m, we simply set $D_j := S_j$ and $\varsigma_j := id_{S_j}$ for all $j \in [m] \setminus f([n])$. The complex proof that this simple construction provides indeed a pushout in **Chain** can be found in [30].

Since the bottom square in Figure 5 is a pushout, the type compatibily conditions (2) and (3) ensure that there is a unique typing chain morphism $(\sigma^D, id_{[m]})$ from $\mathcal{D}$ to $\mathcal{TG}$ such that

$$(\delta, f); (\sigma^D, id_{[m]}) = (\sigma^I, id_{[n]}); (\beta, f)\ ,\ (\varsigma, id_{[m]}); (\sigma^D, id_{[m]}) = (\sigma^S, id_{[m]}) \tag{4}$$

This shows, that we have indeed constructed a type compatible multilevel typing of the graph D.

For the second step of rule application, namely the FPBC construction shown in Figure 6, we first construct FPBC in category **Graph** and obtain T. It will remain to reconstruct the typing of T in order to create an inclusion chain $\mathcal{T} = (\overline{T}, m, \tau^T)$. To achieve this, we construct the reduct of $\mathcal{D} = (\overline{D}, m, \tau^D)$ along $\theta : T \hookrightarrow D$ and $id_{[m]}$ by level-wise intersection (pullback) for all $n \geq i \geq 1$. In such a way, we obtain an inclusion chain $\mathcal{T} = (\overline{T}, m, \tau^T)$ together with a closed

14

$$\begin{tikzcd}
\mathcal{MM} \arrow[dr, "{(\beta, f)}"] & & \\
& \mathcal{TG} & \\
\mathcal{I} \arrow[uu, "{(\sigma^I, id_{[n]})}"] \arrow[dr, "{(\delta, f)}"'] & \mathcal{R} \arrow[l, "{(\rho, id_{[n]})}"] \arrow[uul, "{(\sigma^R, id_{[n]})}"'] \arrow[dr, "{(\nu, f)}"] & \\
& \mathcal{D} \arrow[uu, "{(\sigma^D, id_{[m]})}"] & \mathcal{T} \arrow[l, "{(\theta, id_{[m]})}"] \arrow[uul, dotted, "{(\sigma^T, id_{[m]})}"']
\end{tikzcd}$$

Figure 6: Final pullback complement step

typing chain morphism $(\theta, id_{[m]}) : \mathcal{T} \rightarrow \mathcal{D}$. The multilevel typing of $\mathcal{T}$ is simply borrowed from $\mathcal{D}$, that is, we define (see Fig. 6)

$$(\sigma^T, id_{[m]}) \coloneqq (\theta, id_{[m]}); (\sigma^D, id_{[m]}) \tag{5}$$

and this trivially gives us the intended type compatibility of $(\theta, id_{[m]})$.

The specific conditions that are required are out of the scope of this paper, and can be consulted in the technical report [30].

3. Composition

In current MDSE practice, DSMLs are built by language designers using a metamodel defined by a general-purpose meta-modelling language [35], like MOF. As mentioned in Section 1, this in turn leads to a metamodel that describes the instances that users of the language can build in the immediate metalevel below. Thus, languages are specified within two levels: definition and usage. However, the increasing complexity of software systems advocates the need for more DSMLs as refinement of general-purpose languages [36]. Hence, the need for alternative techniques that alleviate the two-level restrictions (provided, for instance, by MLM) becomes progressively significant.

By using MLM capabilities, one could customise families of similar DSMLs, where certain commonalities are shared. In this context, the challenge for language designers is to take advantage of the existing commonalities among similar DSMLs by reusing, as much as possible, formerly defined language constructs [2]. Furthermore, having a way to modularise a language to create *features* — to later reuse and combine them — can be used in different manners to produce tailor-made DSMLs targeting the needs of well-defined audiences. This feature-oriented approach to DSML engineering requires the definition of DSMLs in a modularised fashion where language features are implemented as interdependent and composable language modules.

3.1. Standard Composition Approach

A consequence of having DSMLs that tackle scoped problem spaces (enhancing separation of concerns), is that often we find ourselves thinking that one of them is not enough to reason about certain global properties or to execute the complete system. In other words, it might be necessary to compose some of the constructed models to achieve such goals. In general, model composition unfolds along two dimensions, structure and behaviour.

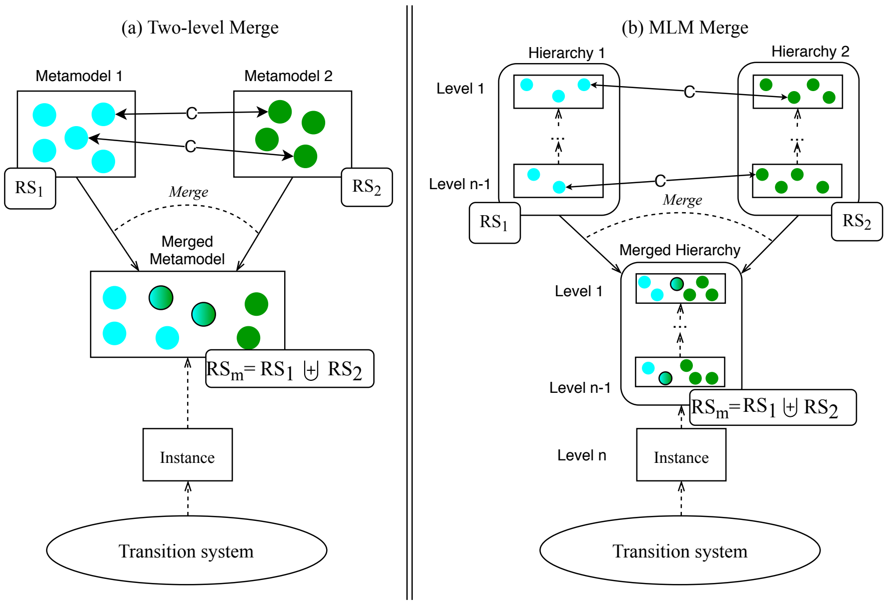


Figure 7: Two-level merge combination vs MLM merge combination

Commonly, frameworks that offer composition operators had to define their own composition rules and provide custom-made implementations of such operators (e.g., through model transformations). To alleviate ad-hoc implementations and to provide standard operations, several researchers have proposed in [19] a paradigmatic *merging* operation for structure composition and *event scheduling* for behavioural composition. Intuitively, merging refers to the operation in which “the common elements are included only once, while the rest are preserved”. Figure 7(a) shows how the *merge* operation in [19] works for two level approaches. Formally, a merge combination operator takes two metamodels, *Metamodel 1* and *Metamodel 2* as inputs, as well as a set of correspondence tuples $C = \{\langle e_x, e_y \rangle, \ldots\}$ with $e_x \in$ *Metamodel 1* and $e_y \in$ *Metamodel 2*. The merge combination operator produces a new output *Merged Metamodel* that contains, for each tuple $\langle e_x, e_y \rangle \in C$, a single metamodel element. All metamodel elements in *Metamodel 1* and *Metamodel 2* that are not given a correspondence in C are simply copied into the *Merged Metamodel*. In fact, this common and standard

16

representation of merging is a colimit construction and goes back to Burstall and Goguen's work in the late 70's [37].

Note that the elements displayed as circles in either of the metamodels, are just abstract representations and could be a node or a reference; they are displayed in this way to show how the combination is done after identifying corresponding elements. To represent each of the merged elements we use gradient colour surrounded by line, which represent combined elements originally coming from two individual ones. Also, the merge operator could take more than two metamodels as inputs, as long as the set of correspondences C is properly specified [38], but we discuss here the case of two for simplification purposes.

RS_1 and RS_2 are the sets of transformation rules attached to *Metamodel 1* and *Metamodel 2*, respectively. In the same way, as we obtain a *Merged Metamodel* by the merge operation, a set of rules (RS_m) to be attached to such a merged metamodel is produced by the disjoint union of each of the rule sets ($\mathsf{RS}_\mathrm{m} = \mathsf{RS}_1 \uplus \mathsf{RS}_2$).

Instance models can be then specified by defining elements that are typed (recall that dashed arrows represent typing graph morphisms) by elements located in the *Merged Metamodel*. These instances can be executed producing the *Transition system* (state space) which is obtained by applying the rules that come from the resulting rule set RS_m.

If we apply the merging approach to the MLM case, we get the situation depicted in Figure 7(b). Following the same approach as for the two-level case, we merge two multilevel hierarchies, *Hierarchy 1* and *Hierarchy 2*, for which the merging process would be done level-wise. If there exists some level mismatch between the hierarchies, one can still establish correspondences among elements, however, the resulting *Merged Hierarchy* must be structurally correct and fulfil the corresponding multilevel constraints. The degree of safeness of the different proposals implementing this approach depends on the amount of *sanity checks* in each of them [31]. As stated at the beginning of Section 2.1, in our implementation we provide mechanisms to assure that potency on elements is preserved, and typings are correctly applied.

Shortcomings of the standard merge operator. A crucial shortcoming present in the merge composition approach is the loss of the "individuality" nature of the merged elements (see also [39] for further shortcomings related to constraint checking). This means that the original elements that have been merged into a new one cannot be used separately after the merge. This capability might be useful in several situations. For example, when the elements about to be composed are not identical, but powering up each other. In these situations, we may need to use in our models the merged elements when we want to take advantage of all the features each of them provides. However, certain parts of the model might require their isolated aspects (i.e., the original, separated elements) to be available. These merged elements are no longer available as individual elements of the metamodel and hence cannot be instantiated at the *Instance* level.

3.2. Composition of hierarchies in MultEcore

Our proposal is to provide elements with multiple *natures*. Natures can be dynamically added and removed, so elements can have their own specific features, while still being able to define a combined and enriched nature. Our formalisation of typing chains allows us to incorporate or remove additional natures, as types, to elements. For instance, given a situation where we are working with two typing chains, each of our nodes and references residing at the instance level would be double-typed, each one provided by each of the typing chains. But also, at any time, a typing chain can be removed without affecting the other. Elements can therefore have, simultaneously, as many types as we need. This can be seen as an aspect-like mechanism that we can use as we require, being able to use aspects independently or together. The same principles apply to the definitions of behaviour by the amalgamation of MCMTs (Section 3.2.2). The fact that typing chains may be added and removed as needed makes the composition of DSMLs very flexible.

3.2.1. Composition of multilevel modelling hierarchies

Our MLM approach does not restrict the number of typing chains that can be specified in a hierarchy. Frequently, we denote a multilevel hierarchy as the *main* or *default* one and call it *application hierarchy*, since it represents the main language being designed. An application hierarchy can optionally include an arbitrary number of *supplementary hierarchies* which add new aspects to the application one. Note that we distinguish the typing chains and individual typing relations of the application hierarchy with blue colours, and use green for the supplementary ones. Adding or removing supplementary hierarchies is made possible by the incorporation or extraction of additional typing chains. For instance, we might have different hierarchies (physically separated, e.g., different projects in the MultEcore tool) that we want to compose. Such a result can be achieved by assigning the role of application hierarchy to one of them and adding the rest as supplementary ones. These two different "roles" assigned to hierarchies are used for the most part in this paper, since it facilitates the reusability and the modularisation of the system being modelled. However, it is important to point out that, as long as the typing chains are properly defined and consistent, the formalisation of application and supplementary typing chains has no real difference. Therefore, we can consider both working with several hierarchies, for which there might be several *Ecore* models at the top, or with several branches within the same hierarchy where there is only one *Ecore* model. The latter alternative can be achieved using the same techniques as the former, as long as some of our constraints are weakened, e.g., the tree shape (discussed in Section 2.1) that we impose on hierarchies or the single individual type (Definition 7) of each element in a hierarchy.

Figure 8 displays the hierarchy in Figure 1, but in it two different branches are combined within the same hierarchy, i.e., we specify two typing chains. The left-hand branch, in which models are connected by blue dashed arrows, represents the main typing chain and guides how we can consistently and precisely

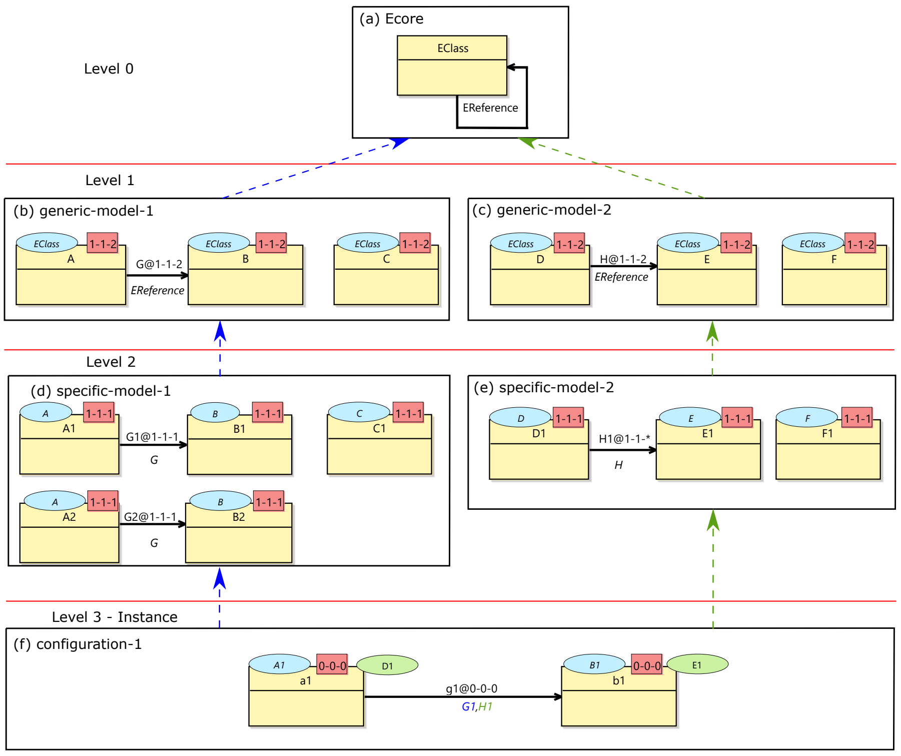


Figure 8: Multilevel hierarchy with two typing chains

type elements. As described above, we can then add extra typing chains, in this case, to our instance level, for example the one represented by the green dashed arrows (characterised by the right-hand branch). Once a new typing chain is incorporated, all the elements (both nodes and references) need to be extended with a new type. Then, these types can be used/modified by the modeller as it is done with the main type.

The model configuration-1 in Fig. 8(f) shows an example of how elements may be double-typed. One can see that node a1 has two types associated, A1 from the left-hand typing chain, its main type, and D1 from the right-hand typing chain, which adds additional information to the node. We have a similar situation with reference g1 and its two types G1 and H1.

Figure 9 compares the merge case exposed in Figure 7(b) with our approach for composition based on multiple typing chains. As already explained, a considerable drawback of the merge operation is that, once the merge is performed, the individuality of the elements that belonged to the different models prior the composition step is lost. Notice in Figure 9(b) that we do not carry any "physical" merge when a composed hierarchy or model is produced, but we can instantiate elements with more than one type. The hierarchies are left un-

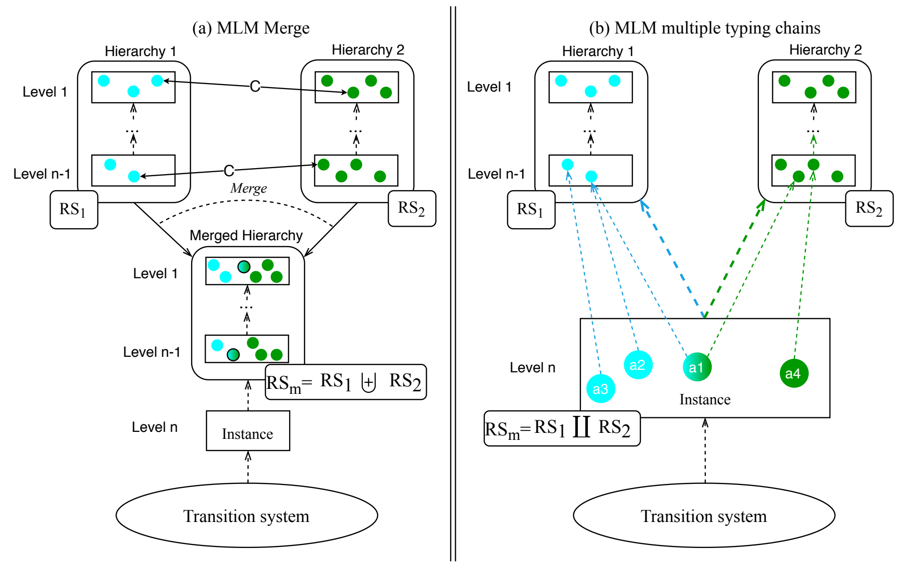


Figure 9: MLM merge combination vs our approach with multiple typing chains

touched, but the rules belonging to each hierarchy might be amalgamated to take into account a desired composed behaviour. Of course, we can preserve the "individual" nature by using just one of the types as shown in either a2, a3 or a4 elements in Figure 9(b). We discuss in section 3.2.2 how we achieve behaviour composition ($\mathsf{RS}_m = \mathsf{RS}_1 \coprod \mathsf{RS}_2$ at the bottom of Figure 9(b)).

The inclusion of an extra typing chain forces all the elements at the instance level to have an additional new type from the newly incorporated typing chain. Elements which do not get a specific type from the newly added typing chain will get a default typing; i.e., the type of the nodes is set to EClass (and arrows to EReference, respectively). Recall that this default typing to Ecore elements is independent on whether the new typing chain is contained in the same hierarchy (i.e., we use the same Ecore as a top most model G_0) or we use a completely new hierarchy. This is illustrated in Figure 10 which depicts a fragment of the hierarchy of Figure 9(b) but using typing arrows (formally $ty(e)$) instead of ellipses. We can see that the model *configuration-1* has two typing chains: a blue and a green one, in the same hierarchy. Note that, in a particular typing chain we omit the default typing to Ecore elements if other intermediate types exist (e.g., a2 has A1 in the blue branch, while it has only the default EClass in the green one). We describe the individual typing for each of the elements below; we denote $TC_x(e)$, with $x = 1, 2$, the corresponding individual typings of

20

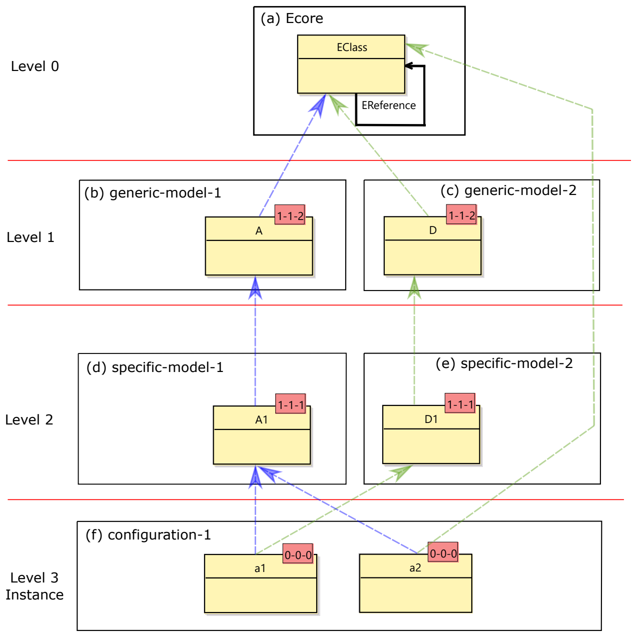


Figure 10: Typing chains to keep individuality of elements

the element e in typing chain x:

$$\begin{aligned}
TC_1(a1) \equiv a1 &\longmapsto ty_1(a1) = A1 \longmapsto ty_1^2(a1) = A \longmapsto ty_1^3(a1) = EClass\\
TC_2(a1) \equiv a1 &\longmapsto ty_2(a1) = D1 \longmapsto ty_2^2(a1) = D \longmapsto ty_2^3(a1) = EClass
\end{aligned}$$

$$\begin{aligned}
TC_1(a2) \equiv a2 &\longmapsto ty_1(a2) = A1 \longmapsto ty_1^2(a2) = A \longmapsto ty_1^3(a2) = EClass\\
TC_2(a2) \equiv a2 &\longmapsto ty_2(a2) = EClass
\end{aligned}$$

3.2.2. Amalgamation of MCMTs

In the previous section, we explained how we support composition of MLM models by multiple typing. In this section, we will explain composition of behaviour by the amalgamation of MCMT rules. The amalgamation of transformation rules has been widely discussed in the literature in the context of traditional (two-level) approaches [40, 41, 42, 43, 44]. In this paper, we study amalgamation in the MLM context and allow potentially conflicting rules to be amalgamated under certain constraints.

We are working with two MLM hierarchies or, as in the running example, with the composition of the two branches of Figure 8, each of them with its own set of MCMTs. The elements will appear double-typed at the instance level (for example, the situation described in Figure 8(f)). Thus, a key aspect is to also be able to amalgamate rules which only pertain to each branch of the hierarchy.

To illustrate the constructions, we will explain the process by amalgamating two MCMTs, one for each branch: Rule A (TR_A) for the left branch, which is the rule depicted in Figure 3 and shown again in Figure 11(a), together with Rule B (TR_B), which is a very similar rule for the right branch (Figure 11(b)).

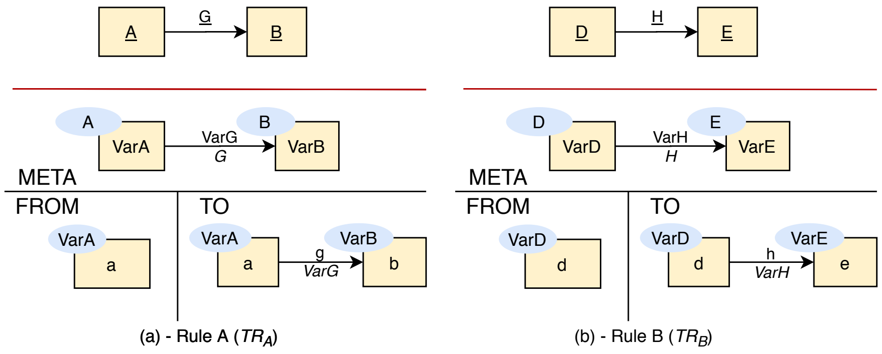


Figure 11: MCMT rules to be amalgamated: (a) Rule A affecting the left-hand branch and (b) Rule B affecting the right-hand branch

An essential step to achieve amalgamation (or, in general, composition) is the identification process where the elements that correspond to each other have to be identified. Most works in the literature use a so-called kernel rule to express correspondences between two or more rules [40, 41, 42, 43]. Also in our approach, we assume that the user provides the correspondences between elements in the rules which are to be amalgamated. That is, given Rule A $L_A \hookrightarrow I_A \hookleftarrow R_A$ and Rule B $L_B \hookrightarrow I_B \hookleftarrow R_B$, the correspondences provided by the user (L_0, I_0 and R_0) will be defined as a subrule TR_0 such that $TR_0 \hookrightarrow TR_A$ and $TR_0 \hookrightarrow TR_B$.

Definition 11 (Subrule)**.** *A rule $TR_0 \coloneqq L_0 \hookrightarrow I_0 \hookleftarrow R_0$ is a subrule of a rule $TR \coloneqq L \hookrightarrow I \hookleftarrow R$, written $TR_0 \hookrightarrow TR$, where there exist three inclusion graph morphisms $L_0 \hookrightarrow L$, $I_0 \hookrightarrow I$, and $R_0 \hookrightarrow R$, such that the following diagrams are commutative.*

$$\begin{tikzcd}
TR_0 \arrow[d, hook] & L_0 \arrow[r, hook] \arrow[d, hook] \arrow[dr, phantom, "="] & I_0 \arrow[d, hook] \arrow[dr, phantom, "="] & R_0 \arrow[l, hook'] \arrow[d, hook] \\
TR & L \arrow[r, hook] & I & R \arrow[l, hook']
\end{tikzcd}$$

This will give rise to three spans with inclusion graph morphisms: $L_A \hookleftarrow L_0 \hookrightarrow L_B$, $I_A \hookleftarrow I_0 \hookrightarrow I_B$ and $R_A \hookleftarrow R_0 \hookrightarrow R_B$. Recall that $L \sqsubseteq I$ and $R \sqsubseteq I$, hence, we can deduce R_0 and L_0 from I_0, meaning that in practise the user only needs to specify I_0.

Amalgamating TR_A and TR_B w.r.t. TR_0 means to combine the components of the rules so that we obtain a single rule TR_M such that $(L_M = L_A +_{L_0} L_B) \hookrightarrow$

$(I_M = I_A +_{I_0} I_B) \hookleftarrow (R_M = R_A +_{R_0} R_B)$. Again, we use pushout constructions, as a common practise, to obtain the components of TR_M. Below, we detail the construction of L_M as the pushout $L_A +_{L_0} L_B$ (see Figure 12). The same constructions will apply for the I and R components of the rules.

$$\begin{tikzcd}
 & L_0 \arrow[rrr, hook, "\lambda_0"] \arrow[dl, hook, "\mu_L"'] \arrow[ddr, hook, "\lambda_L"] & & & I_0 \arrow[dl, hook, "\mu_I"'] \arrow[ddr, hook, "\lambda_I"] & & & R_0 \arrow[lll, hook', "\rho_0"'] \arrow[dl, hook, "\mu_R"'] \arrow[ddr, hook, "\lambda_R"] & \\
L_A \arrow[rrr, hook, "\lambda_A"] \arrow[ddr, "\varsigma_L"'] \arrow[drr, phantom, "\text{PO}"] & & & I_A \arrow[ddr, "\varsigma_I"'] \arrow[drr, phantom, "\text{PO}"] & & & R_A \arrow[lll, hook', "\rho_A"'] \arrow[ddr, "\varsigma_R"'] \arrow[drr, phantom, "\text{PO}"] & & \\
 & & L_B \arrow[rrr, hook, "\lambda_B"'] \arrow[dl, "\delta_L"'] & & & I_B \arrow[dl, "\delta_I"'] & & & R_B \arrow[lll, hook', "\rho_B"] \arrow[dl, "\delta_R"'] \\
 & L_M \arrow[rrr, hook, "\lambda_M"] & & & I_M & & & R_M \arrow[lll, hook', "\rho_M"'] &
\end{tikzcd}$$

Figure 12: Amalgamated rule construction with pushouts

However, since L_A and L_B (and, respectively, I_A, I_B, R_A and R_B) have different multilevel types, we would need to unify the types by defining default types for each of the elements in the other hierarchies, i.e., all L_A elements would have the default type (EClass/EReference) from the typing chain $\mathcal{MM_B} = (\overline{MM_B}, n_b, \tau^{MM_B})$, while all L_B elements will have the default types from the typing chain $\mathcal{MM_A} = (\overline{MM_A}, n_a, \tau^{MM_A})$ (and again, the same for I_A, I_B, R_A and R_B). Furthermore, L_0 would have the default types in both chains.

We illustrate this in Figure 13. On the left-hand side, we break down the $L_A +_{L_0} L_B$ pushout resulting in L_M together with their respective typing chains. As described above, a is typed over $\mathcal{MM_A}$ (VarA, A, EClass) and over $\mathcal{MM_B}$ (EClass), d is typed over $\mathcal{MM_B}$ (VarD, D, EClass) and by EClass over $\mathcal{MM_A}$, a ≡ d in L_0 is only double-typed by EClass in each of the typing chains, and the resulting ad in L_M is typed over $\mathcal{MM_A}$ (via $\sigma^{L_M^A}$) and $\mathcal{MM_B}$ (via $\sigma^{L_M^B}$) as shown in the right-hand side of Figure 13.

Expressed in terms of inclusion chains, the aforementioned typing relations mean that $L_{0,0}^A = L_{0,0}^B = L_{0,0} = L_0$, where $L_{0,0}^A$ is the part of L_0 which is typed by $MM_{A,0}$ (see Lemma 1). The rest of the levels $L_{0,i}^A$, with $0 < i \leq n_a$ will be empty since L_0 has only default types in the two rules' hierarchies. These types are reflected by the two light thin arrows from L_0 to the two EClasses in $\mathcal{MM_A}$ and $\mathcal{MM_B}$ in Figure 13, respectively. Similarly, we have $L_{A,0}^A = L_{A,0}^B = L_{A,0}$ and $L_{B,0}^A = L_{B,0}^B = L_{B,0}$. The levels (except for 0) of the inclusion chains $\mathcal{L_A}$ (resp. $\mathcal{L_B}$) along $\sigma^{L_A^A} : L_A \Rightarrow \mathcal{MM_A}$ (resp. $\sigma^{L_B^B} : L_B \Rightarrow \mathcal{MM_B}$) will be constructed according to Lemma 1. Moreover, the default levels (except for 0) of the inclusion chains $\mathcal{L_A}$ (resp. $\mathcal{L_B}$) along $\sigma^{L_A^B} : L_A \Rightarrow \mathcal{MM_B}$ (resp. $\sigma^{L_B^A} : L_B \Rightarrow \mathcal{MM_A}$) will be empty. Having these typing chains, we apply level-wise pushouts as described in Section 2.2 [30].

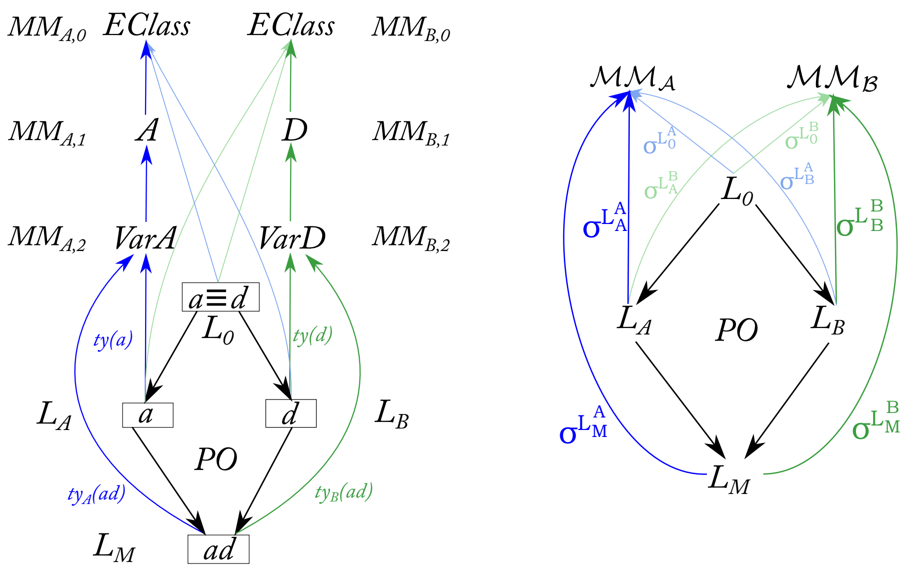


Figure 13: L_M with typing chains as result of pushout of L_A, L_B modulo L_0

The results of these level-wise pushouts would be two inclusion chains:

- $\mathcal{L}^{\mathcal{A}}_{\mathcal{M}} = (\overline{L^A_M}, n_a, \tau^{L^A_M})$ with $\sigma^{L^A_M} : L^A_M \Rightarrow \mathcal{MM}_{\mathcal{A}}$: The levels $L^A_{M,i}$ for all $0 \le i \le n_a$ of the inclusion chain will be produced by the pushouts of the spans $L^A_{A,i} \hookleftarrow L^A_{0,i} \hookrightarrow L^A_{B,i}$.

- $\mathcal{L}^{\mathcal{B}}_{\mathcal{M}} = (\overline{L^B_M}, n_b, \tau^{L^B_M})$ with $\sigma^{L^B_M} : L^B_M \Rightarrow \mathcal{MM}_{\mathcal{B}}$: The levels $L^B_{M,i}$ for all $0 < i \le n_b$ of the inclusion chain will be produced by the pushouts of the spans $L^B_{A,i} \hookleftarrow L^B_{0,i} \hookrightarrow L^B_{B,i}$.

Figure 14 illustrates how the levels $L^A_{M,0}$ and $L^A_{M,1}$ are constructed (the relations between the levels are omitted to simplify the diagrams). The other levels, as well as the pushouts with respect to $\mathcal{MM}_{\mathcal{B}}$, are constructed analogously. $L^A_{M,0}$ and $L^A_{M,1}$ are obtained by the pushouts of the spans $L^A_{A,0} \hookleftarrow L^A_{0,0} \hookrightarrow L^A_{B,0}$ and $L^A_{A,1} \hookleftarrow L^A_{0,1} \hookrightarrow L^A_{B,1}$, respectively. The graphs $L^A_{0,1}$ and $L^A_{B,1}$ will be empty since the inclusion chain is constructed with respect to $\mathcal{MM}_{\mathcal{A}}$. This is because the elements of L_0 and L_B have only the default types in $\mathcal{MM}_{\mathcal{A}}$, hence only level 0 of these inclusion chains are none-empty such that $L_0 = L_{0,0}$ and $L_B = L_{B,0}$. The construction of the two first levels for the rules TR_A and TR_B from the running example is shown in Figure 15. Notice that d is neither identified in $L^A_{0,1}$ nor in $L^A_{B,1}$, as it only has the default EClass type w.r.t. $\mathcal{MM}_{\mathcal{A}}$, located in level 0. Then, in $L^A_{M,1}$ we only have a, which is typed by A in $MM_{A,1}$.

To summarise, the result of the amalgamation process is an amalgamated rule where each element has two types. For the running example, the result of

24

$$\begin{tikzcd}
 & L^A_{A,0} \arrow[dr] \arrow[drr, bend left] & & \\
L^A_{0,0} \arrow[ur] \arrow[dr] \arrow[urrr, bend left=40] & PO & L^A_{M,0} \arrow[r] & MM_{A,0} \\
 & L^A_{B,0} \arrow[ur] \arrow[urr, bend right] & & \\
 & L^A_{A,1} \arrow[dr] \arrow[drr, bend left] & & \\
L^A_{0,1}=\emptyset \arrow[ur] \arrow[dr] & PO & L^A_{M,1} \arrow[r] & MM_{A,1} \\
 & L^A_{B,1}=\emptyset \arrow[ur] & &
\end{tikzcd}$$

Figure 14: Levels 0, 1 of the inclusion chain $\mathcal{L}_{\mathcal{M}}$

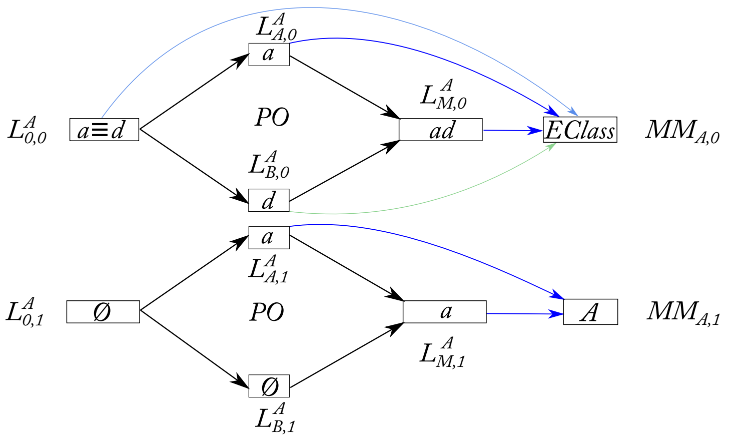


Figure 15: Levels 0, 1 of the inclusion chain $\mathcal{L}_{\mathcal{M}}$ for the rules TR_A and TR_B

amalgamating TR_A and TR_B in Figure 11 is the rule TR_M which is depicted in Figure 16 as a co-span and in Figure 17 in the MultEcore syntax.

3.2.3. Amalgamation cases

If we inspect the constructions described in Section 3.2.2, we can observe several amalgamation cases depending on how TR_A and TR_B are related by TR_0. Table 1 shows a summary of the cases that we contemplate, which are listed below (note that one can see in the *Amalgamation* columns which elements are identified, as the names are concatenated):

Case 1 : TR_A adds, TR_B adds and $I_0 = L_0$, i.e., added elements are not identified (only `a` is identified with `d` which was already existing in L_0).

Case 2 : TR_A adds, TR_B adds and $L_0 \sqsubset I_0$, i.e. the elements newly added by each of the rules are identified between them (for example in this case, `b`

25

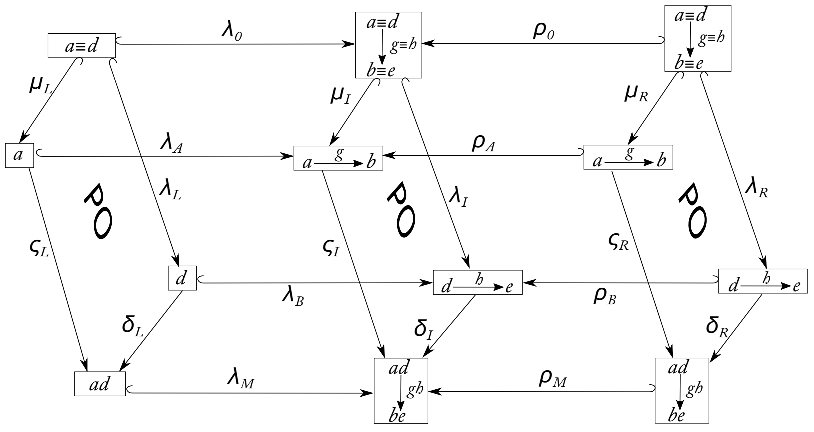


Figure 16: Amalgamation construction application of the situation depicted in Figure 17

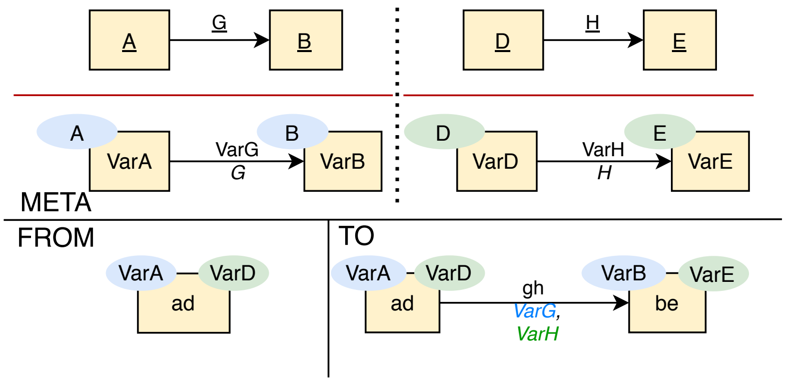


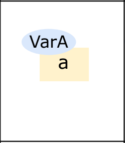
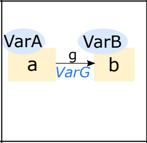
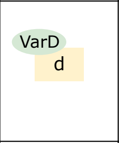
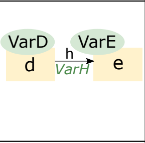
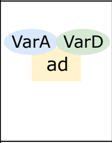
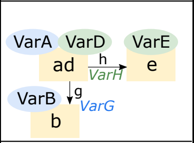
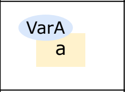
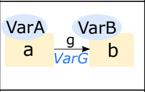
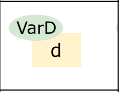
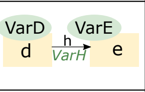
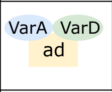
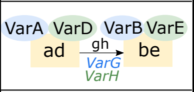
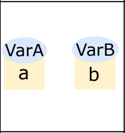
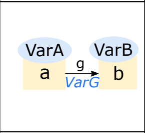
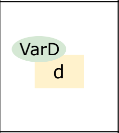
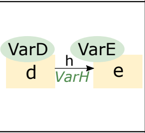
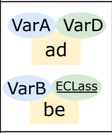
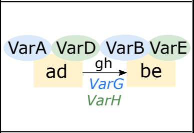
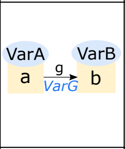
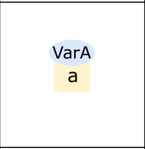
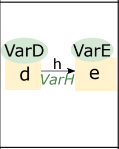
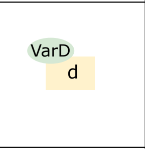
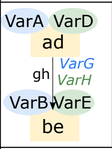
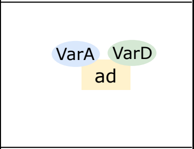
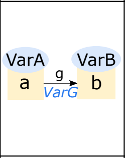
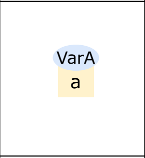
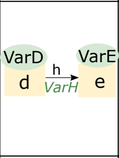
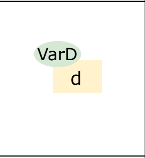
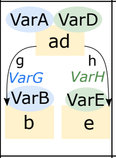
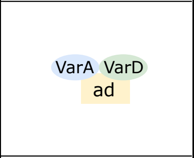
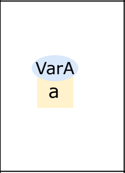
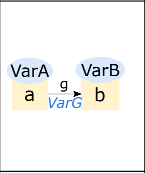
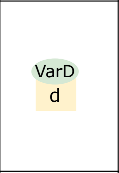
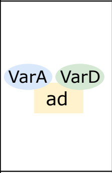
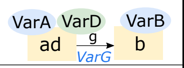
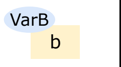
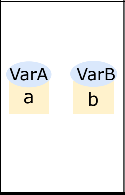
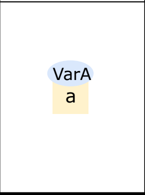
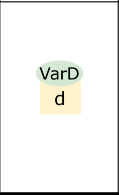
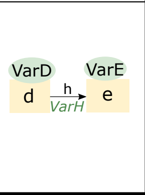
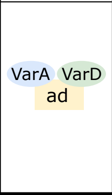
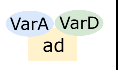
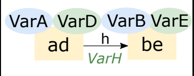
Figure 17: Amalgamated rule TR_M as result of combining TR_A and TR_B

is identified with e and g with h, which are all newly added). This case represents the example shown along Section 3.2.2

Case 3 : TR_A adds, TR_B adds, $L_0 \sqsubset I_0$ and either $(I_0 \setminus L_0) \cap L_A \neq \varnothing$ or $(I_0 \setminus L_0)$ $\cap\, L_B \neq \varnothing$, i.e., newly added elements by TR_A are identified with elements which are in L_B, or vice versa. This is a special case since, for as, b is identified with e in I_0, but e does not exist in L_0. Therefore, as hinted in the constructions shown in Section 3.2.2, we need to constrain the match of L_M in the source graph S by forcing the missing type for be to be directly — i.e., not transitively — EClass. In abuse of notation, we indicate with underlined text in the type rather than in the element (as we do for constants) that the constrained typing relation must match exactly one typing relation in the target hierarchy, instead of a potential series of transitive typing relations.

26

Table 1: Amalgamation cases

CASE	Rule A (TR_A)		Rule B (TR_B)		Amalg. (TR_M)	
	L_A	R_A	L_B	R_B	L_M	R_M
1						
2						
3						
4						
5						
6				$\emptyset$		Pr. TR_A  Pr. TR_B 
7						Pr. TR_A  Pr. TR_B 

From this point, the cases which include deletion of elements might cause a general dangling arrow problem, which has to be solved. One solution is to get rid of the dangling arrows using a special graph minus operator as explained below. Alternatively, we could notify the user about the dangling arrows and ask for user intervention as it is done, for instance, in version control systems (see [45]).

Case 4 : TR_A deletes and TR_B deletes, where $L_0 = I_0$ (i.e., there are no new identifications except for the ones in L_0), $L_M = I_M$ (no additions), and $I_A \setminus R_A = I_B \setminus R_B$ (identified/same elements are deleted). Note, if the only deleted elements are those that have been identified, we have $I_0 \sqsupseteq I_M \setminus R_M$.

Case 5 : TR_A deletes, TR_B deletes, $I_0 = L_0$, $L_M = I_M$, $I_A \setminus R_A \neq I_B \setminus R_B$ (different elements are deleted) and $R_M \sqsupseteq I_0$ (all the identified elements are preserved).

We will now analyse other cases involving deletion which could be covered if R_M is created by pushout of $R_A \hookleftarrow R_0 \hookrightarrow R_B$. However, if we use such a mechanism to construct R_M, we would lose the effect of deletion, and certain conflicts might just disappear. For example, if TR_A deletes an element while TR_B keeps it, the element would be kept. Obviously, a potential dangling arrow problem would also disappear since a deleted node a which is identified in L_0 or I_0 would be kept if it is preserved by the rule which uses a as source or target of an arrow. Therefore we introduce two priority formulae below to prioritise the effect of one of the rules depending on the user's choice (the calculation of R_M, i.e., the square on the far right of Figure 12, would be done via the formulae below).

$$\begin{aligned} \text{Priority in } TR_A: &\quad R_M = R_A \cup (R_B \overset{*}{-} (I_0 \cap R_B)) \\ \text{Priority in } TR_B: &\quad R_M = R_B \cup (R_A \overset{*}{-} (I_0 \cap R_A)) \end{aligned}$$

We define $\overset{*}{-}$ as a graph minus operation that removes any dangling arrow that could be left by the usual graph minus operation.

To illustrate an application of priorities, let us consider the example shown in Figure 18 where we have two rules: TR_A:= *Add and connect* and TR_B:= *Delete node*. The latter was originally conceived to be applied to the right-hand branch of the hierarchy (*specific-model-2, generic-model-2, Ecore*) in Figure 8. In the case of the *Delete node* rule, and following the same logic as explained for the *Add and connect* rule, any match in our instance of the variable d placed in the FROM block, whose type is VarD located at the second level of the META block, and which is typed by the constant D located at the first level of the META block, takes the instance to a new state where the matched element is removed. These rules are conflicting in the sense that the user has identified a with d and, while TR_A adds a new arrow g to a, TR_B deletes the element d. However, as mentioned, applying our standard pushout construction would produce a TR_M in which the affect of the deletion in R_M disappears.

Depending on which rule the user wants to prioritise, the corresponding formula needs to be applied. First, the user has to provide I_0 with the identification. In this case, I_0 only identifies a with d (a ≡ d). Such an identification

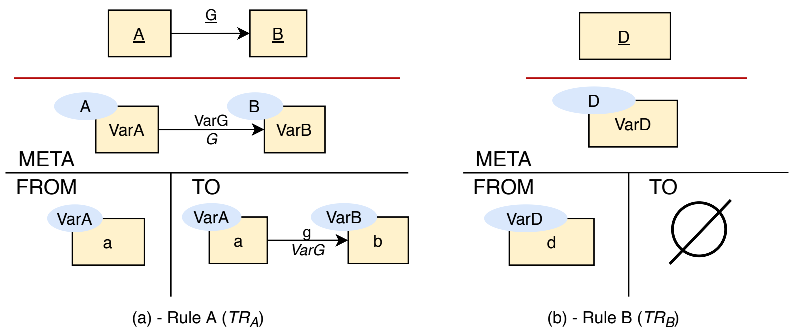


Figure 18: MCMT rules to be amalgamated: (a) Rule A affecting the left branch and (b) Rule B affecting the right branch

indicates us that a, d, or ad appearances in the formula must be treated as same element. If the prioritisation falls on TR_A, we have:

$$R_M = (a \xrightarrow{g} b) \cup (\varnothing \overset{*}{-} (ad \cap \varnothing))$$
$$R_M = (a \xrightarrow{g} b)$$

If the prioritisation is given to TR_B the result is:

$$R_M = \varnothing \cup ((a \xrightarrow{g} b) \overset{*}{-} (ad \cap (a \xrightarrow{g} b))$$
$$R_M = \varnothing \cup ((a \xrightarrow{g} b) \overset{*}{-} a)$$
$$R_M = b$$

Observe how a normal minus operation would keep a dangling arrow pointing to b, while the $\overset{*}{-}$ operation removes also the arrow. We graphically show both

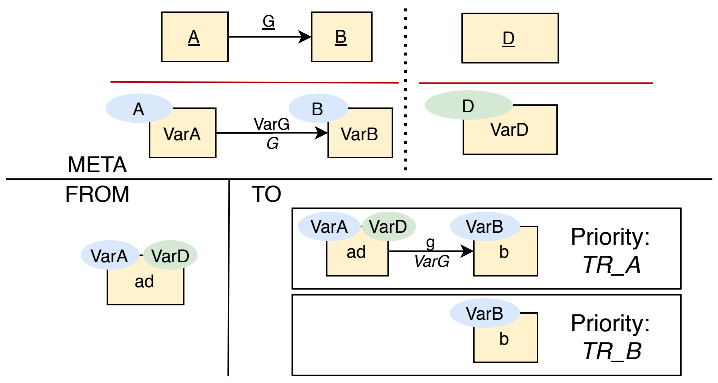


Figure 19: Amalgamated rule TR_M as result of combining TR_A and TR_B where R_M has been calculated with the priority formulae

29

possible results in Figure 19, where the TO block depicts the two alternatives depending on the priority.

Case 6 : One of the rules adds while the other deletes, for instance, TR_A adds something to an element while TR_B deletes that element. This is the case depicted above and shown in Figures 18 and 19 where R_M is given by prioritisation on one of the rules.

Case 7 : This case covers potential combinations of some of the cases afore discussed. There might be several additions and/or deletions at the same time and, therefore, conflicts that would require prioritisation.

3.2.4. Amalgamated rule application

The last step, once the amalgamated multilevel double-typed rule is constructed, consists of its application into the composed multilevel hierarchy. Note that the construction follows the same reasoning as for single multilevel typed rules (detailed in Section 2.2). The complete construction for the amalgamated rule application is depicted in Figure 20.

As we discussed in Section 3.2.3, the calculation of R_M might not be done by the pushout but with the priority formulae, so that we mark the right hand pushout with $*$. We have the two typing chains $\mathcal{MM}_\mathcal{A} = (\overline{MM_A}, n_a, \tau^{MM_A})$ and $\mathcal{MM}_\mathcal{B} = (\overline{MM_B}, n_b, \tau^{MM_B})$ over which the double-typed MCMT rule is defined. The multilevel double-typed rule is given by the four components (L_0, L_A, L_B and L_M for L and respectively for I and R) and their multilevel typings over the two typing chains $\mathcal{MM}_\mathcal{A}$ and $\mathcal{MM}_\mathcal{B}$ such that $\sigma^{L_A^A} : L_A \Rightarrow \mathcal{MM}_\mathcal{A}$ and $\sigma^{L_B^B} : L_B \Rightarrow \mathcal{MM}_\mathcal{B}$, $\sigma^{I_A^A} : I_A \Rightarrow \mathcal{MM}_\mathcal{A}$ and $\sigma^{I_B^B} : I_B \Rightarrow \mathcal{MM}_\mathcal{B}$ and $\sigma^{R_A^A} : R_A \Rightarrow \mathcal{MM}_\mathcal{A}$ and $\sigma^{R_B^B} : R_B \Rightarrow \mathcal{MM}_\mathcal{B}$. Then, we have $\sigma^{L_M^A} : L_M \Rightarrow \mathcal{MM}_\mathcal{A}$, $\sigma^{L_M^B} : L_M \Rightarrow \mathcal{MM}_\mathcal{B}$, $\sigma^{I_M^A} : I_M \Rightarrow \mathcal{MM}_\mathcal{A}$, $\sigma^{I_M^B} : I_M \Rightarrow \mathcal{MM}_\mathcal{B}$, $\sigma^{R_M^A} : R_M \Rightarrow \mathcal{MM}_\mathcal{A}$ and $\sigma^{R_M^B} : R_M \Rightarrow \mathcal{MM}_\mathcal{B}$.

In the multilevel typed setting all the instance graphs S, D and T are multilevel double-typed over another two typing chains $\mathcal{TG}_\mathcal{A} = (\overline{TG_A}, m_a, \tau^{TG_A})$ and $\mathcal{TG}_\mathcal{B} = (\overline{TG_B}, m_b, \tau^{TG_B})$, the instance typing chains. A **match** of the left-hand side $(L_M,\, \sigma^{L_M^A}, \sigma^{L_M^B})$ of the multilevel double-typed rule into a multilevel double-typed instance graph $(S,\, \sigma^{S^A}, \sigma^{S^B})$ is given by a graph homomorphism $\mu_M : L_M \to S$ together with the corresponding typing chain morphisms (β_A, f_A) and (β_B, f_B) where $\beta_A = \beta_{A_i} : MM_{A_i} \to \mathcal{TG}_{\mathcal{A}\,f_A(i)} \,|\, i \in [n_a]$ and $\beta_B = \beta_{B_i} : MM_{B_i} \to \mathcal{TG}_{\mathcal{B}\,f_B(i)} \,|\, i \in [n_b]$, respectively.

Furthermore, $\mu_M : L_M \to S$ has to be compatible with the multilevel typings $\sigma^{L_M^A} : L_M \Rightarrow \mathcal{MM}_\mathcal{A}$ and $\sigma^{L_M^B} : L_M \Rightarrow \mathcal{MM}_\mathcal{B}$, $\sigma^{S^A} : S \Rightarrow \mathcal{TG}_\mathcal{A}$ and $\sigma^{S^B} : S \Rightarrow \mathcal{TG}_\mathcal{B}$ and, finally, with the typing chain morphisms $(\beta_A, f_A) : \mathcal{MM}_\mathcal{A} \to \mathcal{TG}_\mathcal{A}$ and $(\beta_B, f_B) : \mathcal{MM}_\mathcal{B} \to \mathcal{TG}_\mathcal{B}$.

We construct the pushout and then the final pullback complement of the underlying graph homomorphisms in the category **Graph** as shown at the bottom of Figure 20. The type compatibility conditions for the multilevel double-typed rule as well as for the multilevel typed match should ensure that we obtain, in a canonical way, multilevel typings $\sigma^{D^A} : D \Rightarrow \mathcal{TG}_\mathcal{A}$ and $\sigma^{D^B} : D \Rightarrow \mathcal{TG}_\mathcal{B}$,

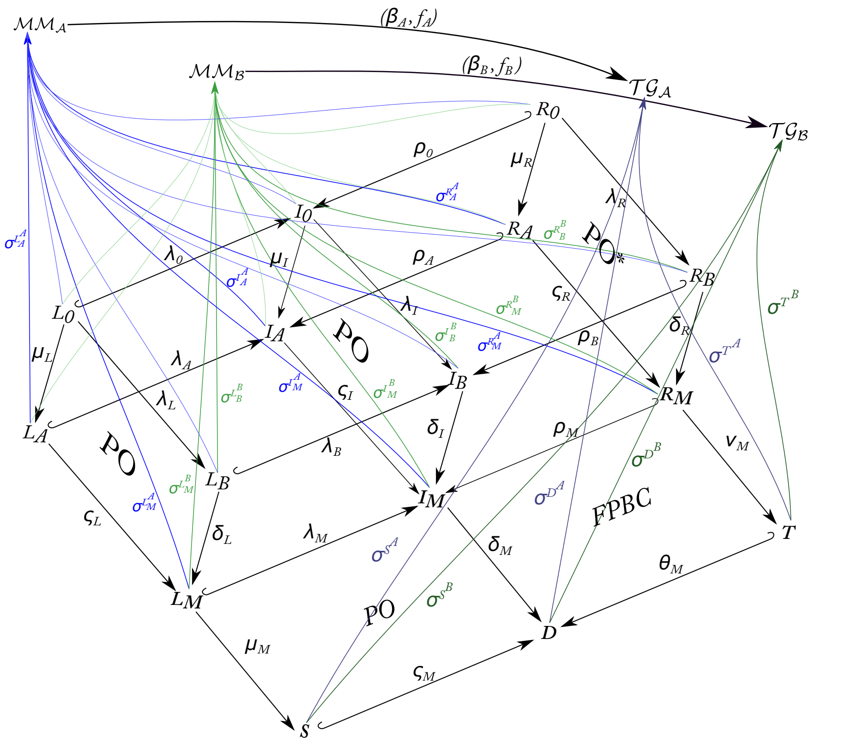


Figure 20: Amalgamated rule application construction

$\sigma^{T^A} : T \Rightarrow \mathcal{TG}_\mathcal{A}$ and $\sigma^{T^B} : T \Rightarrow \mathcal{TG}_\mathcal{B}$ of the constructed graphs such that the constructed graph homomorphisms $\varsigma_M : S \hookrightarrow D$, $\delta_M : I_M \rightarrow D$, $\theta_M : T \hookrightarrow D$ and $\nu_M : R_M \rightarrow T$ are type compatible.

4. Case study

The capability to perform composition of structure and operational semantics takes the construction of DSMLs to a next step. Modelling a system often involves the consideration of several perspectives that describe different aspects of the system. In the case study that we present in this section, the main aspect of the system consists of a DSML defined as a multilevel hierarchy for the management and distribution of process resources in a company. This is the application hierarchy of the case study, called *process management*. The *process management* hierarchy version we present in this paper is a fragment of the hierarchy presented in the MULTI 2019 workshop, as our solution [27] to the MULTI Process challenge [46, 47]. Therefore, all the modelling decisions affecting the complete hierarchy (illustrated in Appendix A) were made to fulfil the requirements of the challenge. The second DSML is described in an independent multilevel hierarchy that captures certain notions related to human beings in general (e.g., stamina). This second hierarchy acts as the supplementary one in our case study, and it is called the *human-being* hierarchy. By applying our approach we observe that composition can be achieved in a natural and modular way. The composition of structure can be done by double typing elements, while the MCMTs can be composed by applying the constructions introduced in Section 3.2.

4.1. The process management hierarchy

This hierarchy represents the domain of process management, where the modeller is interested in a complete description of a language that includes the specification of particular occurrences (i.e., "processes" = "processes instances", "tasks" = "task occurrences") and universal kinds of occurrences ("process definitions", "task types") and relations to actor types and artefact types. Our

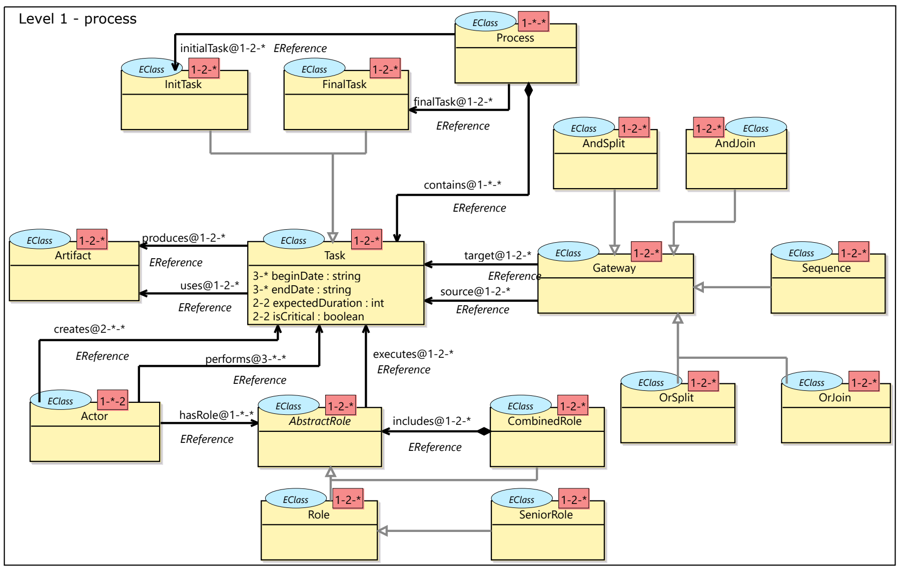


Figure 21: Process management model

original solution ([27]) presented models not only related to the general management of processes but also branches for specific processes in the domains of software engineering and insurance. For the sake of simplicity, we focus only on the software engineering branch as it suffices to illustrate our composition approach.

4.1.1. Structure of the process management hierarchy

The *process* model depicted in Figure 21 is located in the first level (we omit *Ecore*, which lives above *process* model) of the hierarchy and contains the concepts concerning universal processes. This includes *process types*, *task types*, *artefact types*, and *actors*. The composition relation named contains between

32

Process and Task models that a process has one or more tasks. Task has some attributes to model the duration, starting and ending day, and whether it is critical or not. Actors may have multiple roles, which is captured by the reference hasRole between Actor and AbstractRole. We use for roles the traditional object-oriented Composite pattern [48] and define AbstractRole as an abstract node (italic font in the name). A special type of role to designate a SeniorRole is also defined. Roles can have assigned kind of tasks whose instances can execute. Also, each actor can either create or perform tasks. Finally, the two references, produces and uses, from Task to Artifact, capture that tasks can both use and produce artefacts. Ordering constraints between task types are established through Gateways, which may be Sequence, OrSplit, OrJoin, AndSplit and AndJoin.

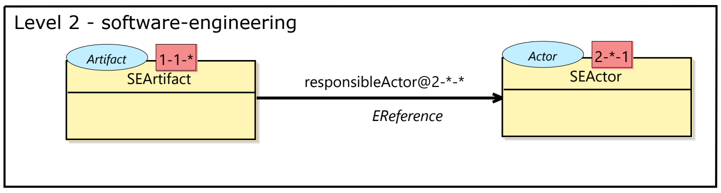


Figure 22: Software engineering process model

The model *software-engineering* (in level 2) in Figure 22 captures specialisations that affect the software engineering domain. For instance, that each soft-

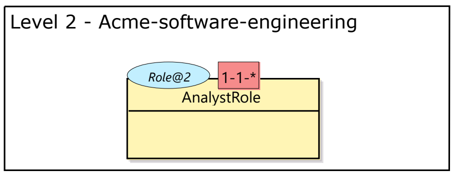


Figure 23: Acme software engineering process model excerpt

ware engineering artefact (SEArtifact node has as type Artifact from the *process* model in Figure 21) must have assigned one responsible software engineering actor.

The *Acme-software-engineering* model describes a concrete modelling language for the Acme company, and characterises how the working flow is going to be, which roles are allowed to execute certain types of tasks, which artefacts are produced, and so on. Figure 23 shows the excerpt of this model that is needed for the current case study — the entire model is depicted in Figure A.41(c). In this excerpt, we find AnalystRole class of type Role. Note that @2 is added to it type as it is located two levels above (at *process* model in Figure 21).

The lowest level of the *process management* hierarchy contains the instance model (called *Acme-configuration*) and it is shown in Figure 24. It depicts a very simple initial model with Alex as a software engineering actor (SEActor)

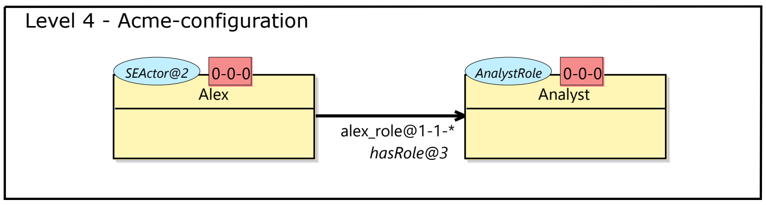


Figure 24: Acme initial configuration at the instance level

which has associated an Analyst role (of type AnalystRole).

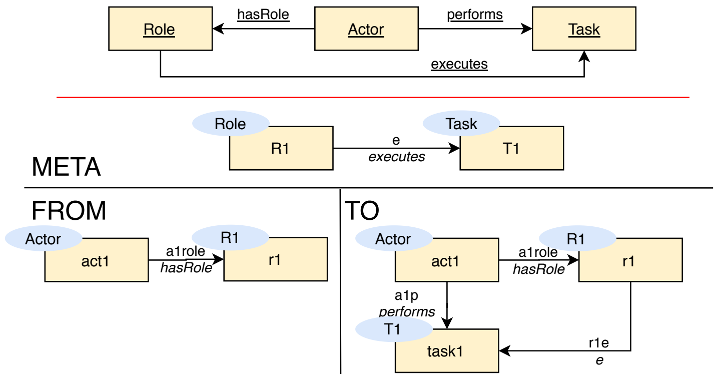


Figure 25: Rule *Create Task*: It creates a specific task associated to a concrete actor whose role allows the execution of such kind of tasks

4.1.2. MCMTs for the process management hierarchy

The dynamics of processes is modelled by MCMTs, which describe the different actions that may occur in the system. We show here three of these rules for the *process management* hierarchy that illustrate their use, and will serve us to manifest their combination with rules in the second hierarchy.

The first rule, called *Create Task*, is shown in Figure 25. Given an actor act1 with a role r1 of some type R1 via a1role, the rule assigns a new task of the right type to it. The role specified in the level 2 of the META block will constrain the task that such role can execute. In addition, the model at the higher level will similarly constrain the type of task that the actor can perform and its role execute.

The second rule, named *Produce Artefact*, is depicted in Figure 26. If an actor act1 and a task task1 he is performing (indicated by the a1p reference) are found, the rule creates an artefact ar1 related both to the actor act1 via r (typed by responsibleActor) and to the task task1 via t1pr.

The third rule that applies to the process management multilevel hierarchy, named *Delete task* and illustrated in Figure 27, is meant to delete a task that an actor is performing. Recall that rule levels are not expected to match consecutive

34

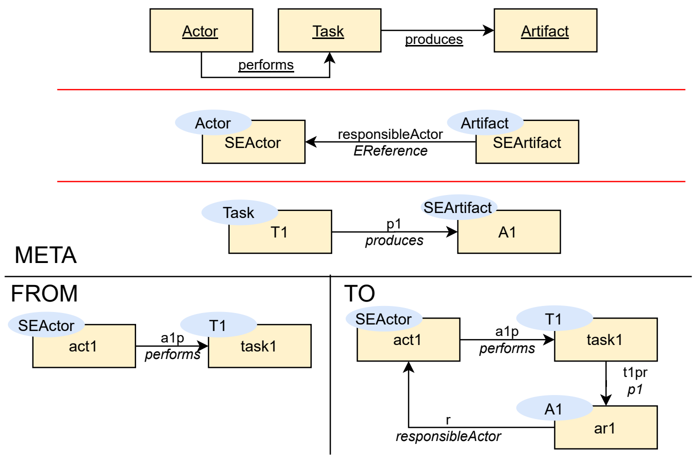


Figure 26: Rule *Produce Artefact*: It creates an artefact related to the task that produces it and the actor responsible for it

levels in the hierarchy on which they are defined. In this case, the META model would match to elements located at level 1 of the hierarchy (Figure 21), while the FROM and TO parts would match at the instance level placed at level 4 (Figure 24). This flexibility is specified in Condition 1.

4.2. The human-being hierarchy

In the *human-being* hierarchy we tackle different aspects inherently related to the human factor of the system.

4.2.1. Structure of the human-being hierarchy

This multilevel hierarchy is depicted in Figure 28. The model represented in Figure 28(a) captures very general human being notions, such as that a human (Human node) can do (does relation) multiple activities (Activity node). Furthermore, a human has a stamina level which is represented as an Integer (int), and an activity can have an impact on a human's stamina. These two characteristics are expressed via attributes in the respective nodes.

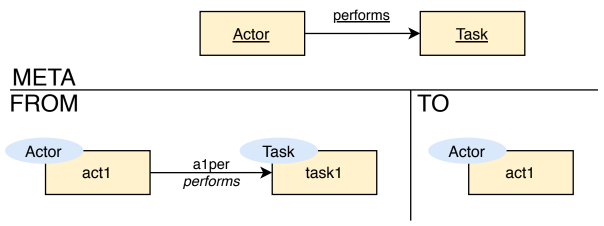


Figure 27: Rule *Delete Task*: It deletes a task an actor is performing

35

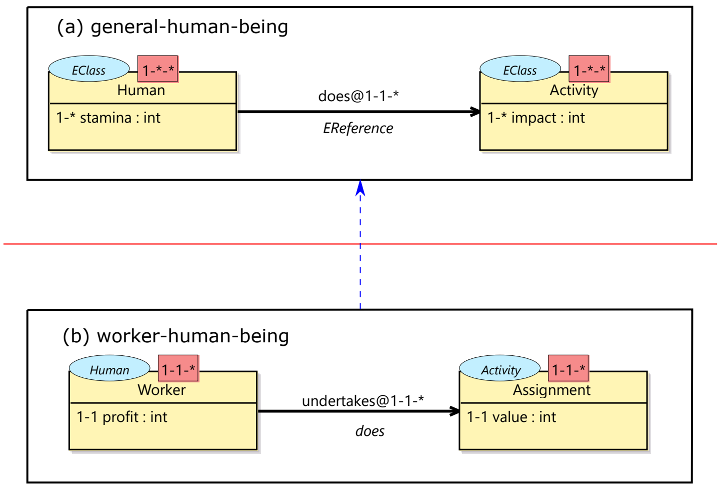


Figure 28: Human-being multilevel hierarchy

To give an example of refinement, we define in Figure 28(b) a model that captures concepts for the domain of working human beings. Note that we could add other models in here at the same level to capture other areas, such as students, retired people, etc. Worker, undertakes and Assignment have, as types, Human, does and Activity, respectively. Additionally in this level, two more attributes are added that only concern the worker domain. The profit attribute (defined in Worker) can be understood as the income that a worker obtains. And value, specified in the Assignment node, is the benefit that completing the assignment provides.

4.2.2. MCMTs for the human-being hierarchy

As we did for the process management hierarchy (Section 4.1.2), behaviour here is also described using MCMTs. We provide two MCMT rules for this hierarchy.

The first rule is called *Undertake activity* and it is shown in Figure 29. It connects a worker work1 and an assignment as1. Attributes are also modified in this rule. In the FROM block, s, p, i and v would capture values in the model for the stamina and profit for work1 and the impact and value for as1, respectively, during the matching process. In the TO block, apart from connecting them via u (typed by undertakes) reference, the attributes on the worker are modified: stamina in work1 gets decreased by the amount that was matched to the impact from the assignment as1 but the profit on the worker work1 gets increased by the amount specified in the value attribute in the assignment as1. Intuitively, a worker that is undertaking an activity gets income at the cost of getting more tired.

A second rule, named Finish Activity, is illustrated in Figure 30. Unlike the previous rule which is defined in the domain of worker human beings, this one

36

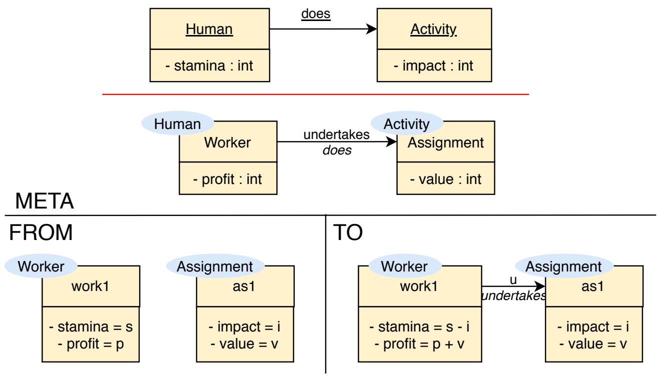


Figure 29: Rule *Undertake activity*: It connects a worker with the assignment being performed and updates its attributes

applies to human beings in general. The application of this rule finds a match in the model where a human human1 connected to an activity act1 via d and removes such a reference.

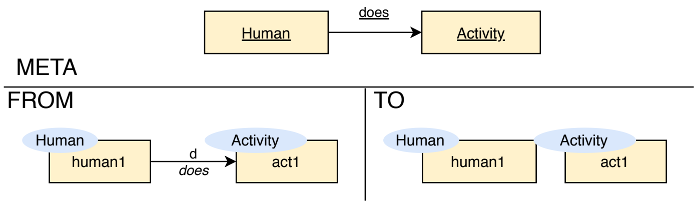


Figure 30: Rule *Finish Activity*: It removes the link between a human being and the activity he was performing

4.3. Multilevel hierarchies combination

A modeller working on a concrete design of the processes of the Acme company (specific actors, tasks, artefacts, etc.) might find useful to complement that given scenario with additional aspects, such as those described in the *human-being* multilevel hierarchy (Figure 28). Through our approach one can put together different perspectives, while there still exists a separation (via typing chains) that can be analysed either together or separately.

For instance, observe the model *Acme-configuration-composed* depicted in Figure 31 where we incorporate the *human-being* multilevel hierarchy (Figure 28) as a supplementary typing chain to reason about some elements defined on it. We can, for example, give to Alex the new type Worker and keep the SEActor type. Analogously, we can instantiate the attribute stamina, which comes from the *worker-human-being* model (Figure 28(b)), with the value 3.

To give a full perspective of how the two hierarchies are put together and how elements at instance level can make use of them, we provide selected parts

37

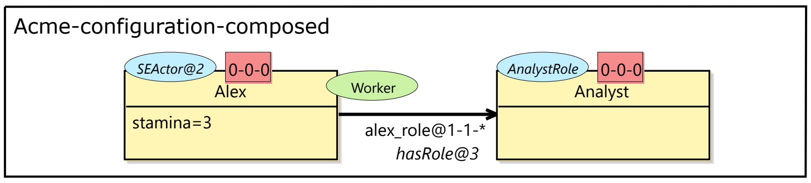


Figure 31: Instance model of Acme software engineering company including *human-being* hierarchy

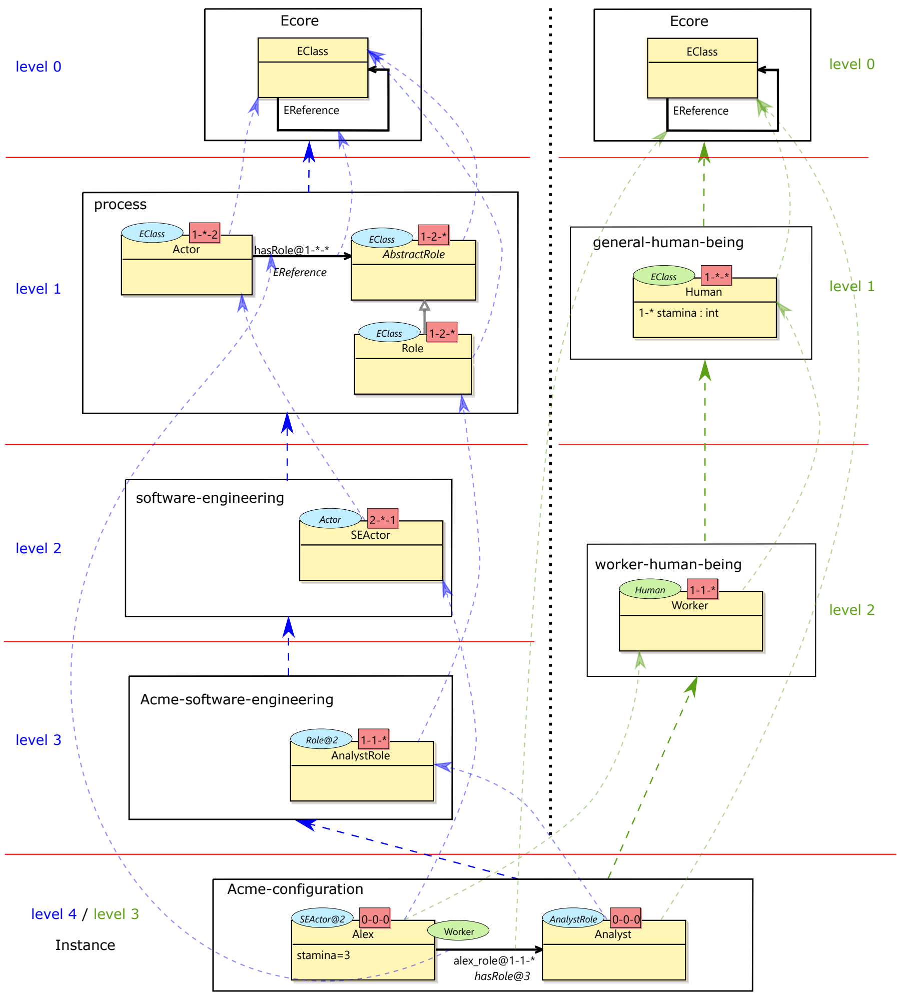


Figure 32: Selected parts of *process* and *human-being* hierarchies creating a composed multi-level hierarchy

38

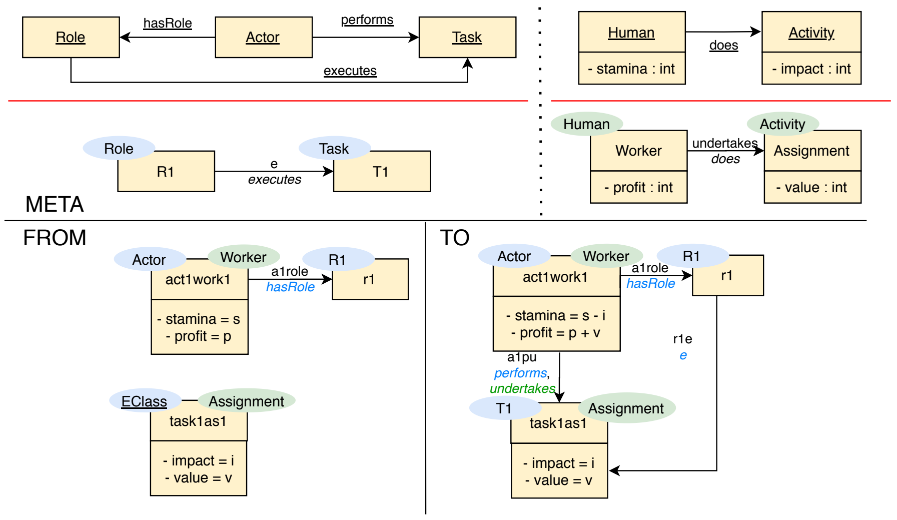


Figure 33: First rule amalgamation: It combines *Create Task* rule from the *process* hierarchy with *Undertake Activity* rule from the *human-being* hierarchy

of each of the models and illustrate them in Figure 32, where one can observe the typing chains for each hierarchy. Note that the model shown in Figure 31 is located at level 4 / level 3 - Instance in Figure 32. Each of the types belonging to each of the hierarchies can be precisely spotted in its corresponding typing chain up to the topmost model. Firstly, Alex is typed by SEActor. Note that the @2 means that SEActor is located at level: Alex's level (level 4) minus 2, i.e., at level 2 — in the *software-engineering* model. Then SEActor's type is Actor located at level 1 which finally leads us to EClass defined at level 0. Secondly, Alex's second type is Worker, which is located at level 2 (*worker-human-being* model). Worker's type is Human placed one level above (*general-human-being* model) and, ultimately, Human's type is EClass. For each of the elements present in any of the models, one must always be able to follow the typing chains up to the topmost model located at level 0. The dashed semi-transparent lines in Figure 32 represent the typing chains of each element.

4.4. MCMTs amalgamation

We show in this section, to demonstrate the application of the constructions detailed in Section 3.2, three amalgamation cases, each of them combining one rule from each hierarchy.

The first amalgamated rule shown in Figure 33 is given by the combination of the *Create Task* (Figure 25) rule from the *process management* hierarchy and the *Undertake Activity* (Figure 29) rule from the *human-being* hierarchy. We identify in the META block both multilevel hierarchies (note they are separated by a vertical dotted line) involved in the two typing chains present in the FROM

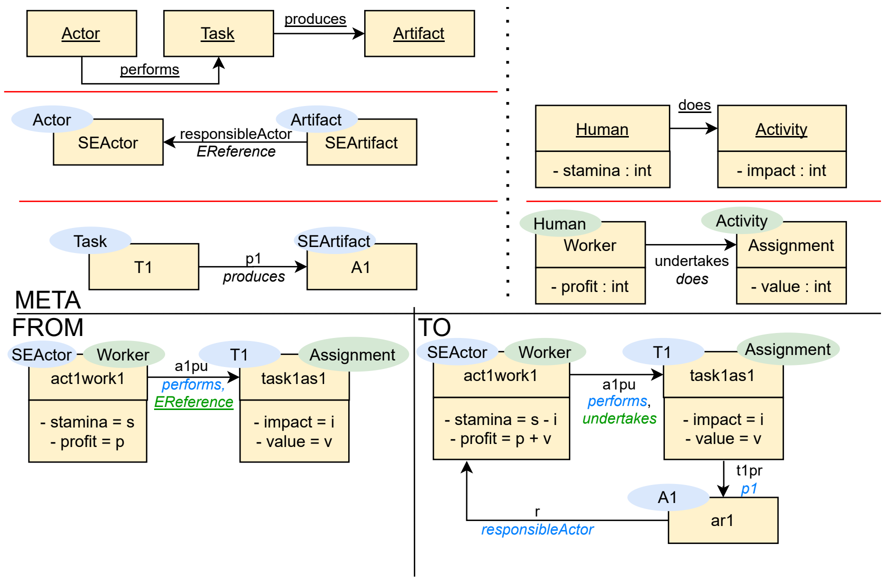


Figure 34: Second rule amalgamation: It combines *Produce Artefact* rule from the *process* hierarchy with *Undertake Activity* rule from the *human-being* hierarchy

and TO blocks, product of the amalgamation process. The complete amalgamated rule is automatically obtained by applying the construction shown in Figure 12, once I_0 has been provided by the user. This rule intuitively assigns to an actor/worker (act1work1) a task/assignment (task1as1) through a1pu, for the first hierarchy, and undertakes, for the second one. As clarified in *Case 3* of Section 3.2.3, act1work1's type from the *process* hierarchy is constrained to be EClass. The rule also connects r1 to task1as1 via r1e. Notice how r1e link is not involved with the *human-being* hierarchy, which makes sense since roles from the *process* hierarchy are not identified with anything into the *human-being* hierarchy. Finally, it also applies the attribute manipulation such as decreasing the stamina and increasing the profit of act1work1. This rule is identified by case number 3 in Table 1.

The second amalgamated rule displayed at Figure 34 is constructed by combining the *Produce Artefact* rule (Figure 26) from the *process management* hierarchy and again the *Undertake Activity* rule from the *human-being* hierarchy.

In this case, we illustrate this rule as it presents a peculiarity. As one can observe in Figure 34, there exists a mismatch between the number of levels in the two hierarchies. While the first hierarchy on the rule (located in the left-hand side of the dotted line in the META block) specifies three META levels, the second hierarchy or at the right-hand side only contains two levels. However, this is not a problem since either of the typing chains do not see themselves affected by the other, and it is perfectly fine to find such kind of situations. The application of this rule creates an artefact ar1 (which is not related to the

40

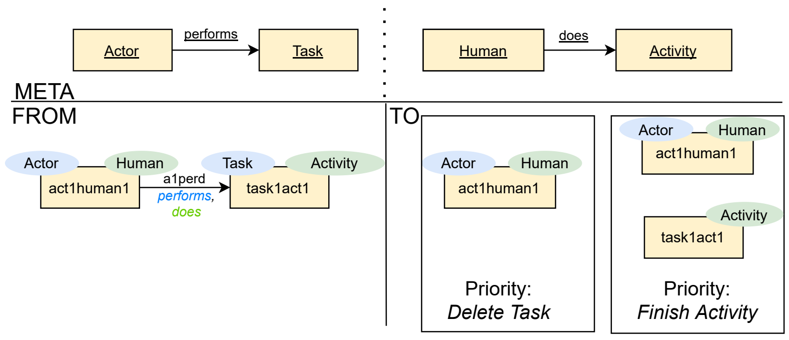


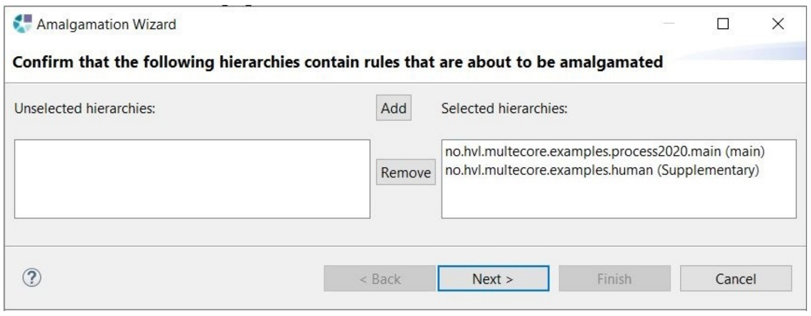
Figure 35: Third rule amalgamation: It combines *Delete Task* rule from the *process* hierarchy with *Finish Activity* rule from the *human-being* hierarchy. The result depends on which rule gets prioritised

human-being hierarchy) related to act1work1 through r and to task1as1 via t1pr. Again, act1work1 also gets updated stamina and profit. This rule construction is covered in case number 3 in Table 1 (notice the EReference second type of a1p in the FROM block).

The last amalgamation example we have obtained, is given by the combination of *Delete Task* rule (Figure 27) from the *process management* hierarchy and *Finish Activity* rule (Figure 30) from the *human-being* hierarchy. We illustrate in this case an example where prioritisation must be given to one of the rules in order the get R_M (TO block). The two results depicted in the TO part are calculated by applying the formulae given in Section 3.2.3. This example corresponds to case number 6 in Table 1, as one rule is keeping the node task1act1 while the other is removing it.

4.5. Amalgamation in MultEcore

In MultEcore, we have developed a guided-procedure that the modeller follows in order to get the set of amalgamated rules. To facilitate the explanation,


Amalgamation Wizard

Confirm that the following hierarchies contain rules that are about to be amalgamated

Unselected hierarchies:

Selected hierarchies:

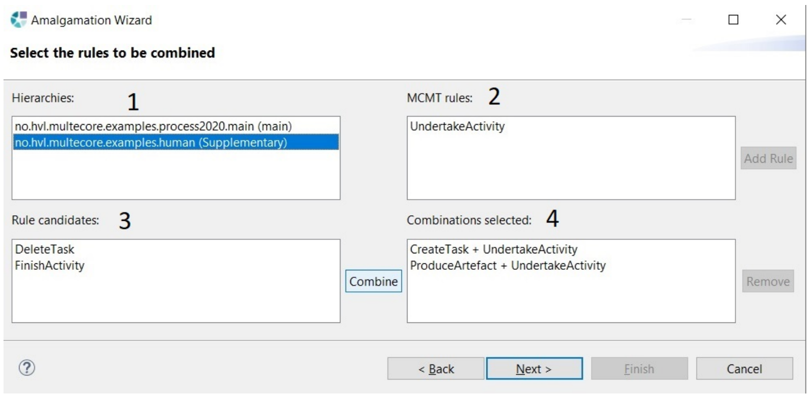
no.hvl.multecore.examples.process2020.main (main)
no.hvl.multecore.examples.human (Supplementary)

Figure 36: First step of amalgamation wizard: Selection of multilevel hierarchies to be combined

41

we describe each step and add a corresponding figure of how it is displayed to the user in the MultEcore wizard. We use the three amalgamated rules shown in Figures 33, 34 and 35 for illustrative purposes and to demonstrate that the produced amalgamated MCMT rules are sound with the expected results. The amalgamation process is semi-automatic, and defined by the following steps:

1. The modeller decides which multilevel hierarchies are going to be combined. These, together with their corresponding set of MCMT rules will be loaded into the wizard. This is shown in Figure 36 where both multilevel hierarchy projects have been selected. It is important to mention that one must select at least two of the available hierarchies in order to advance to the second step.
2. In this step, the user has to pick the MCMT rules that are going to be amalgamated. For instance, if we are combining two hierarchies, the modeller has to specify pairs of MCMTs that are to be amalgamated. We show in Figure 37 and describe below the four sub-steps that are involved at this stage of the wizard:
   - **Figure 37.1**. In this part of the dialog the user sees all the multilevel hierarchies that have been selected. In this example we have: . . . process2020main and . . . human.
   - **Figure 37.2**. In this box the user automatically sees the available MCMT rules that belong to the selected hierarchy in Figure 37.1. Selecting one of them and pressing Add Rule (right side of the Figure) adds the selected MCMT rule to the third box (Figure 37.3). Note that only one rule can be added per hierarchy and, for instance, adding two rules from the same hierarchy is not a valid situation. Once a rule has been added to Figure 37.3, it is removed from the box in Figure 37.2 until it has been resolved, i.e., combined with another rule.



no.hvl.multecore.examples.process2020.main (main)
no.hvl.multecore.examples.human (Supplementary)

UndertakeActivity

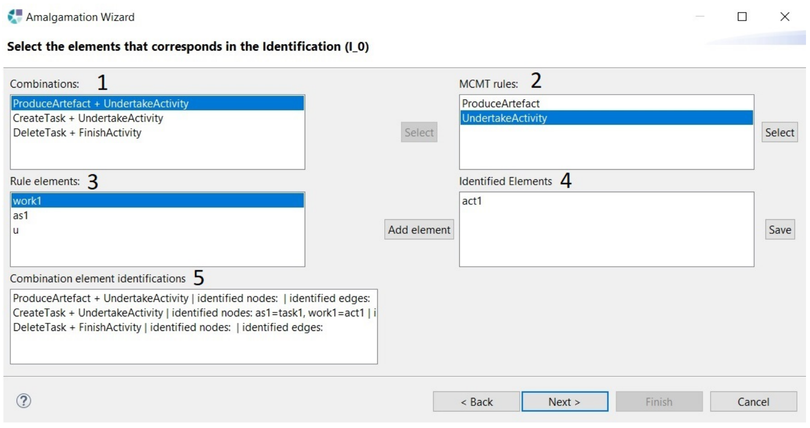
DeleteTask
FinishActivity

CreateTask + UndertakeActivity
ProduceArtefact + UndertakeActivity

Figure 37: Second step of amalgamation wizard: Selection of MCMT rules combinations.

- **Figure 37.3**. This box shows the potential MCMT rules that are a priori candidates to be combined. Note that FinishActivity and DeleteTask are currently shown in there, and pressing Combine will save this combination together with the already decided ones.
- **Figure 37.4**. This last box shows the combinations that have been stored for next steps of the wizard. Currently, two combinations have already been decided: CreateTask + UndertakeActivity and ProduceArtefact + UndertakeActivity. Combinations can be discarded by selecting one and clicking on Remove (bottom right of Figure 37). For the next step of the wizard we assume that the candidate combination in Figure 37.3 is finally combined.

3. For each tuple of assigned MCMTs, one needs to give the identification of the elements. As mentioned earlier, an essential step to achieve amalgamation (or, in general, composition) is the identification process where the elements that correspond to each other have to be identified (I_0). Several approaches in the literature use a so-called kernel rule to express correspondences between two or more rules [40, 41, 49, 42]. Thus, a mandatory step within this process for the user is to provide the correspondences between elements in the rules which are to be amalgamated. We show in Figure 38 the ongoing process of the identification of each node/edge for each rule combination and describe each part:
   - **Figure 38.1**. In this box the user can see the combinations selected in the previous step. Note that we were actually showing two combinations in Figure 37.4 and the last one (DeleteTask and FinishActivity in Figure 37.3) we assume it has been combined at this point. Clicking one of these combinations and pressing Select, reveals each MCMT rules involved in such a combination in the Figure 38.2 box.


Amalgamation Wizard

Select the elements that corresponds in the Identification (I_0)

Combinations: 1
ProduceArtefact + UndertakeActivity
CreateTask + UndertakeActivity
DeleteTask + FinishActivity

MCMT rules: 2
ProduceArtefact
UndertakeActivity

Rule elements: 3
work1
as1
u

Identified Elements 4
act1

Combination element identifications 5
ProduceArtefact + UndertakeActivity | identified nodes: | identified edges:
CreateTask + UndertakeActivity | identified nodes: as1=task1, work1=act1 | i
DeleteTask + FinishActivity | identified nodes: | identified edges:

Figure 38: Third step of the amalgamation wizard: Identification of elements in each rule combination.

43

The three available combinations are: ProduceArtefact + UndertakeActivity (selected), CreateTask + UndertakeActivity and DeleteTask + FinishActivity.

- **Figure 38.2**. In here the MCMT rules of the combinations are shown. The user can click on one of these rules and press Select which breaks down all of the available elements of the selected rule in the **Figure 38.3** box. In the example UndertakeActivity is selected.
- **Figure 38.3**. The individual elements that belong to the selected rule are shown in this part. To choose one to identify it with another corresponding element, click on it and press Add element. In the example, work1 (selected), as1 and u are the candidates elements, and, for example, pressing Add element will move work1 to the Figure 38.4 list. Note that, similarly as in the second step with the rules to be combined, only one element per rule can be selected for one combination, and adding it as a candidate for the identification temporarily removes it from the list in Figure 38.3. In this case, the combination taking place considers one element from UndertakeActivity (work1) and one from ProduceArtefact (act1 which already added in **Figure 38.4**).
- **Figure 38.4**. The identified candidate elements are shown in this box. Currently, only act1 is on the list. Adding work1 from the previous sub step will complete this identification list and will allow the Save button to be pushed, which adds the identification to the pool.
- **Figure 38.5**. This last box simply informs of the current status of the saved identified elements in each amalgamation.

4. Finally, once all the correspondences are established, the modeller gets a summary and is notified if *conflicts* have been detected. As discussed in Section 3.2.2, a conflict may appear, for instance, when an identified node is removed in one of the selected MCMTs, but kept in the other. Our way to resolve conflicts is by granting prioritisation to one of the rules. We show in Figure 39 the last step of the amalgamation wizard where the summary of the MCMT rules that are going to be amalgamated is provided. This step is divided into three categories:
    - **Figure 39.1**. Here the conflicting amalgamated MCMT rules are listed. In this case, there is only one conflicting situation, DeleteTask + FinishActivity. Picking it and pressing Select leads to the second sub-step.
    - **Figure 39.2**. The user can select in this box the rule that should get prioritised. In this example we have chosen DeleteTask.
    - **Figure 39.3**. This last part summarises the amalgamation cases that are going to be produced. Note that we are showing here the DeleteTask... situation to display how would it look like once the

MCMT rule that is going to get prioritised is selected, i.e., by pushing Select in Figure 39.2.

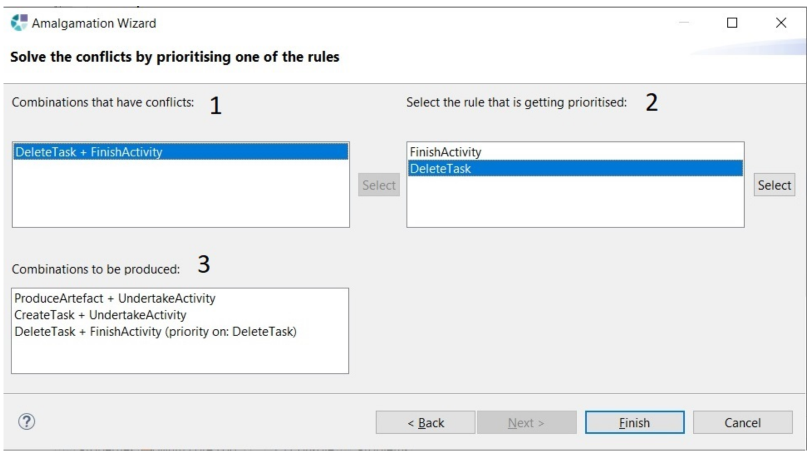
Once the Finish button is selected the engine computes the amalgamated MCMT rules based on the identifications provided and the prioritisations given.

4.6. Textual DSML for MCMTs

MCMT rules in MultEcore are specified using a textual editor where the MCMTs DSML [24, 25] has been built using Xtext [50]. This DSML provides the specification of modules containing a collection of MCMT rules defined independently of the hierarchy. The combined rules produced by the amalgamation engine have the same format than the MCMT rules that the user could manually write. Thus, the amalgamation results can be directly translated into an MCMT file. An example of the results that are obtained is shown in Figure 40.

For the sake of simplicity, we only show the textual representation of the third amalgamated rule DeleteTaskFinishActivity (with priority on *Delete Task*) which was graphically displayed in Figure 35. The other two amalgamated rules are shown in Appendix B (Figures B.42, B.43). In Figure 40, we distinguish three main blocks, the meta, the from and the to (lines 2, 22 and 29, respectively). In the from and to blocks we can define patterns according to the elements previously declared in the meta part. The meta block must contain a valid, non-empty pattern, but the from and to blocks may be empty. Within the textsfmeta we can define constant and variable elements, but we can only define variables in the from and to parts. They contain the same information that the corresponding blocks shown in the graphical rule.

Constant nodes are defined, for instance, as in line number 4 Actor: $process[1]!Actor where Actor is the name of the constant node, $ is used to denote


Amalgamation Wizard
Solve the conflicts by prioritising one of the rules
Combinations that have conflicts:
1
Select the rule that is getting prioritised:
2
Select
Select
Combinations to be produced:
3
< Back
Next >
Finish
Cancel

DeleteTask + FinishActivity

FinishActivity
DeleteTask

ProduceArtefact + UndertakeActivity
CreateTask + UndertakeActivity
DeleteTask + FinishActivity (priority on: DeleteTask)

Figure 39: Fourth step of the amalgamation wizard: Conflicts resolution and summary of the MCMT rules that are going to be amalgamated.

45

```
 1  rule DeleteTaskFinishActivity{
 2          meta{
 3                  //Nodes level 1 - Process
 4                  Actor: $process[1]!Actor
 5                  Task: $process[1]!Task
 6
 7                  //Nodes level 1 - Human
 8                  Human: $human[1]!Human
 9                  Activity: $human[1]!Activity
10
11                  //Edges level 1 - Process
12                  performs: $process[1]!Actor.performs
13
14                  //Edges level 1 - Human
15                  does: $human[1]!Human.does
16
17                  //Source.edge = Target
18                  [Actor.performs = Task]
19
20                  [Human.does = Activity]
21          }
22          from {
23                  act1human1: Actor, Human
24                  task1act1: Task, Activity
25                  a1perd: performs, does
26
27                  [act1human1.a1perd = task1act1]
28          }
29          to {
30                  act1human1: Actor, Human
31          }
32  }
```

Figure 40: Computed DeleteTaskFinishActivity MCMT rule. It corresponds to the graphical MCMT rule depicted in Figure 35 with priority on *Delete Task*

that is a constant, process is an alias of the rule it belongs to (either process or human) and [1] represents that it is located at level 1 of the meta block. Constant edges, such as the one defined in line number 12, are given by its name (performs) and ends with the form source.edge (Actor.performs). In this rule there are not variables defined in the meta block, but they are very similar with the exception that the $ is not written, and the nodes end with its type name. Also, attributes can be declared below each node specifying its type. We refer the reader to Figures B.42 and B.43 for some examples of variables and attributes. At the end of the meta block, we define the assignment expressions that are used to specify the structural relationships between the declared nodes by means of the declared edges. An example is given in line 18 [Actor.performs = Task], where Actor and Task are the source and target of the edge, performs. In the example, the from block of the rule defines a pattern consisting of three variables and one assignment expression, while its to block comprises just one

46

variable declaration. The from and to blocks follow the same structure. Nodes and edges in these levels are defined as shown in lines 24 and 25, respectively, where, for example, task1act1 has two types, Task from the main process hierarchy and Activity from the supplementary human one. Similarly as for the meta block, edges have to be specified within assignment expressions that link them with its respective sources and targets (line 27).

5. Related work

We first discuss approaches within the context of traditional MDSE and the Language Product Lines Engineering field that propose techniques to achieve composition.

Melange [51] is a tool for the construction of DSLs that supports modular language design and language modules composition. The dynamic semantics is defined operationally as aspects in the Kermeta meta-language [52]. Operational semantics of a DSL involves the use of an action language to define methods that are statically introduced in the concepts of the DSL abstract syntax. In our approach, we define the semantics separately, by means of MCMTs, avoiding the need to change the abstract syntax (for us, the multilevel hierarchy) of the DSML. Authors present in [53] an approach for building product lines of metamodels. The key point of these approaches is that a *transformation product line* is defined that becomes applicable for all metamodels in the set providing reusability and flexibility. Even though such approaches typically require specifying a binding between the transformation interface and the metamodel, the range of applicability is much wider than approaches where the transformation can be reused on a closed fixed metamodel set [54]. The approach in [54] is based on *featured model transformations (FMTs)* that can be seen as a kind of metamodel that integrates the variability of a whole family of metamodels which still provide a high degree of reusability. In our approach we go one step further as we do not only consider the reusability of the transformation rules within the same family, but also the incorporation of orthogonal languages.

GeKo [55] is a generic, extensible model weaver that can compose any models that conform to a common metamodel. To operate, it takes as parameters a base model, a pointcut model (the parametric pattern) and an advice model. The tool replaces all instances of the pointcut model that are found in the model with the advice model. While this approach focuses on the composition at the model (instance) level, we discuss in this work the composition of language descriptions via multilevel modelling hierarchies. Furthermore, GeKo operates only on the structure, while our approach also provides support for the amalgamation of dynamic semantics specified by means of MCMTs. MATA [56] is very similar to GeKo but it is founded on graph transformations to do composition of structure of models conforming to a common metamodel.

The work presented in [57] served us as inspiration to develop our approach. In their work, the authors formally define how composition of structure and amalgamation of semantic specifications can be achieved between a functional DSL and several parametric non-functional ones. While they establish a weaving process to construct the combined, final products (both structure- and semantic-wise), we try to be as minimally invasive as possible by incorporating the (supplementary) typing chains which can be later removed in a flexible way. Thus, as mentioned along this article, our structure combination process tends to be *virtual* rather than *physical* in the sense that we do not produce a new combined language, but incorporate/remove the new features we are interested in.

In the context of amalgamation of graph transformations, the authors implement rule amalgamation based on nested graph predicates in GROOVE [58]. In there, a single structure holds the different rules, where pattern rules can indicate the variations of the overall pattern structure. AToM^3 supports the amalgamation of rules to describe the explicit definition of interaction schemes in different rule editors [43]. The authors of the GReAT tool [59], define the concept of *Group*, so they can operate and apply delete, move or copy operations to each of the elements within the group, in the context of a transformation rule. In our approach we explore an alternative method to achieve amalgamation based on multiple typing.

6. Conclusions and future work

In this paper we have described an alternative method to achieve composition of structure and semantics of model descriptions. While some standard approaches might achieve composition, e.g., by implementing a merge operator, we take advantage of the notion of *application* and *supplementary* hierarchies to provide elements with more than one aspect by multiple typing them. Our formalisation based on category theory and graph transformations allows us to achieve such aspect-orientation flavor by incorporating additional typing chains. We have formally demonstrated how amalgamated MCMT rules can be generated by computing their components (namely, L_M, I_M and R_M) via pushouts $L_A +_{L_0} L_B$, $I_A +_{I_0} I_B$ and $R_A +_{R_0} R_B$. We differentiate between rules that are conflict free and those whose amalgamation would lead to conflicts. For the latter, we define an alternative formulation to compute R_M, based on which rule gets prioritised. Finally, we have illustrated and applied the constructions to a case study where two independent multilevel hierarchies are combined and their rules are amalgamated. Note that we rely on the user to provide the modulo components that make it possible to calculate the resulting constructions. We are investigating how to make this process (semi-)automatic by analysing how elements at the instance level are related and multiple typed to suggest and automatically compute amalgamated rules.

The MultEcore framework is currently supporting the amalgamation process described in Sections 4.5 and 4.6. We plan to incorporate the execution of composed hierarchies with their amalgamated MCMT rules into our MultEcore-Maude infrastructure that allows to handle simulation/execution [29]. Also, we plan to extend our case studies with other examples that allow us to evaluate all cases depicted in Table 1.

References

[1] C. Atkinson, T. Kühne, Processes and products in a multi-level metamodeling architecture, International Journal of Software Engineering and Knowledge Engineering 11 (06) (2001) 761–783.

[2] S. Zschaler, P. Sánchez, J. P. Santos, M. Alférez, A. Rashid, L. Fuentes, A. Moreira, J. Araújo, U. Kulesza, VML* - A Family of Languages for Variability Management in Software Product Lines, in: Software Language Engineering, Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers, 2009, pp. 82–102. doi:10.1007/978-3-642-12107-4_7.

[3] J. de Lara, E. Guerra, Deep meta-modelling with MetaDepth, in: Objects, Models, Components, Patterns, Vol. 6141, 2010, pp. 1–20. doi:10.1007/978-3-642-13953-6_1.

[4] C. Atkinson, R. Gerbig, Flexible deep modeling with Melanee, in: S. Betz, U. Reimer (Eds.), Modellierung 2016, Vol. 255 of LNI, Gesellschaft für Informatik, Bonn, 2016, pp. 117–122.

[5] E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. Van Mierlo, H. Ergin, AToMPM: A web-based modeling environment, in: MODELS-JP 2013, Vol. 1115 of CEUR Workshop Proceedings, 2013, pp. 21–25.

[6] S. Van Mierlo, B. Barroca, H. Vangheluwe, E. Syriani, T. Kühne, Multi-level modelling in the Modelverse, in: MULTI@ MoDELS, Vol. 1286 of CEUR Workshop Proceedings, 2014, pp. 83–92.

[7] UML, http://www.uml.org/.

[8] D. Steinberg, F. Budinsky, E. Merks, M. Paternostro, EMF: eclipse modeling framework, Pearson Education, 2008.

[9] P. Mohagheghi, W. Gilani, A. Stefanescu, M. A. Fernández, B. Nordmoen, M. Fritzsche, Where does model-driven engineering help? Experiences from three industrial cases, Software & Systems Modeling 12 (3) (2013) 619–639.

[10] J. Whittle, J. Hutchinson, M. Rouncefield, The state of practice in model-driven engineering, IEEE software 31 (3) (2014) 79–85.

[11] J. D. Lara, E. Guerra, J. S. Cuadrado, When and how to use multilevel modelling, ACM Transactions on Software Engineering and Methodology (TOSEM) 24 (2) (2014) 12.

[12] C. Atkinson, T. Kühne, Reducing accidental complexity in domain models, Software & Systems Modeling 7 (3) (2008) 345–359.

[13] C. Atkinson, T. Kühne, In defence of deep modelling, Inf. Softw. Technol. 64 (2015) 36–51. doi:10.1016/j.infsof.2015.03.010.

[14] C. Atkinson, R. Gerbig, T. Kühne, Comparing multi-level modeling approaches, in: Proceedings of the Workshop on Multi-Level Modelling co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2014), Valencia, Spain, September 28, 2014, 2014, pp. 53–61.

[15] C. Atkinson, T. Kühne, On evaluating multi-level modeling, in: Proceedings of MULTI @ MODELS, 2017, pp. 274–277.

[16] F. Macías, U. Wolter, A. Rutle, F. Durán, R. Rodriguez-Echeverria, Multilevel Coupled Model Transformations for Precise and Reusable Definition of Model Behaviour, Journal of Logical and Algebraic Methods in Programming 106 (2019) 167–195. `doi:10.1016/j.jlamp.2018.12.005`.

[17] J. de Lara, E. Guerra, Generic Meta-modelling with Concepts, Templates and Mixin Layers, in: Model Driven Engineering Languages and Systems - 13th International Conference, MODELS, 2010, pp. 16–30. `doi:10.1007/978-3-642-16145-2\_2`.

[18] D. Méndez-Acuña, J. A. Galindo, T. Degueule, B. Combemale, B. Baudry, Leveraging Software Product Lines Engineering in the development of external DSLs: A systematic literature review, Computer Languages, Systems & Structures 46 (2016) 206–235. `doi:10.1016/j.cl.2016.09.004`.

[19] J. Kienzle, G. Mussbacher, B. Combemale, J. Deantoni, A unifying framework for homogeneous model composition, Software & Systems Modeling 18 (5) (2019) 3005–3023.

[20] Arne Lange and Colin Atkinson, Multi-level modeling with MELANEE, in: Proceedings of MULTI @ MODELS, 2018, pp. 653–662.

[21] J. de Lara, E. Guerra, Refactoring Multi-Level Models, ACM Trans. Softw. Eng. Methodol. 27 (4) (2018) 17:1–17:56. `doi:10.1145/3280985`.

[22] C. Atkinson, T. Kühne, J. de Lara, Editorial to the theme issue on multi-level modeling, Software and Systems Modeling 17 (1) (2018) 163–165. `doi:10.1007/s10270-016-0565-6`.

[23] S. P. Jacome-Guerrero, J. de Lara, TOTEM: Reconciling multi-level modelling with standard two-level modelling, Computer Standards and interfaces In press.

[24] F. Macías, A. Rutle, V. Stolz, R. Rodriguez-Echeverria, U. Wolter, An Approach to Flexible Multilevel Modelling, Enterprise Modelling and Information Systems Architectures 13 (2018) 10:1–10:35. `doi:https://doi.org/10.18417/emisa.13.10`.

[25] F. Macías, Multilevel modelling and domain-specific languages, PhD thesis, Western Norway University of Applied Sciences and University of Oslo (2019).

[26] F. Macías, A. Rutle, V. Stolz, Multilevel Modelling with MultEcore: A Contribution to the MULTI 2017 Challenge, in: Proceedings of MULTI @ MODELS, 2017, pp. 269–273.

[27] A. Rodríguez, F. Macías, Multilevel Modelling with MultEcore: A Contribution to the MULTI Process Challenge, in: Proceedings of MULTI @ MODELS, 2019, pp. 152–163. doi:10.1109/MODELS-C.2019.00026.

[28] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, All about Maude a high-performance logical framework: how to specify, program and verify systems in rewriting logic, Springer-Verlag, 2007.

[29] A. Rodríguez, F. Durán, A. Rutle, L. M. Kristensen, Executing Multilevel Domain-Specific Models in Maude, Journal of Object Technology 18 (2) (2019) 4:1–21. doi:10.5381/jot.2019.18.2.a4.

[30] U. Wolter, F. Macías, A. Rutle, The Category of Typing Chains as a Foundation of Multilevel Typed Model Transformations, Tech. Rep. 2019-417, University of Bergen, Department of Informatics (November 2019).

[31] T. Kühne, A story of levels, in: Proceedings of MULTI @ MODELS, 2018, pp. 673–682.

[32] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer, Fundamentals of Algebraic Graph Transformation, Monographs in Theoretical Computer Science. An EATCS Series, Springer, 2006. doi:10.1007/3-540-31188-2.

[33] H. Ehrig, F. Hermann, U. Prange, Cospan DPO approach: An alternative for DPO graph transformations, Bulletin of the EATCS 98 (2009) 139–149.

[34] A. Rodríguez, A. Rutle, L. M. Kristensen, F. Durán, A Foundation for the Composition of Multilevel Domain-Specific Languages, in: MULTI@ MoDELS, 2019, pp. 88–97. doi:10.1109/MODELS-C.2019.00018.

[35] J. de Lara, E. Guerra, Domain-Specific Textual Meta-Modelling Languages for Model Driven Engineering, in: Modelling Foundations and Applications - 8th European Conference, ECMFA 2012, Kgs. Lyngby, Denmark, July 2-5, 2012. Proceedings, 2012, pp. 259–274. doi:10.1007/978-3-642-31491-9_20.

[36] A. Wortmann, O. Barais, B. Combemale, M. Wimmer, Modeling languages in Industry 4.0: an extended systematic mapping study, Software and Systems Modeling 19 (1) (2020) 67–94. doi:10.1007/s10270-019-00757-6.

[37] R. M. Burstall, J. A. Goguen, Putting theories together to make specifications, in: Proceedings of the 5th International Joint Conference on Artificial Intelligence. Cambridge, MA, USA, August 22-25, 1977, 1977, pp. 1045–1058.

[38] P. Stünkel, H. König, Y. Lamo, A. Rutle, Multimodel correspondence through inter-model constraints, in: S. Marr, J. B. Sartor (Eds.), Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming, Nice, France, April 09-12, 2018, ACM, 2018, pp. 9–17. doi:10.1145/3191697.3191715.

[39] P. Stünkel, H. König, Y. Lamo, A. Rutle, Towards multiple model synchronization with comprehensive systems, in: Fundamental Approaches to Software Engineering - 23rd International Conference, FASE 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Proceedings, Vol. Accepted for publication of Lecture Notes in Computer Science, Springer, 2020.

[40] P. Boehm, H. Fonio, A. Habel, Amalgamation of Graph Transformations: A Synchronization Mechanism, J. Comput. Syst. Sci. 34 (2/3) (1987) 377–408. doi:10.1016/0022-0000(87)90030-4.

[41] G. Taentzer, Parallel and distributed graph transformation - formal description and application to communication-based systems, Berichte aus der Informatik, Shaker, 1996.

[42] E. Biermann, H. Ehrig, C. Ermel, U. Golas, G. Taentzer, Parallel Independence of Amalgamated Graph Transformations Applied to Model Transformation, in: Graph Transformations and Model-Driven Engineering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday, 2010, pp. 121–140. doi:10.1007/978-3-642-17322-6_7.

[43] J. de Lara Jaramillo, C. Ermel, G. Taentzer, K. Ehrig, Parallel Graph Transformation for Model Simulation applied to Timed Transition Petri Nets, Electron. Notes Theor. Comput. Sci. 109 (2004) 17–29. doi:10.1016/j.entcs.2004.02.053.

[44] Y. Lamo, F. Mantz, A. Rutle, J. de Lara, A declarative and bidirectional model transformation approach based on graph co-spans, in: 15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, Madrid, Spain, September 16-18, 2013, 2013, pp. 1–12. doi:10.1145/2505879.2505900.

[45] A. Rossini, A. Rutle, Y. Lamo, U. Wolter, A formalisation of the copy-modify-merge approach to version control in MDE, J. Log. Algebr. Program. 79 (7) (2010) 636–658. doi:10.1016/j.jlap.2009.10.003.

[46] J. Almeida, A. Rutle, M. Wimmer, Preface to the 6th international workshop on multi-level modelling (MULTI 2019), in: 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019, IEEE, 2019, pp. 64–65. doi:10.1109/MODELS-C.2019.00015.

[47] J. P. A. Almeida, A. Rutle, M. Wimmer, T. Kühne, The MULTI Process Challenge, MULTI @MODELS Available at `https://bit.ly/2JeDEYi`.

[48] E. Gamma, Design patterns: elements of reusable object-oriented software, Pearson Education India, 1995.

[49] J. de Lara Jaramillo, C. Ermel, G. Taentzer, K. Ehrig, Parallel Graph Transformation for Model Simulation applied to Timed Transition Petri Nets, Electron. Notes Theor. Comput. Sci. 109 (2004) 17–29. `doi:10.1016/j.entcs.2004.02.053`.

[50] L. Bettini, Implementing domain-specific languages with Xtext and Xtend, Packt Publishing Ltd, 2016.

[51] T. Degueule, B. Combemale, A. Blouin, O. Barais, J.-M. Jézéquel, Melange: A meta-language for modular and reusable development of dsls, in: Proceedings of the 2015 SLE Conference, ACM, 2015, pp. 25–36.

[52] J. Jézéquel, B. Combemale, O. Barais, M. Monperrus, F. Fouquet, Mashup of metalanguages and its implementation in the Kermeta language workbench, Software and Systems Modeling 14 (2) (2015) 905–920. `doi:10.1007/s10270-013-0354-4`.

[53] J.-M. Bruel, B. Combemale, E. Guerra, J.-M. Jézéquel, J. Kienzle, J. de Lara, G. Mussbacher, E. Syriani, H. Vangheluwe, Comparing and classifying model transformation reuse approaches across metamodels, Software and Systems ModelingDoi: `10.1007/s10270-019-00762-9`.

[54] G. Perrouin, M. Amrani, M. Acher, B. Combemale, A. Legay, P. Schobbens, Featured model types: towards systematic reuse in modelling language engineering, in: Proceedings of the 8th International Workshop on Modeling in Software Engineering, MiSE@ICSE 2016, Austin, Texas, USA, May 16-17, 2016, 2016, pp. 1–7. `doi:10.1145/2896982.2896987`.

[55] M. E. Kramer, J. Klein, J. R. H. Steel, B. Morin, J. Kienzle, O. Barais, J. Jézéquel, Achieving Practical Genericity in Model Weaving through Extensibility, in: Theory and Practice of Model Transformations - 6th International Conference, ICMT 2013, Budapest, Hungary, June 18-19, 2013. Proceedings, 2013, pp. 108–124. `doi:10.1007/978-3-642-38883-5\_12`.

[56] J. Whittle, P. K. Jayaraman, A. M. Elkhodary, A. Moreira, J. Araújo, MATA: A unified approach for composing UML aspect models based on graph transformation, LNCS Trans. Aspect Oriented Softw. Dev. 6 (2009) 191–237. `doi:10.1007/978-3-642-03764-1\_6`.

[57] F. Durán, A. Moreno-Delgado, F. Orejas, S. Zschaler, Amalgamation of domain specific languages with behaviour, Journal of Logical and Algebraic Methods in Programming 86 (2017) 208–235. `doi:https://doi.org/10.1016/j.jlamp.2015.09.005`.

[58] A. Rensink, J. Kuperus, Repotting the Geraniums: On Nested Graph Transformation Rules, ECEASST 18. doi:10.14279/tuj.eceasst.18.260.

[59] D. Balasubramanian, A. Narayanan, S. Neema, F. Shi, R. Thibodeaux, G. Karsai, A Subgraph Operator for Graph Transformation Languages, ECEASST 6. doi:10.14279/tuj.eceasst.6.72.

Appendix A. Complete process management multilevel hierarchy

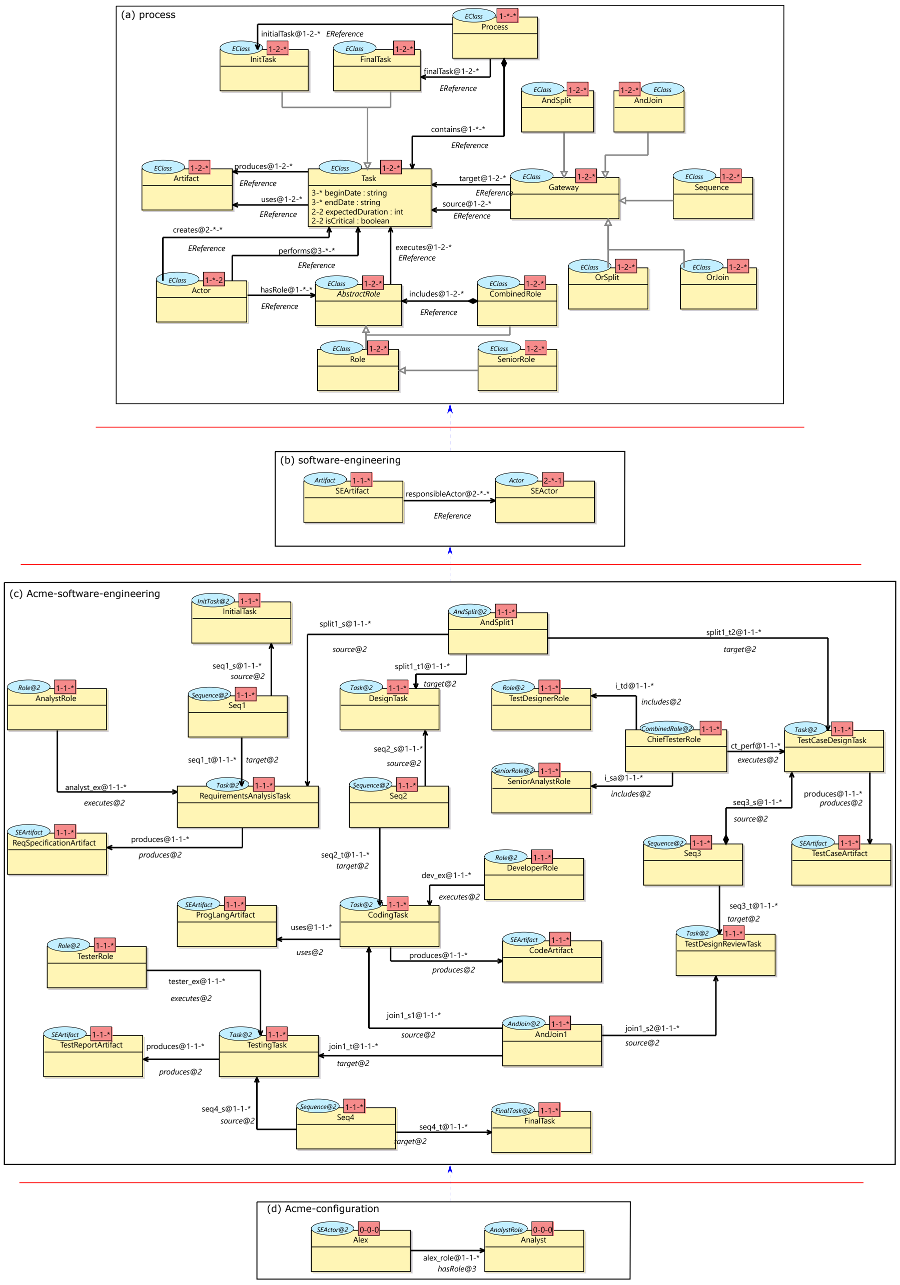


Figure A.41: Process management multilevel hierarchy

```
rule CreateTaskUndertakeActivity{
        meta{
                //Nodes level 1 - Process
                Role: $process[1]!Role
                Actor: $process[1]!Actor
                Task: $process[1]!Task
                //Nodes level 1 - Human
                Human: $human[1]!Human
                        Human.stamina : Integer
                Activity: $human[1]!Activity
                        Activity.impact : Integer
                //Edges level 1 - Process
                hasRole: $process[1]!Actor.hasRole
                performs: $process[1]!Actor.performs
                executes: $process[1]!Role.executes
                //Edges level 1 - Human
                does: $human[1]!Human.does
                //Nodes level 2 - Process
                R1: process[2]!Role
                T1: process[2]!Task
                //Nodes level 2 - Human
                Worker: human[2]!Human
                        Worker.profit : Integer
                Assignment: human[2]!Activity
                        Assignment.value : Integer
                //Edges level 2 - Process
                e: process[2]!Role.executes
                //Edges level 2 - Human
                undertakes: human[2]!Human.does
                //Source.edge = Target

                [Actor.hasRole = Role]
                [Actor.performs = Task]
                [Role.executes = Task]
                [Human.does = Activity]
                [R1.e = T1]
                [Worker.undertakes = Assignment]
        }
        from {
                act1work1: Actor, Worker
                        act1work1.stamina = #s#
                        act1work1.profit = #p#
                r1: R1
                task1as1: EClass, Assignment
                        task1as1.impact = #i#
                        task1as1.value = #v#
                a1role: hasRole

                [act1work1.a1role = r1]
        }
        to {
                act1work1: Actor, Worker
                        act1work1.stamina = #s - i#
                        act1work1.profit = #p + v#
                r1: R1
                task1as1: T1, Assignment
                        task1as1.impact = #i#
                        task1as1.value = #v#
                a1role: hasRole
                a1pu: performs, undertakes
                r1e: e

                [act1work1.a1role = r1]
                [act1work1.a1pu = task1as1]
                [r1.r1e = task1as1]
        }
}
```

Figure B.42: Full CreateTaskUndertakeActivity MCMT rule computed in MultEcore. It corresponds to the MCMT rule depicted in Figure 33

```
rule ProduceArtefactkUndertakeActivity{
        meta{
                //Nodes level 1 - Process
                Actor: $process[1]!Actor
                Task: $process[1]!Task
                Artifact: $process[1]!Artifact
                //Nodes level 1 - Human
                Human: $human[1]!Human
                        Human.stamina : Integer
                Activity: $human[1]!Activity
                        Activity.impact : Integer
                //Edges level 1 - Process
                performs: $process[1]!Actor.performs
                produces: $process[1]!Task.produces
                //Edges level 1 - Human
                does: $human[1]!Human.does
                //Nodes level 2 - Process
                SEActor: process[2]!Actor
                SEArtifact: process[2]!Artifact
                //Nodes level 2 - Human
                Worker: human[2]!Human
                        Worker.profit : Integer
                Assignment: human[2]!Activity
                        Assignment.value : Integer
                //Edges level 2 - Process
                responsibleActor: process[2]!EReference
                //Edges level 2 - Human
                undertakes: human[2]!Human.does
                //Nodes level 3 - Process
                T1 : process[3]!Task
                A1 : process[3]! SEArtifact
                                //Edges level 3 - Process
                p1 : process[3]!Task.produces

                //Source.edge = Target
                [Actor.performs = Task]
                [Task.produces = Artifact]
                [Human.does = Activity]
                [SEArtifact.responsibleActor = SEActor]
                [Worker.undertakes = Assignment]

                [T1.p1 = A1]
        }
        from {
                act1work1: SEActor, Worker
                        act1work1.stamina = #s#
                        act1work1.profit = #p#
                task1as1: T1, Assignment
                        task1as1.impact = #i#
                        task1as1.value = #v#
                a1pu: performs, EReference

                [act1work1.a1pu = task1as1]
        }
        to {
                act1work1: SEActor, Worker
                        act1work1.stamina = #s - i#
                        act1work1.profit = #p + v#
                task1as1: T1, Assignment
                        task1as1.impact = #i#
                        task1as1.value = #v#
                ar1 : A1
                a1pu: performs, undertakes
                t1pr : p1
                r : responsibleActor

                [act1work1.a1pu = task1as1]
                [task1as1.t1pr = ar1]
                [ar1.r = act1work1]
        }
}
```

Figure B.43: Full ProduceArtefactUndertakeActivity MCMT rule computed in MultEcore. It corresponds to the MCMT rule depicted in Figure 34

57

PAPER E

EXECUTION AND ANALYSIS OF MULTECORE MULTILEVEL MODELLING LANGUAGES USING MAUDE

Alejandro Rodríguez, Francisco Durán, Lars Michael Kristensen

In International Journal on Software and Systems Modeling, **Springer Nature**, 2020. (Submitted)

Noname manuscript No.
(will be inserted by the editor)

Execution and Analysis of MultEcore Multilevel Modelling Languages using Maude

Alejandro Rodríguez[1] · Francisco Durán[2] · Lars Michael Kristensen[1]

Received: date / Accepted: date

Abstract Multilevel Modelling (MLM) approaches make it possible for designers and modellers to work with an unlimited number of abstraction levels when specifying domain-specific modelling languages (DSMLs). Even though there exists plenty of work in the literature to support MLM solutions from a structural point of view, there is no consensus on how to specify the behaviour of such models. In this paper, we present a functional infrastructure that allows modellers to define the structure and the operational semantics of multilevel modelling hierarchies that can be later simulated and analysed. Using the MultEcore tool, one can design and distribute the models that compose the language family in a multilevel hierarchy, and specify their behaviour by means of multilevel transformation, so-called Multilevel Coupled Model Transformations (MCMTs). This work extends these MCMTs to describe the behaviour of MLM systems with basic support for attribute manipulation, rule conditions, and possibly nested boxes to handle submodel collections. We give a rewrite logic semantics to MLM, on which we have based our automated transformation from MultEcore to the rewriting logic language Maude. Then, we rely on Maude to simulate/execute MultEcore models and to exploit different analysis techniques supported by Maude, like reachability analysis, bounded and unbounded model checking of invariants and LTL formulas on systems with both finite and infinite reachable state spaces using equational abstraction. We illustrate our developed techniques on a DSML family for Petri nets.

Keywords Multilevel Modelling · Domain-specific modelling languages · Model transformations · Verification · Rewriting logic · Maude

✉ Alejandro Rodríguez
arte@hvl.no

Francisco Durán
duran@lcc.uma.es

Lars Michael Kristensen
Lars.Michael.Kristensen@hvl.no

[1] Western Norway Univ. of Applied Sciences, Bergen, Norway
[2] ITIS Software, University of Málaga, Málaga, Spain

1 Introduction

Multilevel Modelling (MLM) is a notable research area where models and their specifications can be organised into several levels of abstraction [4]. Indeed, the MLM community has shown that MLM is a favourable approach in domains such as process modelling and software architecture [6,8]. Although there exist diverse approaches for MLM (see [32,2,61,64] for some of them), they all share a common idea: lift the restriction on not limiting the number of levels that designers can use to specify modelling languages.

This restriction is present in traditional Model-Driven Software Engineering (MDSE) approaches which are based on the Object Management Group (OMG) [44] 4-layer architecture such as the Unified Modelling Language (UML) [63] and the Eclipse Modelling Framework (EMF) [59,45]. Like in traditional MDSE approaches, MLM uses abstractions and modelling techniques to tackle the continually increasing complexity of software by considering models as primary artefacts in each phase of the software engineering life-cycle [11]. Using MLM, modellers are no longer forced to fit their modelling language specifications within two levels of abstraction: one for (meta)models and one for their instances. This might be too restrictive for certain situations where the language is large and/or complex, and even more when defining behavioural domain-specific modelling languages (DSMLs). DSMLs that are, for instance, variations on general purpose languages, i.e., to specify different refinements aimed at specific domains, would require further concretisations of the metamodels. Moreover, these limitations may lead to complications like model convolution, accidental complexity, and mixing concepts belonging to different domains (see, e.g., [34,6,7] for discussions on these issues).

One of the most prominent applications of MDSE is the construction of DSMLs [45]. These are modelling languages that are tailored to a concrete application area [28] which bridges the gap between software engineers and domain experts. DSMLs are usually built on top of a more abstract modelling language, which requires well-defined infrastructures to handle the separation of different abstraction levels. Furthermore, MLM techniques are excellent for the creation of DSMLs, especially when focusing on behavioural languages, since behaviour is usually defined at the metamodel level while it is executed (at least) two levels below at the instance level [33,3,39]. The reason is that behaviour is reflected in the running instances of the models which in turn conform to their metamodel.

The approach for MLM proposed by the tool MultEcore [38,56], formally specified in [37], rests on the premise that one must be able to specify models (distributed along tree-like hierarchies) which are both generic and precise [39]. Even though various approaches have been proposed for the definition and simulation of behavioural models based on reusable model transformations (e.g., [49,35,51]), these rely on traditional two-level modelling hierarchies. Furthermore, modelling the behaviour through multilevel model transformations [3] and performing execution or analysis in MLM has not been widely explored yet.

Having a hierarchical organisation of the models that are in fact separate artefacts which altogether precisely capture the desired system facilitates future extensions and modifications. This applies not only in the existing levels, but also for adding or removing models to the existing multilevel hierarchy. Therefore, it can help to prevent pollution of models where specialisation of concepts would have to be done in the same model (even if they naturally fit in different levels of abstraction). Furthermore, this enhances modularisation and facilitates extendibility [57].

To cope with execution/simulation of models within the MLM context, Multilevel Coupled Model Transformations (MCMTs) were formally introduced in [39] as a multilevel transformation language that bridges the gap for the execution of multilevel modelling hierarchies. MCMTs are meant to achieve reusable multilevel model transformations for the specification of behaviour. In this work, we have improved the expressive capabilities of MCMTs by extending them with basic support for attributes, the specification of conditions to block their execution, and the possibility of expressing multiple patterns through the use of nested parametric boxes.

Even though the potential of the MCMTs has been illustrated in several examples, its practical applicability was limited. Indeed, the proposal in [39] was only theoretical and no proper implementation was available. We show here how we have turned the MultEcore editing facilities into a complete development environment in which we can, not only edit our MLM models, but also experiment with them through their simulation and execution, and analyse them by giving access to advanced verification and model checking tools.

We have provided such capabilities for simulation and analysis thanks to a formal specification of MultEcore models in rewriting logic [40,42], and specifically by providing a model-to-model transformation into the rewriting logic language Maude [15,17]. As we will see in the rest of the paper, the syntactical facilities of Maude have allowed us to use a representation of MLM hierarchies and MCMT rules very close to that of MultEcore. Indeed, this minimal representation distance has facilitated the automation of the bidirectional transformation between them. These transformations give MultEcore users access to the Maude execution engine, which is possibly the most efficient engine for rewriting modulo (combinations of) associativity, commutativity and identity [21,18]. In addition, it also gives access to Maude's formal tool environment, which includes, e.g., tools for reachability analysis, model checking, and confluence and termination analysis.

In summary, the contributions of this paper, which extend preliminary work presented in [55], are:

- The MCMTs version described in [39,55] had some practical problems and expressivity limitations. We extend and improve them introducing three main features: (i) attribute definition and manipulation, which brings additional expressivity to the specification of behaviour; (ii) rule conditions that add extra requirements for a rule to be applied; and (iii) nested boxes to handle submodel collections, improving expressiveness and reducing the proliferation of rules. A preliminary and very limited version of these boxes was presented in [55,57]. We present here a fully-operational full-fledged version of them, where boxes may appear in both sides of the rules, boxes may be nested, and each of them may have an explicit cardinality specified. Basic support for the Object Constraint Language (OCL) [13] has been added for the manipulation of attribute values and for the specification of conditions, which greatly improves the expressiveness of the tool.

- We present in this paper a rewriting logic semantics of MLM hierarchies and MCMT rules through their representation in Maude. This formal representation of MultEcore models allows us to execute and analyse such models using Maude's formal tools.
- A bidirectional transformation between MultEcore MLM models and Maude specifications has been developed. This functional infrastructure connects MultEcore to Maude, allowing us, not only to design our multilevel modelling hierarchy and specify its MCMTs, but also to simulate the specified systems and analyse and verify them using several techniques. Within our infrastructure, we encapsulate Maude as a background process that handles the instructions and return the execution and analysis results, given by the interface that the user uses to interact.
  While there was some basic infrastructure in [55], this is now a mature tool, not only more efficient and configurable, but covering all the features of the language. Although as we discuss in the conclusions section there is much work ahead of us, the MultEcore editor and the transformation between MultEcore and Maude is completely operational.
- The application of the complete infrastructure to a case study for a multilevel DSML for Petri nets, from the design phase to the final execution of the system that we later verify through reachability analysis and model checking techniques. While the case study is described in this paper, we refer the reader to [54] for the complete MultEcore and Maude specifications, including additional details on their analysis.

Outline: We describe in Section 2 the features that characterise our MLM approach using a multilevel DSML for a Petri nets multilevel hierarchy. We level-wise explore each model comprising the hierarchy, from both the structural and behavioural points of views. Section 3 provides an overview of the infrastructure that transforms the multilevel DSML defined in MultEcore into a Maude specification. This section provides details of the generated Maude specification. We demonstrate the use of such an infrastructure with a case study in Section 4, where we perform execution and analysis of a Petri net model of a gas station. In Section 5 we discuss related work. Finally, Section 6 concludes the paper and outlines directions for future work.

2 Multilevel Modelling of Petri nets

The MultEcore tool is designed as a set of Eclipse plugins, giving access to its mature ecosystem (integration with EMF) and incorporating the flexibility of MLM. In the MultEcore approach [39], the abstract syntax is provided by MLM models and the behaviour is provided by Multilevel Coupled Model Transformations (MCMTs) [39,37]. Using the MultEcore tool, modellers can (i) define MLM models using the model graphical editor; (ii) define MCMTs using its rule editor; and (iii) execute and analyse specific models. The execution of MultEcore models rely on a transformation of the models into Maude [15] specifications. When we design a multilevel DSML, we first define its syntax/structure with multilevel modelling hierarchies. Then the behaviour is specified via our multilevel transformation language.

For implementation reasons, MultEcore prescribes the use of Ecore [59] as root graph at level 0 in all example hierarchies. Models are distributed in *multilevel modelling hierarchies*. A multilevel modelling hierarchy in our context is a tree-shaped hierarchy of models with a single root typically depicted at the top of the hierarchy tree. Thus, hierarchies enclose a set of models connected via typing relations. Levels are indexed with increasing natural numbers starting from the uppermost one, having index 0.

To illustrate the different concepts and techniques discussed in this paper, we use as case study a DSML for Petri nets. In the next sections, we describe each of the models that constitute the Petri net multilevel hierarchy. We depict in Appendix A (Figure 19) the complete developed PNs multilevel hierarchy (where we omit Ecore at the top).

2.1 Petri nets metamodel

Petri nets (PNs) is a well-established formalism to model concurrent systems [46,47]. There is a rich body of theoretical results enabling analysis of PNs, and an enormous set of supporting tools.

A PN is a directed bipartite graph, in which the nodes represent transitions (i.e., events that may occur, represented by rectangles/bars) and places (i.e., states, represented by circles). For example, Figure 1 shows a Petri net model using a well-known concrete syntax. In it, we find places `p1`...`p4` and transitions `tr1` and `tr2`, where `p1` and `p2` are connected to `tr1` via input arcs, `p3`

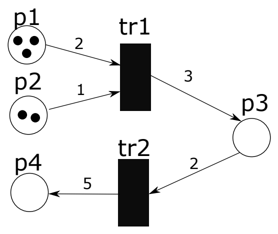


Fig. 1 Simple Petri net model

is connected to tr1 through an output arc and to tr2 via an input arc, and finally p4 is an output place of tr2.

The nodes and arcs constitute the static structure of a PN. The dynamic behaviour of the net is given by the *token game*, representing various states of the system. This token game is based on the firing of transitions that lead to the consumption/production of tokens; each fired transition produces a new model state.

A particular state is a snapshot of the system's behaviour. The state of a place is called its *marking*, represented by the presence or absence of tokens (commonly represented as black dots), in the places. In the example shown in Figure 1 there are three tokens in p1 and two tokens in p2. The current state of the modelled system (marking) is given by the number of tokens in each place.

The increasing complexity of systems has promoted a proliferation of Petri nets variants and extensions during the last decades, as often classical Petri nets are too basic to capture the needs of certain environments. A brief comparison of different kind of Petri nets can be found in [9]. Although our hierarchy could include other types of PNs, here we only include classical or regular PNs and reset/inhibitor nets [65].

We show in Figure 2 a PN metamodel aimed to capture the abstract concepts of Petri nets. This metamodel represents the level 1 of the hierarchy (Figure 19(a)). The purpose of this model is merely structural. In other words, subsequent levels below it should define the concrete semantics of the PN language(s) (as we show in this section). A PN contains nodes, which can be either a Place or a Transition, and Arcs. The tool MultEcore allows us to make use of the *inheritance* relation and to mark Node as an abstract class, which cannot be instantiated (note the italics). As shown in the figure, the type of a node, provided by some element in an upper level metamodel, is indicated in an (light blue) ellipse at its top left side, e.g., EClass is the type of PN, Node, Transition, Place, and Arc. The type of an arrow is written near the arrow in italic font type, e.g., EReference for arcs, nodes, source, target, inArcs and outArcs. We support attribute declarations that can be currently typed by one of the four basic Ecore types, namely Integer, Real, Boolean and String. These attributes can be instantiated in a lower level with a value, as illustrated in Section 2.4. For the manipulation of attribute values, and the specification of rule conditions, a subset of OCL [66,13,12] is currently supported.[1]

The annotations displayed as three numbers in a (red) box at the top right of each node, and concatenated to the name after "@" for every reference, specify their potencies. Potency in attributes is displayed as two numbers as an attribute does not have depth, since first it is declared, and eventually in a level below it is instantiated. The two numbers are specified in front of the attribute name. *Potency* [29] is a well-known concept in MLM and it is used on elements as a way of restricting the levels at which this element may be used to type other elements. By using potencies on elements, we can define the degree of flexibility/restrictiveness we want to allow on the elements of our multilevel hierarchy. The first two values, *start* and *end*, specify the range of levels below, relative to the current level, where the element can be directly instantiated. The third value, *depth*, is used to control the maximum number of times that the element can be transitively instantiated, or re-instantiated, regardless of the levels where this occurs. For instance, the potency specified for Arc, Node, Transition and Place is 1-2-3, which means that an element can be directly instantiated one and two levels below (levels 2 or 3 in the hierarchy), and such instances can be re-instantiated up to 3 additional times. This depth is therefore dependent on the value of the type, and the depth of an element must always be strictly less than the depth of its type.

2.2 Regular Petri nets

The regular Petri nets that we consider in this paper are not restricted to the so-called *Ordinary Petri Nets* where input and output arcs consume or produce, respectively, only a single token [25]. We allow natural numbers on arcs so that more than one token can be added/removed at a time. Following the PNs convention, we denote this number as the *weight* of the arc.

2.2.1 A metamodel for regular Petri nets

Figure 3 displays the *regular-petri-nets* model. It is located at level 2 of the hierarchy (Figure 19) where we instantiate the concepts defined at level 1. Thus, in this

[1] OCL was chosen since it has been part of UML for several years, is one of the most used languages in EMF-based applications, and it is consider a standard in the MDSE community. A full description of the supported subset of OCL, as well as the adaptation of OCL to the multilevel modelling context, will be published elsewhere.

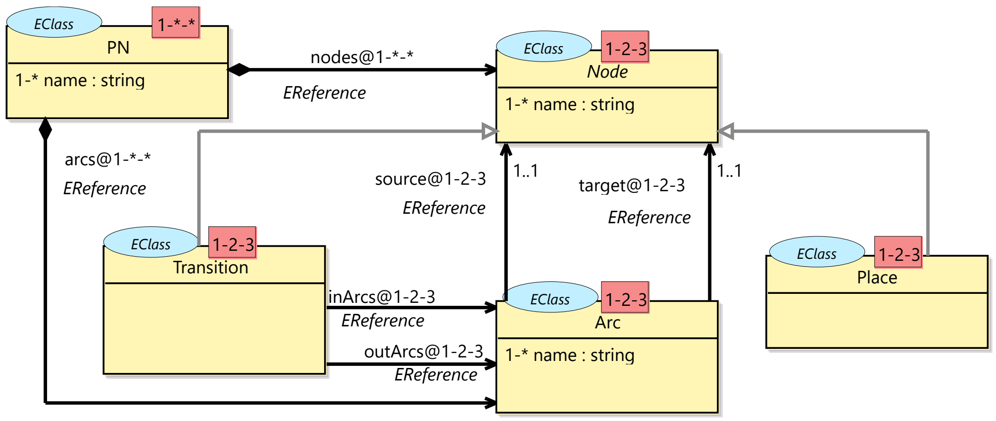


Fig. 2 Conceptual Petri nets metamodel (also shown in Figure 19(a))

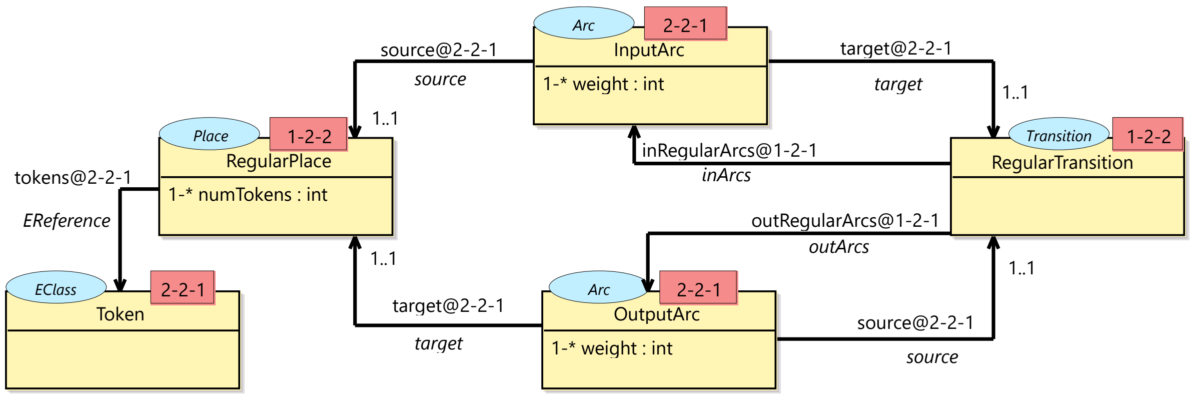


Fig. 3 Regular Petri nets metamodel (also shown in Figure 19(b))

model, we provide the structural basis for the modeller to be able to define further Petri nets instance models. InputArc and OutputArc connect regular places and regular transitions. A RegularPlace controls how many tokens it is holding via the numTokens attribute[2]. The weight of arcs is represented as attributes weight of type int in classes InputArc and OutputArc.

2.2.2 Operational semantics of regular Petri nets

The multilevel transformation language that MCMTs define allows us to exploit multilevel capabilities and is powerful enough to specify behavioural descriptions in an operational way. Transformation rules can be used to represent actions that may happen in the system. A rule has the form of $LHS \Rightarrow RHS$ *if* C, where LHS is a multilevel model pattern (which may contain variables), and RHS is model pattern in which we can use the variables already appearing in LHS.

[2] Please, note that the number of tokens may be calculated with the OCL expression rp.tokens->size(). The attribute is however used to speed up calculations and to illustrate the use of attributes.

C is a boolean condition, in which we can use variables from LHS. Given a model M that represents a state of the system, we say that there is a match of LHS on M if there is a submodel $M|_p$ of M such that for some assignment σ of the variables in LHS, we have $LHS\sigma = M|_p$, where $LHS\sigma$ denotes the application of the assignment σ to LHS. Given a match of LHS on M, for some assignment σ and the submodel $M|_p$, the condition $C\sigma$ is evaluated, where, similarly, $C\sigma$ denotes the application of the assignment σ to the condition C. If it evaluates to *true*, then the application of the rule consists in the replacement of the submodel $M|_p$ of M by $RHS\sigma$, which we denote $M[RHS\sigma]_p$. In other words, if there is a match of the rule on the model, and its condition is satisfied, then the matched submodel is replaced by the model specified in the right-hand side of the rule.

The way to express the behaviour of systems using transformation rules is by specifying rules modelling each of the possible actions that may occur. In PNs, actions occur when transitions are fired. In a regular PN, we only have regular arcs connecting places with transitions. Although transitions can have an arbitrary number of input and output places, such an action can

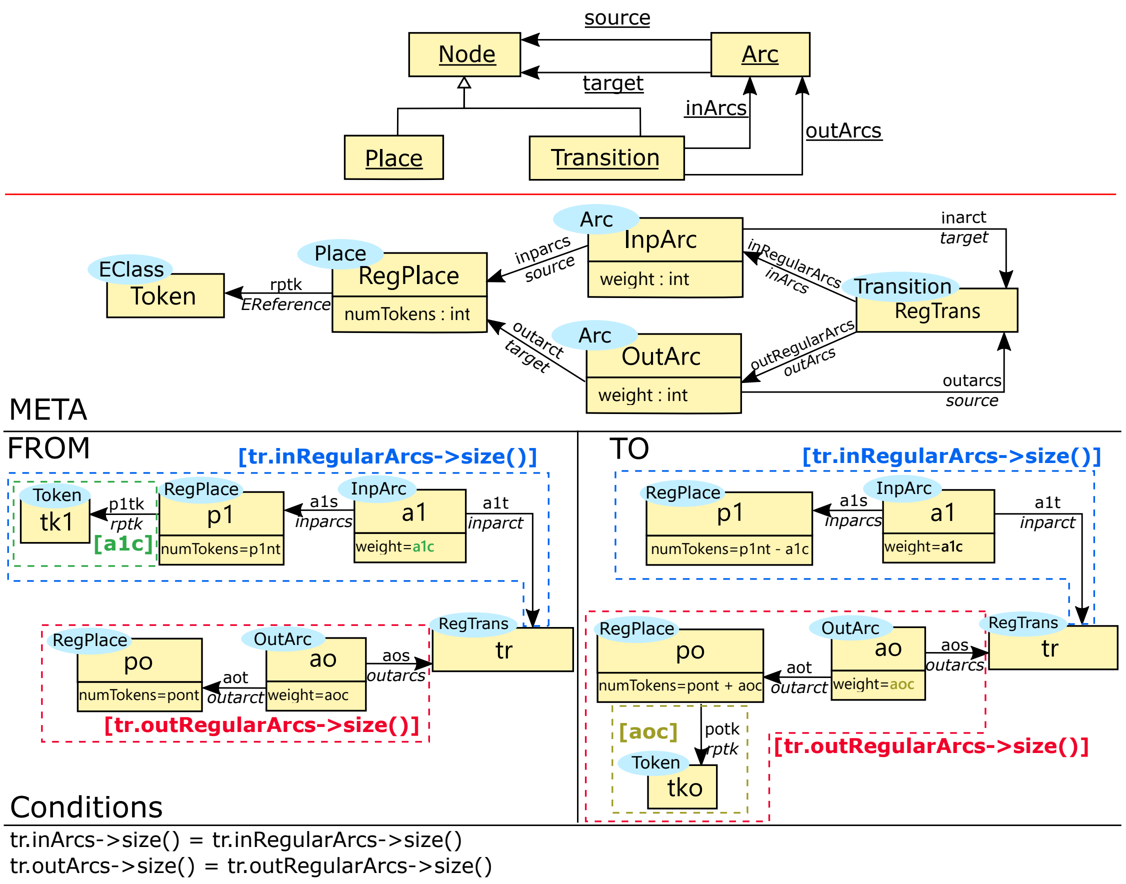


Fig. 4 Rule *Fire regular transition*: It removes tokens in the input places and creates new ones in the output places

be specified with the MCMT rule called *Fire regular transition* depicted in Figure 4. Specifically, this rule models a transition being fired, taking into account the information of the input places (arcs connected to it) and the output places (arcs where new information is to be produced). The FROM and TO blocks describe the left pattern and the right pattern of the rule, respectively. The META block depicts a multilevel pattern allowing us to locate types at any level that can be used as individual types for the items in the FROM and TO blocks, respectively. Notice that the META facilitates the definition of an entire multilevel pattern, therefore, we can specify several META levels within the block.

At the top level of Figure 4, we mirror parts of the *petri-nets-concepts* model (depicted in Figure 2), defining elements like Node, Arc, Transition, Place, source, target, inArcs and outArcs as constants — constant elements have their names underlined and their types are not specified, either via ellipses for nodes or italics text for references. We depict in the META block those elements and their relationships that are useful for the specification of the FROM and TO patterns.

In the second META level (below the red horizontal line), we capture elements to serve as types to scope the execution of regular PNs. In this level we find elements as variables such as Token which type is denoted in the ellipse right above it (EClass), RegPlace of type Place, and the reference rptk of type EReference. Similarly, we express attributes (such as numTokens) that later are going to be used in the levels below.

Please note that the horizontal lines do not enforce consecutiveness between the levels specified in the rule with respect to the hierarchy. This leads to a more natural way of defining that a type is defined at some level above, without explicitly stating at which level. In fact, this also promotes flexibility in case of future modifications of the number of branches (horizontal dimension) and the depth (vertical dimension) of hierarchies. For instance, the three levels depicted in the rule in Figure 4 would match to levels 1, 2 and 4 in the multilevel hierarchy depicted in Figure 19. As the aim of the running PNs multilevel hierarchy is not to highlight the horizontal/vertical flexibility, we refer the interested reader to [55, Section 4.2] for details on this.

We specify in the FROM block what elements must be found in the model in order to be able to fire a transition. As one can observe, dashed boxes are specified around certain parts of the FROM model. A key point

]

when defining model transformation rules is to make them as reusable as possible. Furthermore, in a PN, there might not only be as many input/output places connected to a transition as one requires, but also an arbitrary number of tokens residing within each of these places. Clearly, it is not practical to define one rule per possible combination of these connections, as the number of rules would rapidly blow up. MCMTs allow the use of nesting boxes to define patterns where its unfolding would result in a collection of elements. As seen in the rule, boxes may appear in both sides, and they can be nested.

The blue dashed box in Figure 4 encapsulates the nodes tk1, p1 and a1, as well as the references p1tk, a1s and a1t, covering all the potential input places connected to the transition (matched to tr) in the model. The number of instances of this pattern submodel is given by the OCL expression tr.inRegularArcs→size(), which represents the number of incoming arcs, i.e., the size of the collection of incoming regular arcs of the transition tr.

In OCL, the size() operator calculates the size of the collection it is applied on. The tr.inRegularArcs expression returns the collection of edges whose source is tr and its type is inRegularArcs. Note, however, that the way in which types are used in MLM is a bit different than for standard OCL. This allows transitive typing, which as we will see below, may be very useful. If instead, as in the condition of the rule, we use tr.inArcs, then we get the collection of edges of type inArcs or any of its instances. Note that the expression tr.inArcs→size() = tr.inRegularArcs→size() checks whether all the incoming arcs of a given transition tr are of type inRegularArcs. This means that the rule is only applicable on transitions whose arcs are all *regular*. The number of total input (resp. output) arcs, inArcs (resp. outArcs), must be equal to the number of input (resp. output) regular arcs, inRegularArcs (resp. outRegularArcs).

Analogously, and using the OCL expression tr.outRegularArcs→size(), a second (red) dashed box allows us to specify a number of output places (and corresponding arcs) connected to the transition.

Note the (green) nested box in the FROM part, inside the (blue) one we were just referring to for the incoming arcs. This inner box allows us to take an arbitrary number of tokens from the input place. For a specific instantiation of the rule, the cardinality of the box is matched to the variable a1c that takes the value of the weight attribute of arc a1. Indeed, given these boxes, a transition may have multiple incoming arcs, and for each incoming place-arc, multiple tokens.

There are also boxes on the TO part. Notice that the input and output arcs are left unmodified, but the appropriate number of tokens are added to the corresponding output places. The number of tokens to put in an output place is provided by the weight attribute of the outcoming arc. The nested (green) box in the TO part, inside the (red) box, indicates that the number of tokens (tko) to be added to each output place po connected to tr via ao, is given by the value aoc of the weight of the arc ao. Finally, note the use of OCL expressions for the manipulation of attributes. In this case, the numTokens attributes of places p1 and po are correspondingly updated: each input place p1 from which some tokens are removed and each output place po that receives tokens, gets its numTokens attribute, respectively, decreased (numTokens = p1nt - a1c) or increased (numTokens = pont + aoc) with the corresponding number of tokens.

In summary, the rule can be executed if the unfolded number of elements is found during the matching process, and all the conditions are satisfied. If this happens, the model in the TO part is produced. In this case, the execution of the rule removes all the tokens present in each of the input places as specified in the boxes, and creates new tokens on the output places.

2.3 Reset/inhibitor Petri nets

A reset/inhibitor PN [65] is a PN that in addition to regular arcs may also have reset and inhibitor arcs. A reset arc is an input arc that connects a place to a transition and that removes all the tokens of the place when the transition is fired. This is useful as a "cleaning mechanism" in models that capture, e.g., certain environments where messages might be retransmitted and buffers could accumulate old messages. An inhibitor arc is an input arc which is used to reverse the logic of an input place. With an inhibitor arc, the absence of a token in the input place is what enables the connected transition (not its presence). For instance, inhibitor arcs can be used to delay certain actions until a system is idle, or to wait until the end of a loop.

Figure 5 shows a very simple example of a reset/inhibitor PN in which we have one arc of each type. In this example, p1 is connected to tr1 via a regular input arc (defined in Figure 3), p2 via a reset arc (denoted with double arrow heads) and p3 via an inhibitor arc (distinguished with a small circle instead of an arrow head). Thus, this transition could be fired according to the semantics of each of the arcs: since (i) p1 has 3 tokens and its regular input arc requires 2; (ii) p2 does not block the firing of the transition, but it will be emptied by its connected reset arc; and (iii) p3 has 0 tokens which fulfils the enabling semantics of the inhibitor arc.

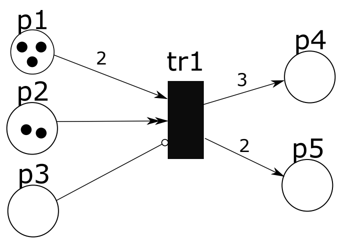


Fig. 5 Concrete syntax of a reset/inhibitor Petri net example

2.3.1 A metamodel for reset/inhibitor Petri nets

Figure 6 shows the model *reset-inhibitor-petri-nets*, placed at level 3 of the hierarchy (see Figure 19(c)). The model captures rules extended with the so-called *reset arcs* and *inhibitor arcs*. As in the model at level 2, the refined Transition in Figure 6 keeps track of the inhibitor and reset arcs connected to it (through references inInhibitorArcs and inResetArcs, respectively). While ExtendedPlace and ExtendedTransition are typed by elements in the level right above, ResetArc and InhibitorArc nodes are typed directly by Arc, which is located two levels above (as denoted by the @2 after the type).

One could argue that these elements that hold specialisation semantics can be realised using inheritance in the metamodel, which indeed is a valid alternative. However, this would lead to a single bigger metamodel where specialisations on elements (e.g., Arc) that belong to different domains are put together and further extensions in each domain would have to be handled in the same metamodel. Although a more detailed discussion may be found, e.g., in [34,6,7], note that, by having this "physical" separation, the modeller has more control on the individual artefacts and therefore the subsequent modifications would be done easier (enhancing reusability). Furthermore, there might be extra horizontal extensions when considering alternative domains, which can be more naturally achieved by promoting this level separation. Note that even we might separate elements within different levels, we do not necessarily make this separation because such elements are related through "type-instance" relationships.

In our approach, we follow the so-called *abstraction semantics* to organise elements within the multilevel hierarchy based on how abstract they are. Thus, for us, organising elements in different models is a feature that primarily enhances modularisation and promotes separation of concerns [6]. In other words, we do not encourage the *level segregation* principle [30], which establishes that level organisational semantics should be unique, i.e., aligned to one particular organisational scheme, such as *classification* or *generalisation*. Nonetheless, we do encourage the *level cohesion* principle [30], that is, we recommend to organise elements that are semantically close (by means of potency and level organisation).

2.3.2 Behaviour for reset/inhibitor Petri nets

Reset/inhibitor Petri nets have additional semantics that have to be properly managed. The MCMT rule *Fire reset/inhibitor transition* is depicted in Figure 7. Please, compare this rule with the *Fire regular transition* rule shown in Figure 4. The rule *Fire reset/inhibitor transition* handles the case in which a transition has any number of arcs of any of the three types (regular, reset or inhibitor), but in particular, if there are only regular arcs, it behaves as the *Fire regular transition* rule. Observe that the rule in Figure 7 includes a third META level, where we capture variable elements such as ExtPlace (of type RegPlace), InhArc (representing inhibitor arcs), ResArc (denoting reset arcs) and ExtTrans. As in the levels above, we determine inResetArcs and inInhibitorArcs references with ExtTrans as source, which can be later used in the OCL expressions for the boxes/conditions.

In the FROM block, we need to specify that we might find any number instances of each of the three kinds of arcs. We do it by encapsulating patterns for each of the arc types into a separate box. Corresponding boxes in the right-hand side specify the corresponding action to take on such an arc and its corresponding place. Notice that boxes for regular and reset arcs have corresponding nested boxes specifying the appropriate number of instances. These boxes are described as follows:

Regular arcs: The boxes handling regular arcs are exactly as those depicted in the *Fire regular transition* rule, where the box in the FROM block with cardinality tr.inRegularArcs→size() captures each regular arc a1 connecting a place p1 to the transition tr, and removes the number of tokens of each place as given by the weight a1c on the arc. The corresponding number of tokens is then put in the corresponding output places in the TO block.

Reset arcs: The box in the FROM block with cardinality tr.inResetArcs→size() captures the reset arcs a2 that connect input places p2 to the transition tr. To remove all the tokens tk2 present in the connected place p2, the number of tokens in the place is used as cardinality of the inner box. Note that these

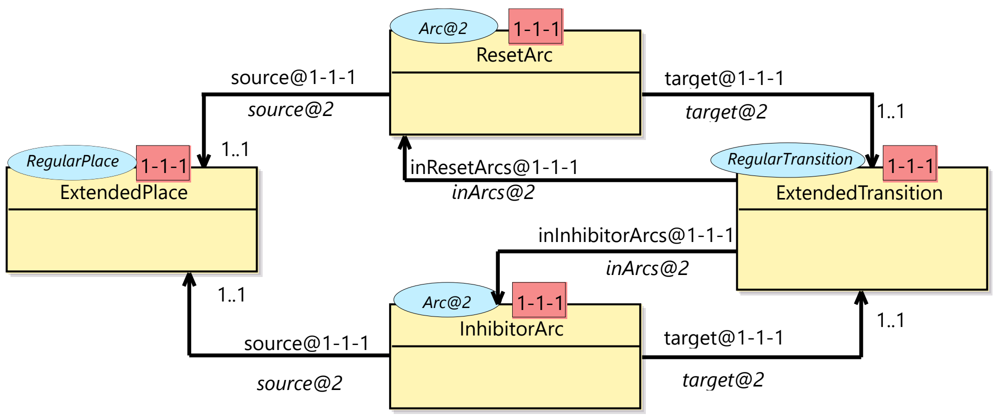


Fig. 6 Reset/inhibitor Petri nets metamodel (also shown in Figure 19(c))

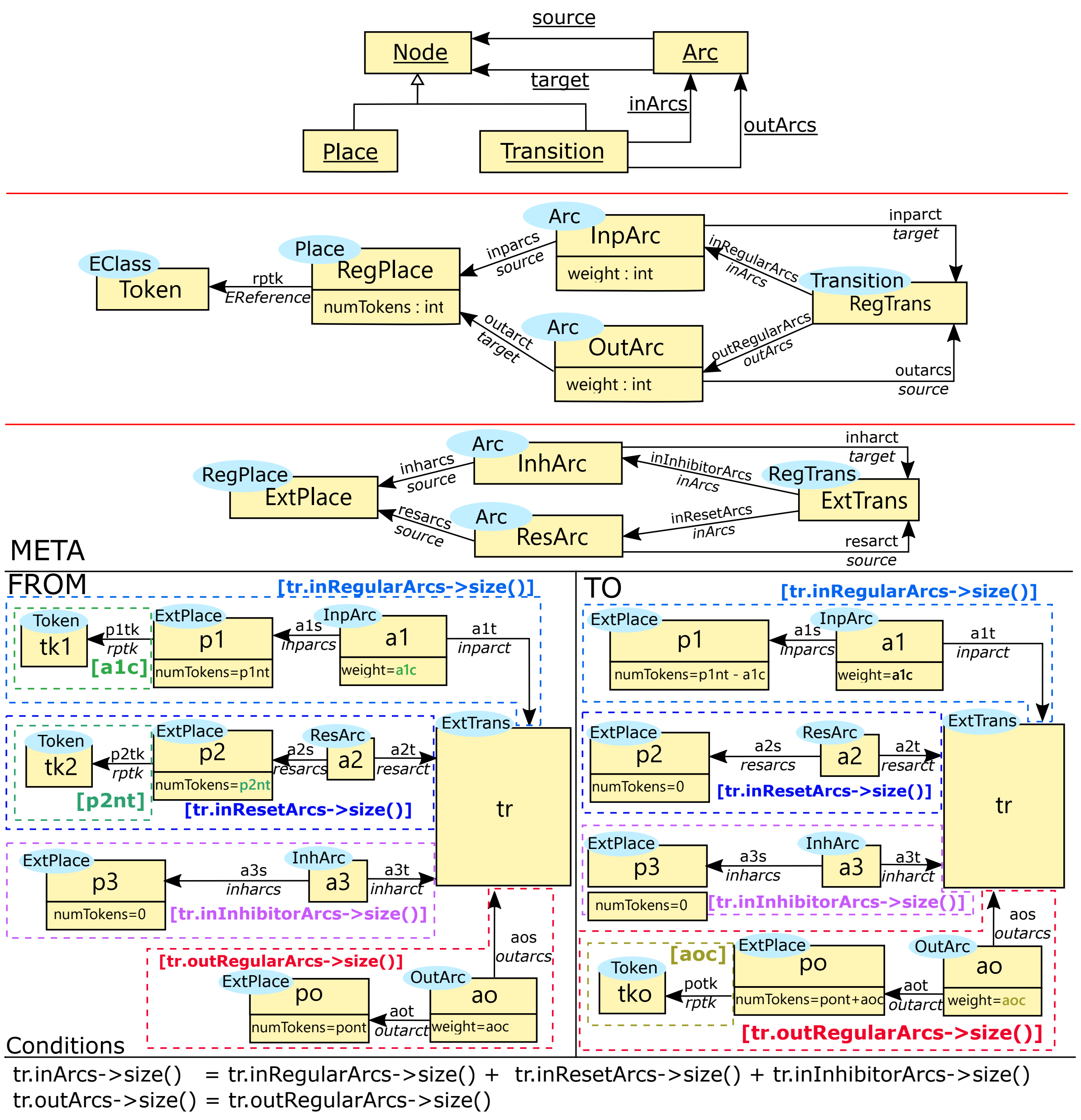


Fig. 7 Rule *Fire reset/inhibitor transition*: modelling the firing of transitions with regular, reset and inhibitor arcs

tokens do not appear in the corresponding box in the TO block. In this way, all of them are removed.

Inhibitor arcs: A third box with cardinality tr.inInhibitorArcs→size() captures inhibitor arcs. Since for the transition to be enabled the number of tokens of each place connected via an inhibitor arc must be 0, we simply specify this directly in p3, where it is stated that the attribute numTokens has value zero.

The rest of the rule looks very similar to what we have already seen. The condition tr.inArcs→size() = tr.inRegularArcs→size() + tr.inResetArcs→size() + tr.inInhibitorArcs→size() checks that the total number of input arcs is the sum of the number of regular input arcs, the reset arcs and the inhibitor arcs. The condition tr.outArcs→size() = tr.outRegularArcs→size() checks that the total number of output arcs is the number of regular output arcs. These conditions would be key for further extensions of the current PN hierarchy.

If the FROM block of the rule matches a submodel of the PN and the conditions are satisfied, the application of the rule results in the removal of the corresponding tokens from the places connected either via regular or reset arcs, and the creation of new tokens in the output places. Notice that the attributes on the places that keep track of the number of tokens get updated.

2.4 Petri nets examples

With the hierarchy described along Sections 2.1–2.3, we can now define models of regular PNs and models of reset/inhibitor PNs. This is possible, as potency specifications allow us to design the hierarchy in a way where *deep instantiation* [5] can be achieved, being able to instantiate elements residing in any level above.

To illustrate how PNs are represented using the given hierarchy, we show a first example using a concrete syntax for Petri nets and then its corresponding one using the MultEcore (abstract) syntax. Figure 8 shows a simple example using regular PNs where four places (two input and two output) and one transition are depicted. To the left we can see that p1 and p2 carry three and two tokens, respectively. Firing tr1 transition would remove 2 tokens from p1 and 1 from p2, and would create 3 and 2 tokens in p3 and p4, respectively, as expressed by the weight in the arcs.

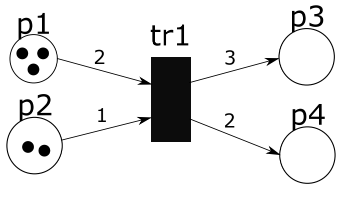


Fig. 8 Concrete syntax of a regular Petri net example

The MultEcore representation of the PN in Figure 8 is shown in Figure 9. Since we consider this model at the instance level, we use potency 0-0-0 in the elements. This is used to enforce that elements at the bottom level (in this case level 4) are used purely as instances, which cannot be refined further at levels below it.

As a second example, the MultEcore representation of the PN depicted in Figure 5 is depicted in Figure 19(d).

3 Execution of Multilevel DSMLs using Maude

Maude [14,15,17] is a specification language based on rewriting logic [41], a logic of change that can naturally deal with states and non-deterministic concurrent computations. A rewrite logic theory is a tuple $(\Sigma;E;R)$, where $(\Sigma;E)$ is an equational theory that specifies the system states as elements of the initial algebra $T_{(\Sigma;\mathrm{E})}$, and R is a set of rewrite rules that describe the one-step possible concurrent transitions in the system. Σ is a signature that specifies the type structure (e.g., sorts and subsorts) and operations, and E is the collection of equations and memberships declared in the functional module. Rewrite specifications thus described are executable, if they satisfy restrictions such as termination and confluence of the equational subspecification, and coherence of equations and rules.

Maude provides support for rewriting modulo associativity, commutativity and identity, which perfectly captures the evolution of models made up of objects linked by references as in graph grammars. In summary, Maude provides, among others, the following useful features:

Formal specification. The Maude specification of multilevel hierarchies and MCMTs represents a formal semantics of MultEcore models in rewriting logic. Based on such formalisation, the transformation MultEcore $\longleftrightarrow$ Maude has been automated.

Execution of the specification. The Maude specification obtained from MultEcore models using the above transformation is executable, and therefore it can be used to simulate our MultEcore models in Maude. The versatile rewriting engine is not only efficient, but also provides functionalities to customise the way we go through the execution steps. We can simulate our systems by letting Maude choose the path to follow, or we can specify a concrete path specifying it step by step, or by means of execution strategies.

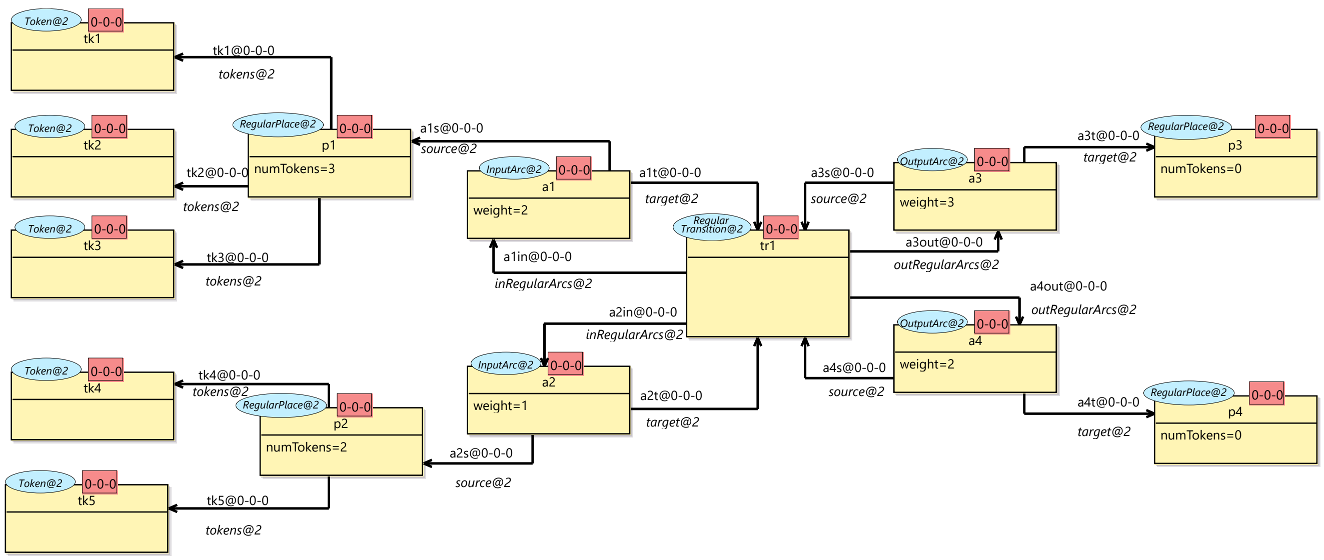


Fig. 9 MultEcore syntax of a regular Petri net example

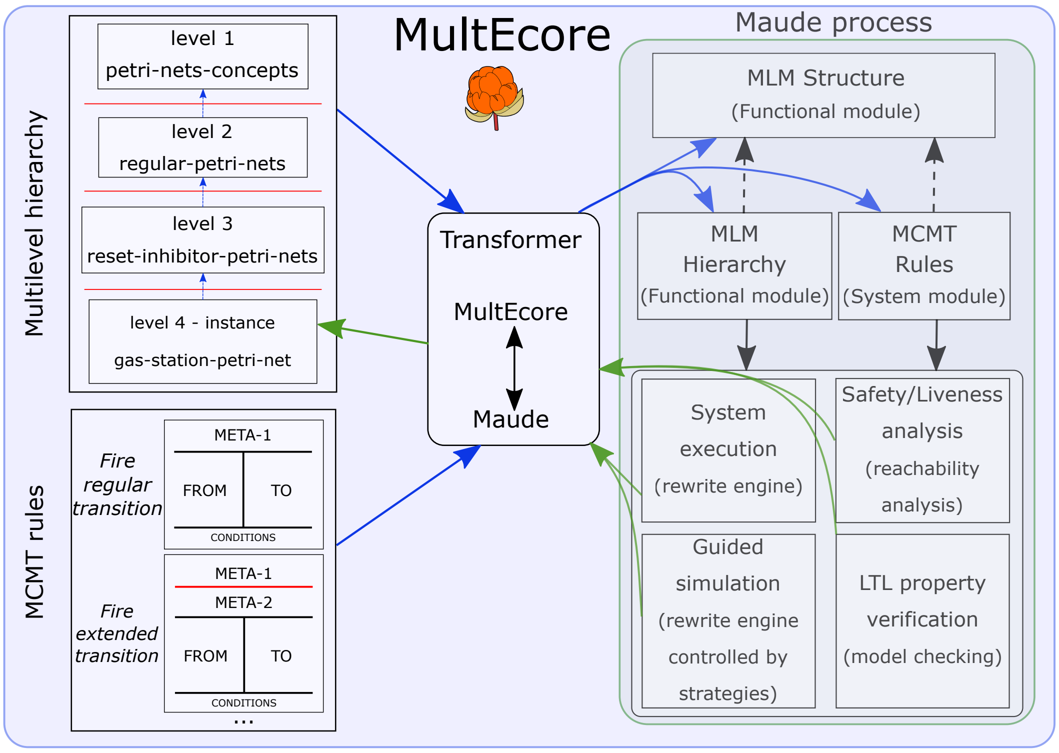


Fig. 10 Infrastructure for the execution and analysis of multilevel modelling hierarchies

Formal environment. Once the rewriting logic specification of a MultEcore model is available, we can use the tools in Maude's formal environment to analyse it. For example, we can check properties such as confluence or termination of our specifications, and can also perform reachability analysis, model checking or theorem proving.

The overall MultEcore-Maude infrastructure is sketched in Figure 10. The left-hand side shows the MultEcore part, where we specify multilevel DSMLs by providing a Multilevel Hierarchy and a set of MCMT rules. The Transformer MultEcore $\longleftrightarrow$ Maude has been developed as a bidirectional transformation that takes MultEcore textual specifications and automatically generates Maude specifications, and then takes the XML output files that Maude produces as result of performing execution and analysis, and automatically translates them into MultEcore models graphically displayed.

To grasp an intuition of how the transformation works, each MultEcore object (including both a hierarchy and its MCMTs) is mapped into a corresponding Maude object. References and conditions are handled in exactly the same way, by using references as names, and using the same set of expressions (types and operators) for conditions. The rewriting modulo associativity, commutativity and identity available in Maude captures quite naturally the intended operational semantics of MCMTs. The major challenges were the handling of boxing and the performing of the rewriting on multilevel hierarchies. The support for OCL is based on the Maude semantics of OCL proposed in [58].

The right-hand side of Figure 10 shows the Maude process perspective. The transformer produces a functional module with the equational theory used to represent MLM hierarchies, the MLM Hierarchy, and a system module with rewrite theory that represents the MCMT Rules. The representation of MLM hierarchies and MCMTs is presented in Sections 3.1 and 3.2-3.3, respectively. We illustrate in Section 4.1 some of the possibilities for execution and analysis of the models on a case study. As we will see in this section, MultEcore encapsulates the interaction with the Maude tools, which are hidden to the user. The Maude specification is however available to the user, who can interact directly with the Maude environment to get full access to all its features. The complete MultEcore description (both the hierarchy and the MCMTs), the corresponding full Maude specification and the experiments and properties verified can be found in [54].

3.1 Multilevel hierarchies in Maude

In Maude, object-oriented systems can be specified by object-oriented modules in which classes and subclasses are declared, with the usual support for inheritance, and dynamic binding. A class is declared with syntax class $C \mid a_1\colon S_1, \ldots, a_n\colon S_n$, where C is the name of the class, a_i are attribute identifiers, and S_i are the sorts of the corresponding attributes. The objects of a class C are record-like structures of the form $< O : C \mid a_1\colon v_1, \ldots, a_n\colon v_n >$, where O is the identifier of the object, and v_i are the current values of its attributes.

To represent multilevel metamodels we have introduced declarations to represent multilevel hierarchies as collections of objects each of which represents one of the level models. Specifically, in our approach, a multilevel hierarchy is represented as a structure of sort System of the form

$$\{ \ model_1 \ model_2 \ \ldots \ model_n \ \}$$

where each $model_i$ is an object of class Model that represents a model in the hierarchy.

Figure 11 shows an excerpt of the Maude specification obtained from the MultEcore Petri net multilevel hierarchy — notice the ellipses added for space reasons. Since levels are numbered starting from 0 (Ecore), the object representing level i's model uses level(i) as identifier. Such an object uses attributes to share the name of the model (name), the name of its immediate metamodel (om), its collection of nodes (elts), and a collection of the relations between these nodes (rels). Elements and relations are themselves represented as objects, of classes Node and Relation, respectively. Each node has attributes to store its name (name), type (type) and its own attributes (attributes). These attributes are again represented as objects with attributes to keep, depending on the level, its name or value (nameOrValue) and its type (type). A relation object has attributes to store its source (source), target (target), and multiplicities, provided by the two usual values (min-mult and max-mult). To avoid name clashes between levels, object identifiers are represented using the operator oid, and nodes and relations using the operator id. Both operators take the level number in which they are defined as first argument, and either a unique number or a string with its actual name.

For instance, the object in lines 1–6 represents level 0, the Ecore model, which has one node with name and type id(0, "EClass") (line 4) and one relation EReference (lines 5–6). Notice how the source and target of this relation refer to the names of the source and target nodes, respectively, which in this case is the same id(0, "EClass"). The *petri-nets-concepts* model in lines 7–14 represents the model in Figure 2 (also Figure 19(a)). Lines 10–12 show the representation of node Node, of type EClass, which has several attribute, among which we can see its attribute with name name of type String. The instance model at level 4 is shown in lines 16–28. Note that among its nodes, there is one with name id(4, "p1") (in lines 20-21), of type id(3, "ExtendedPlace") — a node in its metamodel — which has an attribute of type id(2, "numTokens") with value 0.

3.2 Box-free MCMTs in Maude

In Maude, object-oriented systems are axiomatised by equational theories describing their states as algebraic

```
{ < level(0) : Model |
    name : "Ecore",
    om   : "Ecore",
    elts : (< oid(0,  1) : Node     | name : id(0, "EClass"),     type : id(0, "EClass"), attributes : none >),
    rels : (< oid(0,  2) : Relation | name : id(0, "EReference"), type : id(0, "EReference")),
                                      source : id(0, "EClass"),   target : id(0, "EClass"), ... > >
  < level(1) : Model |
    name : "petri-nets-concepts",
    om   : "Ecore",
    elts : (< oid(1,  1) : Node     | name : id(1, "Node"), type : id(0, "EClass"),
                                      attributes : (< oid(1, 2) : Attri | nameOrValue : id(1, "name"),
                                                                          type : id(1, "String") > )> )>
                                                    ...),
    rels : (...) >
  ...
  < level(4) : Model |
    name : "reset-inhibitor-petri-net-example",
    om   : "reset-inhibitor-petri-nets",
    elts : (...
            < oid(4, 12) : Node     | name : id(4, "p1"),  type : id(3, "ExtendedPlace"),
                                      attributes : (< oid(4, 13) : Attri | nameOrValue : 0, type : id(2, "numTokens") >) >
            ...
            < oid(4, 28) : Node     | name : id(4, "tk1"), type : id(2, "Token"), attributes : none >
            ...),
    rels : (...
            < oid(4, 36) : Relation | name : id(4, "tk1"),  type : id(2, "tokens"),
                                      source : id(4, "p1"), target : id(4, "tk1"), ...>
            ...) > }
```

Fig. 11 Excerpt of the Petri net multilevel hierarchy in Maude representation

data types and collections of conditional rewrite rules specifying their *behaviour*. Rewrite rules are written as

$$\mathsf{crl}\ [l] : T \Rightarrow T' \ \mathsf{if}\ C$$

where l is the rule's label, T and T' are terms, and C is its guard or condition. As MultEcore's MCMTs, Maude rules describe the local, concurrent transitions that are possible in the system, i.e., when a part of the system state fits the pattern T, then it can be replaced by the corresponding instantiation of T'. Also as for MCMTs, the guard C acts as a blocking precondition: a conditional rule can only be fired if its condition is satisfied. Rules may be given without label or condition.

Given the representation of multilevel hierarchies presented in the previous section, the transformation of MCMT rules without boxes is straightforward. Basically, since the META section does not change, its corresponding representation appears in both left- and right-hand sides. The left-hand side of the rule is completed with the representation of the FROM block, and the representation of the TO block is added to its right-hand side. Variables in MCMTs are represented as Maude variables, and conditions are placed in the conditions as such. Since we restrict conditions to basic types, basic operators and certain selected OCL operations, the expressions can be handled directly by Maude. The last issue to consider is new names and identifiers in right-hand sides. As we explained above, identifiers are represented using the id and oid operators, which take the level in which the object is created and a unique number as arguments. Such numbers are generated using a Counter object, whose value attribute gets increased every time a new identifier is created.

Let us illustrate this general procedure on a specific example. Consider the rule *Fire regular transition* depicted in Figure 4, but let us assume first that it has no boxes in it, that is, let us assume that it takes one single token from the unique input place of a transition and moves it to its unique output place. If this were the case, the corresponding generated Maude rule would be the one shown in Figure 12. Again, notice the ellipses.

The left- and right-hand sides of the rule are given in lines 16–34 and 39–58, respectively. The corresponding condition is in lines 61–62. The META section is represented in lines 2–15 and 37–38. The three objects representing the META section are replicated in both sides. They provide the appropriate context for the rule, but it is in the FROM and TO sections where the actual change is modelled. Notice the use of variables to identify the levels (L1, L2 and L3). These are used to match any specific levels in the hierarchy on which the rule is applied. They do not need to be consecutive levels, the only restriction is given in the condition, where it is checked that L1 < L2 < L3 (line 61). Notice that the Maude rule represents quite closely the corresponding MultEcore MCMT. For instance, constants in the MCMT rules are mapped into Maude constants and ground terms. Variables are used both to represent free elements in the rules and also any other elements not explicitly specified. The condition of the MCMT rule is written as such in line 62. Finally, notice the use of the Counter object, which in the left-hand side has some value N (line 35) and in the right-hand side has

```
crl [Fire-1-to-1-Regular-Arcs] :
   { < level(L1) : Model | name : M,
         elts : (< 001 : Node | name : id(L1, "Arc"), type : id(0, "EClass"), Atts01 >
                 < 002 : Node | name : id(L1, "Node"), type : id(0, "EClass"), Atts02 >
                 < 003 : Node | name : id(L1, "Transition"), type : id(0, "EClass"), Atts03 >
                 < 004 : Node | name : id(L1, "Place"), type : id(0, "EClass"), Atts04 >
                 Elts),
         rels : (
                 < 005 : Relation | name : id(L1, "inArcs"), type : id(0, "EReference"), source : id(L1, "Transition"), Atts05 >
                 < 006 : Relation | name : id(L1, "outArcs"), type : id(0, "EReference"), source : id(L1, "Transition"), Atts06 >
                 < 007 : Relation | name : id(L1, "source"), type : id(0, "EReference"), source : id(L1, "Arc"), Atts07 >
                 < 008 : Relation | name : id(L1, "target"), type : id(0, "EReference"), source : id(L1, "Arc"), Atts08 >
                 Rels),
         Atts >
     < level(L2) : Model | ... >
     < level(L3) : Model | name : M''',
         elts : (< 026 : Node | name : tr_1, type : id(L2, "Transition"), Atts26 >
                 < 027 : Node | name : ao_1, type : id(L2, "OutputArc"),
                   attributes : (< 028 : Attri | nameOrValue : 1, type : id(L2, "weight"), Atts28 > Attri27), Atts27 >
                 < 029 : Node | name : po_1, type : id(L2, "Place"),
                   attributes : (< 030 : Attri | nameOrValue : pont, type : id(L2, "numTokens"), Atts30 > Attri29), Atts29 >
                 < 031 : Node | name : tk1_1, type : id(L2, "Token"), Atts31 >
                 < 032 : Node | name : a1_1, type : id(L2, "InputArc"),
                   attributes : (< 033 : Attri | nameOrValue : 1, type : id(L2, "weight"), Atts33 > Attri32), Atts32 >
                 < 034 : Node | name : p1_1, type : id(L2, "Place"),
                   attributes : (< 035 : Attri | nameOrValue : p1nt, type : id(L2, "numTokens"), Atts35 > Attri34), Atts34 >
                 Elts'''),
         rels : (< 036 : Relation | name : a1t_1, type : id(L2, "target"), source : a1_1, target : tr_1, Atts36 >
                 < 037 : Relation | name : p1tk_1, type : id(L2, "tokens"), source : p1_1, target : tk1_1, Atts37 >
                 < 038 : Relation | name : a1s_1, type : id(L2, "source"), source : a1_1, target : p1_1, Atts38 >
                 < 039 : Relation | name : aos_1, type : id(L2, "source"), source : ao_1, target : tr_1, Atts39 >
                 < 040 : Relation | name : aot_1, type : id(L2, "target"), source : ao_1, target : po_1, Atts40 >
                 Rels'''),
         Atts''' >
     < counter : Counter | value : N >
     Conf }
=> { < level(L1) : Model | ... > ---- as in the left-hand side
     < level(L2) : Model | ... > ---- as in the left-hand side
     < level(L3) : Model | name : M''',
        elts : (< 026 : Node | name : tr_1, type : id(L2, "Transition"), Atts26 >
                < 027 : Node | name : ao_1, type : id(L2, "OutputArc"),
                  attributes : (< 028 : Attri | nameOrValue : 1, type : id(L2, "weight"), Atts28 > Attri27), Atts27 >
                < 029 : Node | name : po_1, type : id(L2, "Place"),
                  attributes : (< 030 : Attri | nameOrValue : pont + 1, type : id(L2, "numTokens"), Atts30 > Attri29), Atts29 >
                < 031 : Node | name : a1_1, type : IA_1,
                  attributes : (< 032 : Attri | nameOrValue : 1, type : id(L2, "weight"), Atts32 > Attri31), Atts31 >
                < 033 : Node | name : p1_1, type : id(L2, "Place"),
                  attributes : (< 034 : Attri | nameOrValue : p1nt - 1, type : id(L2, "numTokens"), Atts34 > Attri33), Atts33 >
                < oid(L3, N) : Node | name : id(L3, N + 1), type : id(L2, "Token"), attributes : none >
                Elts'''),
        rels : (< 036 : Relation | name : a1t_1, type : id(L2, "target"), source : a1_1, target : tr_1, Atts36 >
                < 038 : Relation | name : a1s_1, type : id(L2, "source"), source : a1_1, target : p1_1, Atts48 >
                < 039 : Relation | name : aos_1, type : id(L2, "source"), source : ao_1, target : tr_1, Atts39 >
                < 040 : Relation | name : aot_1, type : id(L2, "target"), source : ao_1, target : po_1, Atts40 >
                < oid(L3, N + 2) : Relation | name : id(L3, N + 3), type : id(L2, "tokens"),
                                              source : po_1, target : id(L3, N + 1), min-mult : 1, max-mult : 1 >
                Rels'''),
        Atts''' >
    < counter : Counter | value : N + 4 >
    Conf }
if L1 < L2 /\ L2 < L3
/\ tr . inArcs -> size() = tr . inRegularArcs -> size() /\ tr . outArcs -> size() = tr . outRegularArcs -> size() .
```

Fig. 12 Excerpt of the Maude rewrite rule corresponding to the box-free version of the *Fire regular transition* MCMT rule

value N + 4 (line 59) since four new identifiers are introduced.

The model changes applied by the rule have been framed to ease its comprehension. Specifically, given a transition tr_1 (line 17) with only one place (notice the weight 1 of the input arc in lines 23–24), and given one of the tokens in it (the token is specified in line 22 and the relation associating it to the place in line 29), the rule removes such a token and creates a new one (lines 49 and 55–56). The number of tokens in the input place is decremented (lines 26 and 48) and the number of tokens in the output place is incremented (lines 21 and 44). Finally, the created token and relation objects (lines 49 and 55–56) have identifiers oid(L3,N), oid(L3,N+1), oid(L3,N+2), and oid(L3,N+3).

3.3 MCMTs in Maude

As illustrated with the rules depicted in Figures 4 and 7, boxes allow us to express very general situations in a quite intuitive way. However, Maude does not provide any mechanism similar to that of MCMT boxes, and therefore the transformation is not as simple. To handle boxes in a generic and efficient way, we use Maude's meta-programming capabilities to unfold boxes at runtime as needed.

If we look at the *Fire regular transition* rule in Figure 4, this time considering its boxes, we know that each time the rule is applied, depending on the specific situation, there will be a number of replicas of each of the boxes. Actually, notice that we may have multiple boxes in both sides, with different cardinalities, and we can have nested boxes, as many times as needed. Note also that these cardinalities are explicitly specified, otherwise, for example, a given transition could be applied taking an arbitrary number of tokens from several of the available input places. These cardinalities could be provided as OCL expressions, which need to be evaluated to get the corresponding value at the time it is required. In this particular case, the transition has tr.inRegularArcs->size() input regular arcs, each of which has a weight a1c which specifies the number of tokens to be removed from it when the transition is fired. Similarly, the transition has tr.outRegularArcs->size() out regular arcs, which tell us the number of output places, each of which has a weight that indicates the number of tokens to be created on that place.

The only assumption that we make to handle boxes is that their cardinality must be greater than zero. In case we want to consider the possibility of zero replications of a box, we need to provide the corresponding rule without such a box. This is the case for the rule in Figure 7. The cases in which we have transitions with no reset, inhibitor or regular arcs must be handled in different Maude rules. Although these cases can be handled by automatically creating the zero-case corresponding rule, we focus here on the general case.

An MCMT rule with boxes produces two Maude rules (plus the corresponding ones for the zero-cardinality cases). Excerpts of the two rules for the MCMT rule *Fire regular transition* in Figure 4 are shown in Figures 13 and 14. The first one of the rules has a left-hand side as if there were no boxes in it. That is, the left-hand side of the rule in Figure 13 is exactly as the left-hand side of the rule in Figure 12. In its condition, the boxes are expanded in a copy of the rule in accordance with the actual match and its application is attempted (lines 15-24 in Figure 13). If such an application succeeds, the result is given as result of the application of the rule, that is, it is used to replace the current system (line 7). If the application in the condition fails, the rule fails. To understand this rule, we need to introduce some additional Maude machinery and some auxiliary functions.

First, in addition to equality checks, Maude rule conditions may include so-called matching equations using the operator :=. Given a pattern term P (a canonical term possibly with free variables) and a term T which may use variables in the left-hand side of the rule and also variables introduced in previous matching conditions, the condition expression P := T evaluates the term T and tries to match its result to the pattern P. If P is a variable, it works like a *let* or *where* clause to assign that value to the variable so that it can later be used. If a more general pattern is used, the match may result in the simultaneous assignment of values to multiple variables. In the rule in Figure 13, we can see how matching conditions are used several times. First, it is used to refer to the left-hand side of the rules as { Conf' } (in lines 9-13), then to refer to the match of the rule as Subst, and finally to get the result of the application of the unfolded rule (line 15).

In Maude, terms and modules have a metarepresentation that we can manipulate as regular terms. Up and down functions allow us to move terms and modules between levels. For instance, given the module GENERIC-PN in which these rules are defined, the expression upModule('GENERIC-PN, false) gives us its metarepresentation. Similarly, upTerm({ Conf' }) gives us the metarepresentation of the term { Conf' }, and downTerm(T, { none }) moves down the result of the application of the unfolded rule T. The built-in function metaApply(M,T,L,S,N) returns the Nth solution of applying rule L in module M on term T using the substitution S to constraint the application of the rule. Then, assuming that the makeModule function takes

```
 1 crl [FireRegularArcs] :
 2   { < level(L1) : Model | ... >                 ---- left-hand side as for the rule without boxes
 3     < level(L2) : Model | ... >
 4     < level(L3) : Model | ... >
 5     < counter : Counter | ... >
 6     Conf }
 7   => downTerm(T, { none })
 8   if (tr . outArcs -> size() = tr . outRegularArcs -> size()) /\ (tr . inArcs -> size() = tr . inRegularArcs -> size())
 9   /\ Conf' := < level(L1) : Model | ... >
10              < level(L2) : Model | ... >
11              < level(L3) : Model | ... >
12              < counter : Counter | ... >
13              Conf
14   /\ Subst := ( ... )                           ---- match of the rule
15   /\ { T, Ty, Subst' } := metaApply(
16                             makeModule(
17                               upModule('GENERIC-PN, false),
18                               upTerm({ Conf' }),
19                               'FireRegularArcsBoxes,
20                               Subst),
21                             upTerm({ Conf' }),
22                             'FireRegularArcsBoxes,
23                             Subst,
24                             0) .
```

Fig. 13 Excerpt of the first of the Maude rewrite rule corresponding to the *Fire regular transition* MCMT rule

```
 1 rl [FireRegularArcsBoxes] :
 2    { < level(L1) : Model | ... >     ---- as the left-hand side of the rule without boxes
 3      < level(L2) : Model | ... >
 4      < level(L3) : Model | ... >
 5      boxes((tr . outRegularArcs -> size()]{ O27, O29, O39, O40 },                        ---- box information
 6              box[tr . inRegularArcs -> size()]{ O32, O34, O36, O38, box[a1c]{ O22, O29 } } ))
 7      < counter : Counter | value : N >
 8      Conf }
 9 =>
10    { < level(L1) : Model | ... >    ---- as the right-hand side of the rule without boxes
11      < level(L2) : Model | ... >
12      < level(L3) : Model | ... >
13      boxes(( box[tr . outRegularArcs -> size()]{ O27, O29, O39, O40, box[aoc]{ oid(L3, N), oid(L3, N + 2)} },
14              box[tr . inRegularArcs -> size()]{ O32, O34, O36, O38 } ))
15      < counter : Counter | value : N + 13 > Conf } .
```

Fig. 14 Excerpt of the second of the Maude rewrite rule corresponding to the *Fire regular transition* MCMT rule

the module with the rules and expands the boxes of the indicated rule as required, the call to metaApply in the last matching condition in the rule in Figure 13 will apply the expanded rule on the current state of the system using the original substitution, that is, forcing the same match. The metaApply functions gives a triple {T, Ty, Subst} as result, where T is the term resulting from the application of the rule, Ty its type, and Subst is the complete substitution used in the application.

The FireRegularArcsBoxes rule is shown in Figure 14. It is similar to the rule without boxes explained in Section 3.2, but notice that it also includes information on the boxes and the level 3 object in the RHS which now represents the TO part (line 12).

Boxes are specified as a collection of terms of the form box[C]{OS}, with C being the cardinality of the box and OS the set of identifiers of the objects (nodes and relations) and nested boxes within the box. The makeModule function is a metalevel function that operates on the metarepresented module, expanding the indicated rule by unfolding the boxes in it. It proceeds recursively, removing one box level at a time. As we have seen above, the cardinality of a box specifies the number of replicas of that box that we need to generate. After a box is expanded, the cardinalities of the next-level boxes can be evaluated. The operation is repeated until no further boxes are left. Once all boxes in a rule are completely expanded, the application of the rule is attempted in one single step.

Notice that boxes in right-hand sides may, and in fact do in the FireRegularArcsBoxes rule in Figure 14, contain identifiers of new objects. Notice also that the counter object is updated according to the number of objects being created, either inside or outside boxes.

4 Execution and Analysis of Models

The capability to execute MultEcore systems and use the powerful tools Maude implements for reachability and model checking of the multilevel models takes our infrastructure to a next step. Furthermore, modelling behavioural languages, such as Petri nets, gets interesting if one can transfer simulation and analysis onto the concrete system models. We use, as case study to demonstrate these capabilities, a gas station model adapted from [26]. We consider the whole modelling cycle, where the modeller sketches and designs the multilevel hierarchy that represents the system, then specify the behaviour by means of transformation rules, and then automatically transforms its setting to Maude where simulation and execution can be done to later verify and analyse the obtained system.

The concrete syntax of the model is depicted in Figure 15. This model would be located at level 4 of the PN hierarchy, as an instance of the *reset-inhibitor-petri-nets* model (Figures 6 and 19(c)). The PN model represents a system in which car tanks get filled up at a gas station. The station has a tank with a maximum capacity, which can be evacuated for cleaning reasons and then replenish. If there is no car in the station, a new car can arrive and set its indicator on. Once the car's tank is filled, it leaves the station.

The initial marking of the model, depicted in Figure 15, has 4 tokens in the place Station Tank, which is its full capacity. For a car tank to be refuelled, there must be a token in the Fuel Indicator On place. This can only happen if the transition Turn Fuel Indicator On has been fired, and for this to happen, there cannot be any token neither in the Car Tank place (the tank is empty) nor in the Fuel Indicator On place (the indicator is off). This is modelled by the two inhibitor arcs connected to the Turn Fuel Indicator On transition. Then, once the indicator has been turned on, we can only progress by the firing Fuel Car transition, which makes the Car Tank to be filled (i.e., gets one token). Ultimately, once the car tank is full, the Leave Station transition can be fired, leading the car to exit the station by putting a token into the Outside Station place.

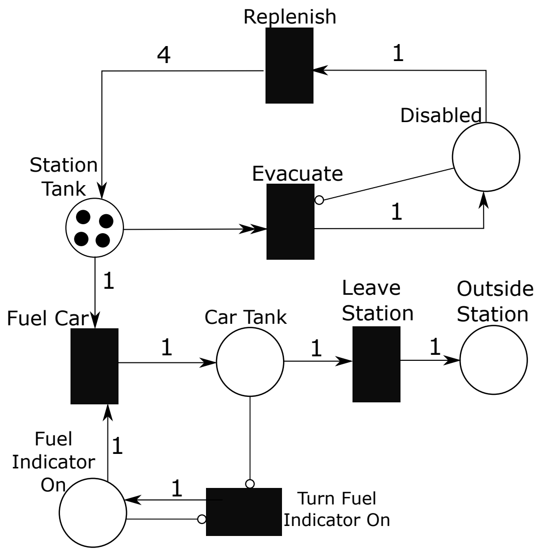


Fig. 15 Gas station model with initial marking

4.1 Execution and analysis using Maude

Given a model with an initial marking, we can simulate it. In fact, we have two ways of doing so. We can let the default strategy choose the rules to apply, and the way in which to apply them, or we can force a specific sequence of rule applications.

Rule rewriting is a highly non-deterministic process, and in general, at every step many rules could be applied. Moreover, since a rewrite system may be non-terminating, as is the case of the gas station example, a maximum number of rewriting steps to be taken may be specified.

A finer control on rule application may sometimes be desirable. In MultEcore, we may specify the sequence of rules to be applied, which can be selected from a list of possible ones. Let us show an example. But first, as we said in Section 3.3, for any rule with boxes, several Maude rules are generated, corresponding to the different combinations of boxes with cardinality zero. Thus, for the *Fire reset/inhibitor transition* rule in Figure 7, seven rules are generated, for transitions with no regular arcs (FireResetInhibitorArcs), for transitions with no inhibitor and no reset arcs (FireRegularArcs), for transitions with no reset and no regular arcs (FireInhibitorArcs), etc. Then, we can specify the strategy with which we desire to rewrite our initial marking model by indicating the corresponding sequence of rule labels. For example, we can guide the execution from an initial marking given the following sequence:

```
FireInhibitorArcs   --------- Fuel Indicator On
FireRegularArcs  ----------- 1st car fuelled (Car Tank)
FireRegularArcs  ----------- 1st car leaves (Outside Station)
FireInhibitorArcs   --------- Fuel Indicator On
FireRegularArcs  ----------- 2nd car fuelled (Car Tank)
FireRegularArcs  ----------- 2nd car leaves (Outside Station)
FireResetInhibitorArcs ---- Disabled
FireRegularArcs  ----------- Station tank filled
```

The comments on the right indicate the corresponding effect on the model. The MultEcore integration with Maude allows us to specify sequence of rules to automatically generate the desired model state and get its graphical representation right away. A screenshot of the MultEcore tool is depicted in Figure 17, in which we can see how rule labels are selected from a list corresponding to, the above sequence of rules. Note that the MultEcore syntax of the initial state (depicted in Petri nets syntax in Figure 15) is in the background

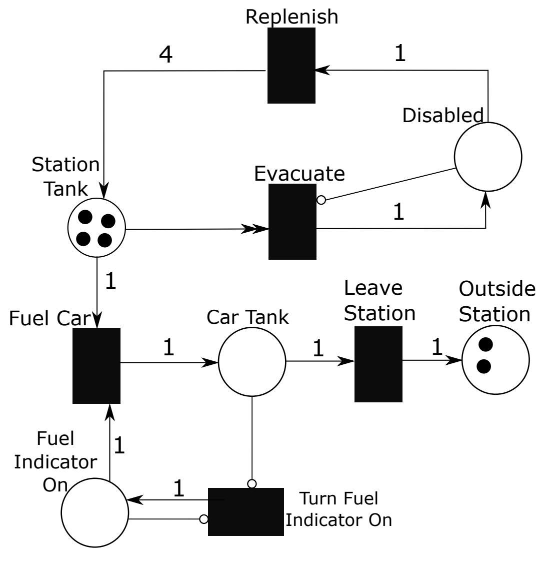


Fig. 16 Resulting Gas station state

of Figure 17. Pressing Finish on such a wizard automatically provides the model state in the MultEcore syntax, shown in Figure 18, where we have four tokens (token19211...token19214) in the Station Tank place, representing that the station tank has been replenished (highlighted at the top-left of Figure 18), and four tokens in the Outside Station place, representing that 4 cars have left the station (highlighted at the bottom-right of Figure 18).

4.2 Reachability analysis

To perform a more exhaustive verification of the model, we can perform reachability analysis and bounded model-checking of invariants, with which we can check safety properties. Specifically, we can study the reachability of given states using the *search* command, where the states to check can be specified both using patterns or conditions on the states. The search command explores the reachable state space following a breadth-first strategy.

To carry on a search, we need to provide: (i) the model from which to initiate the search, (ii) the maximum depth of the search (even for terminating systems, a search may take a long time), (iii) the pattern model to be reached (a model with variables), and (iv) an optional property that has to be satisfied by the reached state. Since the structure of the PN does not change along the execution, we can specify our pattern model leaving as variables the tokens in each place. Then, the condition to satisfy at the target state may be specified as an OCL expression. As result, MultEcore will determine whether such a state is reachable, in the specified number of steps, and if so, it may provide the specific path leading to the state found.

Let us see how we can use the search command to verify properties on the gas station example. For example, we can verify that, starting from the marking depicted in Figure 15, that the system can reach a marking where four cars have left the station and the station tank is again full to continue fuelling further cars. To do that, we just need to select the initial model, the target pattern model, and write, for example, the following OCL boolean expression:

```
id(4, "Station Tank").tokens->size() = 4
and id(4, "Outside Station").tokens->size() = 4
```

MultEcore responds positively, and provides the sequence of rule names leading to such a solution:

```
FireInhibitorArcs
FireRegularArcs
FireRegularArcs
FireInhibitorArcs
FireRegularArcs
FireRegularArcs
FireInhibitorArcs
FireRegularArcs
FireRegularArcs
FireInhibitorArcs
FireRegularArcs
FireRegularArcs
FireResetInhibitorArcs
FireRegularArcs
```

Notice that this is the path to one of the possible states satisfying this condition, which, like in this case, may be not unique.

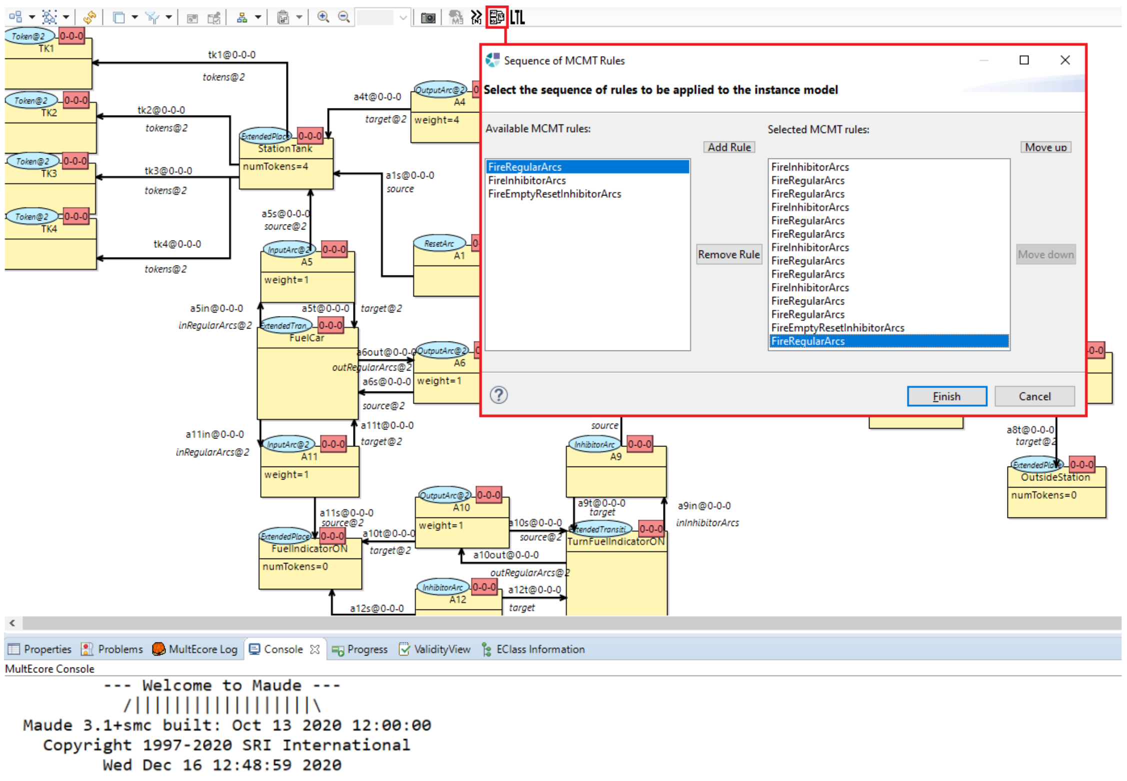
We can also check whether certain miss-behaviours may occur. Below we list properties we have verified in order to assure the correctness of the model:

– **Property 1.** It is not possible to find a state where, simultaneously, the places Station Tank and Disabled contain the tokens. This would imply that either Evacuate or Replenish transition could be fired more than once in a row, which is not the behaviour we expect for this specific system. We can check this property using the same initial and pattern markings as before together with the following property:

  ```
  id(4, "Station Tank").tokens->size() > 0
  and id(4, "Disabled").tokens->size() > 0
  ```

  Since the system is not terminating, and we expect not to find it, we specify a bound, for example, of 20. The answer from MultEcore is ‘false’, indicating that such a state cannot be reached within the given depth.

– **Property 2.** It is not possible to find a state where either the Fuel Indicator ON, Car Tank, and Disabled places have more than one token. In other words,



Sequence of MCMT Rules

Select the sequence of rules to be applied to the instance model

Available MCMT rules:

FireRegularArcs
FireInhibitorArcs
FireEmptyResetInhibitorArcs

Selected MCMT rules:

FireInhibitorArcs
FireRegularArcs
FireRegularArcs
FireInhibitorArcs
FireRegularArcs
FireRegularArcs
FireInhibitorArcs
FireRegularArcs
FireRegularArcs
FireInhibitorArcs
FireRegularArcs
FireRegularArcs
FireEmptyResetInhibitorArcs
FireRegularArcs

Add Rule | Remove Rule | Move up | Move down | Finish | Cancel

```
        --- Welcome to Maude ---
          /||||||||||||||||||\
  Maude 3.1+smc built: Oct 13 2020 12:00:00
    Copyright 1997-2020 SRI International
          Wed Dec 16 12:48:59 2020
```

Fig. 17 MultEcore screenshot for sequence of rules-based execution

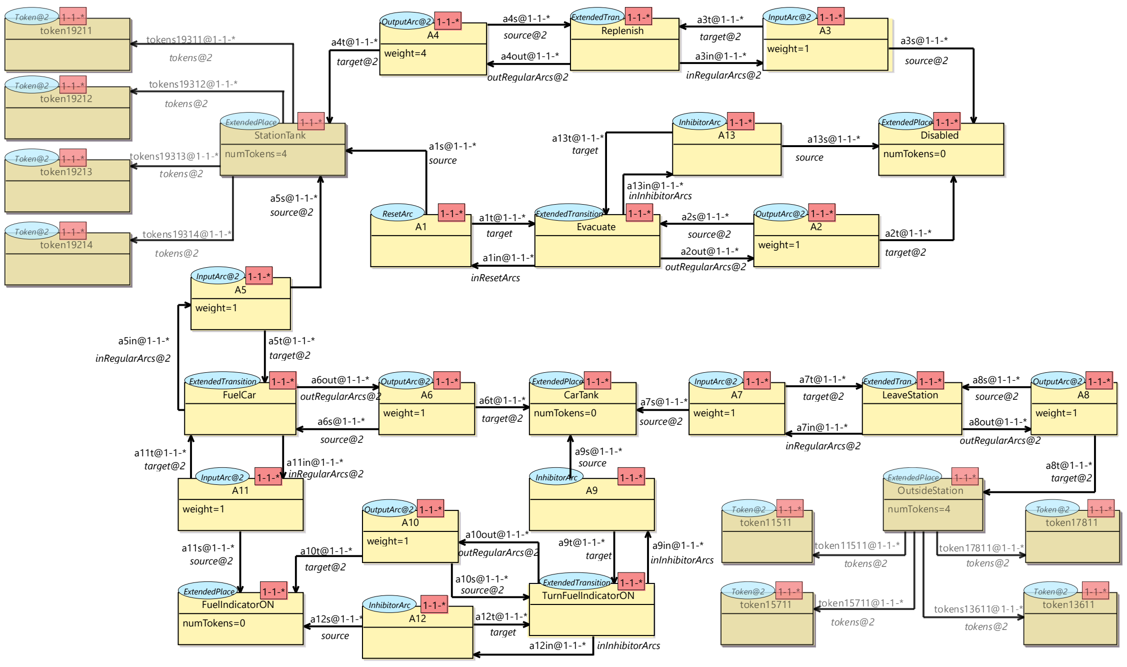


Fig. 18 MultEcore syntax of the Gas station model state after execution

the indicator is either ON or OFF, the car tank is either full or not, and we cannot disable the station consecutively two times. This property can be verified as the previous one using the following OCL expression:

```
id(4, "Fuel Indicator ON").tokens->size() > 1
or id(4, "Car Tank").tokens->size() > 1
or id(4, "Disabled").tokens->size() > 1
```

Again, for a bound of 20, the answer from MultEcore is 'false'.

– **Property 3.** We can find a state where 5 cars have successfully exited the station. If 5 cars can exit the station, then any number of cars can leave as well.

```
id(4, "Outside Station").tokens->size() > 5
```

The answer obtained is positive, and the path to the first found solution is provided.

All the properties listed above have provided the expected results, which further validates our model.

4.3 System abstraction for unbounded analysis

In the previous section we have carried out bounded model checking of several behavioural properties. Although limited to a maximum depth, the verification of the properties checked using the search command greatly increase the confidence in the correctness of the system. However, bounded model checking is an incomplete procedure, since a counterexample could exist at greater depth. The problem in this case is that the state space is infinite, and therefore, we cannot complete the analysis in this way. One way to fully verify the system is using a finite-state abstraction of it, that is, on an appropriate quotient of the original system whose set of reachable states is finite. The method proposed in [43] creates an abstraction of the original system by adding a set of equations to collapse the infinite set of reachable states into a finite set. The specification, extended with these equations, need to still satisfy the usual executability conditions — the equations must be ground Church-Rosser and terminating, and the rules should be ground coherent with them — but the procedure is quite simple, and the abstraction is then correct by construction. The method is valid both for the verification of invariants and LTL formulas with an additional deadlock-freedom requirement. Indeed, an automatic procedure to complete specifications so that the requirement is satisfied has been given in [15].

The key idea about abstraction for invariant verification is that if we can verify an invariant on the abstracted specification — the specification with the equations defining the abstraction — then it also holds in the original specification. The implication, however, only works in one direction, if we find a counterexample in the abstracted system it does not necessarily mean that a counterexample exists for the original system.

Let us apply the technique to our example. There are several reasons why our Petri nets system is infinite. On the one hand, we have that tokens get accumulated in the Outside Station place, since every time a car gets its tank filled, the car leaves the station and a new token is added to the place. On the other hand, since we are representing tokens as objects, every time a new token is created it gets a new unique identifier. Thus, even though from the Petri nets point of view, tokens are anonymous dots in a place, in our representation tokens are objects that have names and identifiers, and the counter object keeps getting its value attribute increased. We may, however, abstract from this information, since neither the tokens' names nor identifiers are relevant, nor are we really concerned about the number of tokens we have in the Outside Station place. For the operation of the Petri net, the number of tokens in the other places is not relevant either. Specifically, since all arc weights are one, the analysis would be the same if having four or three tokens in the Station Tank place.

We introduce equations that abstract the Petri net system is the following way: (1) The Outside Station place gets its number of tokens decremented if it is bigger than one, (2) the number of tokens in the Station Tank place becomes 3 if it gets 4 tokens, (3) names and identifiers of token objects are reset into a range of values not used by the counter object, and (4) the counter object gets its value attribute restarted to its initial value. Notice that when we eliminate or rename a token, we also act on the relation object associating it to the place in which it is located. In this way, we not only make all places to have either zero or one tokens in them, but names and identifiers are reused from a small set of possible values.

This abstraction makes the state space finite. And we can use it to verify LTL formulas on the abstracted model. Temporal logic allows the specification of safety properties (something bad never happens) and liveness properties (something good eventually happens), which are related to the infinite behaviour of a system. However, we need a few additional definitions first.

Kripke structures are the natural models for propositional temporal logic. We need to understand how a Kripke structure is associated to the rewrite theory specified by a Maude system module. Basically, a Kripke structure is a (total) transition system to which we have added a collection of unary state predicates on its set of states. Therefore, since the models of rewriting logic are also transition systems, we need to make explicit the type of each of the states (System in our specification) and the atomic propositions on which we define our state predicates. In our case, again, we use OCL boolean expressions as basic propositions, associating to each state those boolean expressions that are satisfied in such a state. For example, we may check the following properties

– **Property 1.** To check the property stating that it is always true that eventually the system gets to a state in which there is a token on either the Disabled or the Outside Station place can be checked using the following LTL formula:

  ```
  [] <> (id(4, "Disabled").tokens->size() > 0
          \/ id(4, "Outside Station").tokens->size() > 0)
  ```

  In this case, the answer obtained is positive.
– **Property 2.** The following LTL formula states that if we reach a state in which there is a token in the Fuel Indicator On place, then eventually a state in which there is a token in the Outside Station place is reached.

  ```
  [] (id(4, "Fuel Indicator ON").tokens->size() > 0
     -> []<> id(4, "OutsideStation").tokens->size() > 0)
  ```

  In this case the response is negative. Indeed, it may happen that the Petri net loops in the upper part, evacuating and replenishing the station tank over and over again. As usual, if the formula is not true, the model checker gives a counterexample, in the form of a sequence of states.

The interested reader can find the complete outputs in [54], together with the source files used to reproduce the execution.

5 Related Work

Even though there exist a plethora of MLM approaches and tools, only a few of them support DSML behaviour specification and execution. Melanee [2] is one of the most advanced tools for MLM. The tool supports a variant of OCL with deep semantics (DeepOCL) which has been integrated with the Atlas Transformation Language (ATL) for model transformations. Lange shows in [31] how this tool can be used to check constraints spanning multiple classification levels which can be defined and executed. Although Melanee itself is not natively supporting tools for simulation/execution through the specification of the execution semantics, i.e., (multilevel) transformation rules, there are some works on top of it that aims to achieve this (see, e.g., [3]). In that work, the model execution mechanism is based on a service API and a plug-in mechanism, and the communication between the modelling and the execution environments is realised using socket-based communication. We provide in our approach the whole set of tools necessary to directly be able to define the structure of the multilevel hierarchy, specify the multilevel model transformation rules (MCMTs), execute/simulate the models, and analyse the system.

The MetaDepth tool [32] is a well-known framework within the MLM community. It is integrated with the Epsilon languages [20], which permits using the Epsilon Object Language (EOL) as an action language to define behaviour for metamodels, as well as the Epsilon Validation Language (EVL) for expressing constraints. Both EOL and EVL are extensions of OCL. The approach implements the interface of the connectivity layer in a way to make EOL aware of the multiple ontological levels providing it with a multilevel nature. However, the authors of [32] state that MetaDepth can be used as a normal two-level meta-modelling environment when it comes to the execution of behaviour of the models. Thus, for the actual execution they would have to flatten their multilevel language to a two-level version in order to run the models. To the best of our knowledge there is not yet a MLM tool that supports or integrates model checking and analysis capabilities within its MLM tool-set.

Other authors have attempted to handle pattern identification and specification to define reusable model transformation rules. There exist a diverse set of approaches that bring solutions to pattern definition and application in the context of graph transformations. In [24], Guerra and de Lara explore *recursion* as a graph transformation mechanism. They provide double pushout (DPO) rules with base and recursive conditions, together with mechanisms to pass the matching between successive recursion steps. Lindqvist et al. [36] propose the star operator, which is suited to find repetitive occurrences of a specific modelling pattern. However, the star operator is only defined for matching model extracts, and not to perform transformations. In [23], Grønmo, Krogdahl, and Møller-Pedersen present a collection operator for graph transformation and show its usage to a variety of Coloured Petri nets. Using this operator, it is possible to match several similar structures within the model. They theoretically define how nesting would work by producing an ad-hoc rule that would fit to the specific case. We follow a similar approach by defining the rule in a generic way and then the transformation engine provides also a generic version in the Maude specification. The advantage of our approach with respect to the collection operator is that we do not physically produce an unfolded rule, but it is dynamically unfolded and used at run-time. It is during the matching at run-time when the rule is unfolded guided by the cardinalities provided in the rule. In [50], Rensink and Kuperus propose a transformation language that uses an amalgamation scheme for nested graph transformation rules, where pattern elements are combined with universal and existing quantifiers. The transformation language is used in the GROOVE tool. Henshin [60] is an in-place model transformation language for the EMF. Among other features, it implements a *rule-nesting* mechanism [1] that provides a for-each operator for rules. In nested rules, the outer rule is referred to as *kernel rule* and the inner rule as *multi rule*. During execution of a nested rule, the kernel rule is matched and executed once. Afterwards, the match is used as a starting point to match the multi-rule as often as possible and execute it for each match.

There are several traditional MDSE approaches that deal with execution and verification that are somehow related to our proposal. In [52], Rivera et at. use Maude to represent 2-level models to be able to simulate and perform formal analysis and model checking on them. Such work served us as inspiration and starting point. We were considering either implementing ourselves an execution engine within MultEcore or using an existing tool where to rely on. Studying some works where Maude was used and analysing how the language could be customised together with its plethora of existing capabilities made us to follow such a path. We also analysed other mature tools in the context of model execution via operational semantics, e.g., Henshin [60] or the GEMOC Studio [16,10], which helped us to understand how the user could interact with the execution tools from an Eclipse-based application. In the context of verification, GROOVE (GRaph-based Object-Oriented VErification) [48,22] is a tool for software model checking of object-oriented systems. It can be used for modelling, analysis and verification and integrates all these functionalities in an easy to use interactive GUI. While we already integrate into MultEcore some Maude functionalities for execution and verification, we still have to work in this direction to ease the process to the modeller. We see GROOVE as a good influence to achieve a better usability degree in MultEcore.

6 Conclusions and Future Work

In this paper have presented an infrastructure for execution and analysis of multilevel modelling languages.

To make this possible, we have integrated Maude into our tool MultEcore, making it possible not only to define our multilevel hierarchy (language) and specify the behaviour by means of MCMTs, but also carry on simulation and further model checking and analysis techniques. However, Maude is used as a backend tool, hidden to the user such that the interaction is entirely done with MultEcore, making the modeller unaware of the Maude details. Although in the last years several traditional two-level tools have provided support for the whole cycle of behavioural DMSLs (from design to simulation and verification), to the best of our knowledge this is the first work where a MLM tool incorporates capabilities to perform model checking and other formal analyses. We believe that the work presented in this paper can open new doors to the MLM field, as this tool can be used to define behavioural multilevel DSMLs, execute them, and verify them.

Apart from the major improvements in the infrastructure itself, we have improved and extended the MCMTs capabilities, by allowing nested boxes to represent collections, incorporated attribute manipulations, and specification of conditions. Basic support for OCL is also provided, which is very useful for the manipulation of attributes, the specification of box cardinalities, the specification of rule conditions, and the specification of expressions and conditions to be used in the formal checks, including LTL formulas.

To validate and demonstrate that our infrastructure works and that actual execution and analysis can be carried on, we have provided a case study where a Petri net model that captures a gas station is simulated applying consecutively the MCMT rules defined in MultEcore. The goal of this case study has been to evaluate the usability and practicability of the developed infrastructure. We are already considering how to evaluate our tool against other MLM approaches that allow the modeller to perform execution on models by defining in-place model transformations. We have validated and verified the modelled system using reachability analysis and model checking techniques on an abstracted version of the model. We refer the reader to the main MultEcore webpage [53] for further details and examples.

We plan to integrate into MCMTs the capabilities that a programming language brings such as reasoning about functions, expressions, type specifications, and data manipulations. Our current implementation of certain OCL functions represents a step towards this goal. We are already working on adapting the complete mOdCL (Maude + OCL) [19] to our Multilevel infrastructure so we can make use of the full power of OCL in a Multilevel context. This would allow us, for instance, to specify Coloured Petri nets [27] which combine classical Petri nets with a programming language [62].

While MCMTs are flexible with respect to further horizontal/vertical extensions, we identify a key point of improvement as being able to reuse META levels on

MCMTs into other rules. This would improve our approach with a higher degree of modularity and reusability. Ultimately, we plan to further advance on the interface that connects MultEcore to Maude, bringing more advanced functionalities such as an interactive editor, a smoother experience to the user with the graphical editor and additional Maude capabilities to, for instance, customise the strategy language and have more control on the execution and analysis.

References

1. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In: 13th International Conference on Model Driven Engineering Languages and Systems (MODELS 2010)., pp. 121–135 (2010). DOI 10.1007/978-3-642-16145-2_9
2. Atkinson, C., Gerbig, R.: Flexible Deep Modeling with Melanee. In: S. Betz, U. Reimer (eds.) Modellierung 2016, *LNI*, vol. 255, pp. 117–122. Gesellschaft für Informatik, Bonn (2016)
3. Atkinson, C., Gerbig, R., Metzger, N.: On the Execution of Deep Models. In: 1st International Workshop on Executable Modeling co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS 2015)., pp. 28–33 (2015)
4. Atkinson, C., Kühne, T.: Processes and products in a multi-level metamodeling architecture. International Journal of Software Engineering and Knowledge Engineering **11**(06), 761–783 (2001)
5. Atkinson, C., Kühne, T.: The Essence of Multilevel Metamodeling. In: «UML» 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, pp. 19–33 (2001). DOI 10.1007/3-540-45441-1_3
6. Atkinson, C., Kühne, T.: Reducing accidental complexity in domain models. Software & Systems Modeling **7**(3), 345–359 (2008)
7. Atkinson, C., Kühne, T.: In defence of deep modelling. Inf. Softw. Technol. **64**, 36–51 (2015). DOI 10.1016/j.infsof.2015.03.010
8. Atkinson, C., Kühne, T.: On Evaluating Multi-level Modeling. In: Proceedings of MULTI @ MODELS, pp. 274–277 (2017)
9. Bernardinello, L., de Cindio, F.: A survey of basic net models and modular net classes. In: Advances in Petri Nets 1992, The DEMON Project, pp. 304–351. Springer (1992). DOI 10.1007/3-540-55610-9_177
10. Bousse, E., Wimmer, M.: Domain-level observation and control for compiled executable dsls. In: M. Kessentini, T. Yue, A. Pretschner, S. Voss, L. Burgueño (eds.) 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2019, Munich, Germany, September 15-20, 2019, pp. 150–160. IEEE (2019). DOI 10.1109/MODELS.2019.000-6
11. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers (2012). DOI 10.2200/S00441ED1V01Y201208SWE001
12. Cabot, J., Gogolla, M.: Object constraint language (OCL): A definitive guide. In: M. Bernardo, V. Cortellessa, A. Pierantonio (eds.) Formal Methods for Model-Driven Engineering - 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures, *Lecture Notes in Computer Science*, vol. 7320, pp. 58–90. Springer (2012). DOI 10.1007/978-3-642-30982-3_3
13. Clark, T., Warmer, J.: Object Modeling With the OCL: The Rationale Behind the Object Constraint Language, vol. 2263. Springer (2003)
14. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martı-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: Specification and programming in rewriting logic. Theoretical Computer Science **285**(2), 187–243 (2002)
15. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L. (eds.): All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, *Lecture Notes in Computer Science*, vol. 4350. Springer (2007). DOI 10.1007/978-3-540-71999-1
16. Combemale, B., Barais, O., Wortmann, A.: Language engineering with the GEMOC studio. In: 2017 IEEE International Conference on Software Architecture Workshops, ICSA Workshops 2017, Gothenburg, Sweden, April 5-7, 2017, pp. 189–191. IEEE Computer Society (2017). DOI 10.1109/ICSAW.2017.61
17. Durán, F., Eker, S., Escobar, S., Martí-Oliet, N., Meseguer, J., Rubio, R., Talcott, C.L.: Programming and symbolic computation in Maude. J. Log. Algebraic Methods Program. **110** (2020). DOI 10.1016/j.jlamp.2019.100497. URL `https://doi.org/10.1016/j.jlamp.2019.100497`
18. Durán, F., Garavel, H.: The rewrite engines competitions: A rectrospective. In: D. Beyer, M. Huisman, F. Kordon, B. Steffen (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Proceedings, Part III, *Lecture Notes in Computer Science*, vol. 11429, pp. 93–100. Springer (2019). DOI 10.1007/978-3-030-17502-3_6. URL `https://doi.org/10.1007/978-3-030-17502-3_6`
19. Durán, F., Roldán, M.: Validating OCL constraints on Maude prototypes of UML models. Tech. rep., Universidad de Málaga (2012)
20. The Epsilon Object Language (EOL). https://www.eclipse.org/epsilon/doc/eol/
21. Garavel, H., Tabikh, M., Arrada, I.: Benchmarking implementations of term rewriting and pattern matching in algebraic, functional, and object-oriented languages - the 4th rewrite engines competition. In: V. Rusu (ed.) Rewriting Logic and Its Applications - 12th International Workshop, WRLA 2018, Held as a Satellite Event of ETAPS, Proceedings, *Lecture Notes in Computer Science*, vol. 11152, pp. 1–25. Springer (2018). DOI 10.1007/978-3-319-99840-4_1. URL `https://doi.org/10.1007/978-3-319-99840-4_1`
22. Ghamarian, A.H., de Mol, M., Rensink, A., Zambon, E., Zimakova, M.: Modelling and analysis using GROOVE. Int. J. Softw. Tools Technol. Transf. **14**(1), 15–40 (2012). DOI 10.1007/s10009-011-0186-x
23. Grønmo, R., Krogdahl, S., Møller-Pedersen, B.: A collection operator for graph transformation. Software and Systems Modeling **12**(1), 121–144 (2013). DOI 10.1007/s10270-011-0190-3
24. Guerra, E., de Lara, J.: Adding Recursion to Graph Transformation. ECEASST **6** (2007). DOI 10.14279/tuj.eceasst.6.56
25. Halder, A., Venkateswarlu, A.: A study of petri nets modeling analysis and simulation. Department of Aerospace

Engineering Indian Institute of Technology Kharagpur, India (2006)
26. Hee, van, K., Leurs, M., Post, R.: Yasper : Yet another smart process editor (poster). In: 2005 Symposium on Verification and validation of software systems (VVSS 2005) (2005)
27. Jensen, K., Kristensen, L.M., Wells, L.: Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. International Journal on Software Tools for Technology Transfer **9**(3), 213–254 (2007). DOI 10.1007/s10009-007-0038-x
28. Kelly, S., Tolvanen, J.: Domain-Specific Modeling - Enabling Full Code Generation. Wiley (2008)
29. Kühne, T.: Exploring Potency. In: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS, pp. 2–12 (2018). DOI 10.1145/3239372.3239411
30. Kühne, T.: A story of levels. In: Proceedings of MULTI @ MODELS, pp. 673–682 (2018)
31. Lange, A.: dACL: the deep constraint and action language for static and dynamic semantic definition in Melanee (2016). URL `https://madoc.bib.uni-mannheim.de/43490/`. Unpublished
32. de Lara, J., Guerra, E.: Deep meta-modelling with MetaDepth. In: Objects, Models, Components, Patterns, *lncs*, vol. 6141, pp. 1–20. springer (2010). DOI 10.1007/978-3-642-13953-6_1
33. de Lara, J., Guerra, E.: Generic Meta-modelling with Concepts, Templates and Mixin Layers. In: Model Driven Engineering Languages and Systems - 13th International Conference, MODELS, pp. 16–30 (2010). DOI 10.1007/978-3-642-16145-2_2
34. de Lara, J., Guerra, E., Cuadrado, J.S.: When and how to use multilevel modelling. ACM Transactions on Software Engineering and Methodology (TOSEM) **24**(2), 12 (2014)
35. de Lara, J., Vangheluwe, H.: AToM 3: A Tool for Multi-formalism and Meta-modelling. In: International Conference on Fundamental Approaches to Software Engineering, pp. 174–188. Springer (2002)
36. Lindqvist, J., Lundkvist, T., Porres, I.: A Query Language With the Star Operator. ECEASST **6** (2007). DOI 10.14279/tuj.eceasst.6.55
37. Macías, F.: Multilevel modelling and domain-specific languages. PhD thesis, Western Norway University of Applied Sciences and University of Oslo (2019)
38. Macías, F., Rutle, A., Stolz, V.: Multilevel Modelling with MultEcore: A Contribution to the MULTI 2017 Challenge. In: Proceedings of MULTI @ MODELS, pp. 269–273 (2017)
39. Macías, F., Wolter, U., Rutle, A., Durán, F., Rodriguez-Echeverria, R.: Multilevel Coupled Model Transformations for Precise and Reusable Definition of Model Behaviour. Journal of Logical and Algebraic Methods in Programming **106**, 167–195 (2019). DOI 10.1016/j.jlamp.2018.12.005
40. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theor. Comput. Sci. **96**(1), 73–155 (1992). DOI 10.1016/0304-3975(92)90182-F
41. Meseguer, J.: Conditioned rewriting logic as a united model of concurrency. Theor. Comput. Sci. **96**(1), 73–155 (1992). DOI 10.1016/0304-3975(92)90182-F
42. Meseguer, J.: Twenty years of rewriting logic. J. Log. Algebr. Program. **81**(7-8), 721–781 (2012). DOI 10.1016/j.jlap.2012.06.003
43. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational abstractions. Theor. Comput. Sci. **403**(2-3), 239–264 (2008). DOI 10.1016/j.tcs.2008.04.040. URL `https://doi.org/10.1016/j.tcs.2008.04.040`
44. Meta Object Facility (MOF) specification 2.5.1. https://www.omg.org/spec/MOF
45. Mohagheghi, P., Gilani, W., Stefanescu, A., Fernández, M.A., Nordmoen, B., Fritzsche, M.: Where does model-driven engineering help? Experiences from three industrial cases. Software & Systems Modeling **12**(3), 619–639 (2013)
46. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77**(4), 541–580 (1989)
47. Reisig, W.: Understanding Petri Nets - Modeling Techniques, Analysis Methods, Case Studies. Springer (2013). DOI 10.1007/978-3-642-33278-4
48. Rensink, A.: The GROOVE simulator: A tool for state space generation. In: J.L. Pfaltz, M. Nagl, B. Böhlen (eds.) Applications of Graph Transformations with Industrial Relevance, Second International Workshop, AGTIVE 2003, Charlottesville, VA, USA, September 27 - October 1, 2003, *Lecture Notes in Computer Science*, vol. 3062, pp. 479–485. Springer (2003). DOI 10.1007/978-3-540-25959-6_40
49. Rensink, A.: The GROOVE simulator: A tool for state space generation. In: International Workshop on Applications of Graph Transformations with Industrial Relevance, pp. 479–485. Springer (2003)
50. Rensink, A., Kuperus, J.: Repotting the Geraniums: On Nested Graph Transformation Rules. Electronic Communication of the European Association of Software Science and Technology **18** (2009). DOI 10.14279/tuj.eceasst.18.260
51. Rivera, J.E., Durán, F., Vallecillo, A.: A graphical approach for modeling time-dependent behavior of DSLs. In: Visual Languages and Human-Centric Computing, 2009. VL/HCC 2009. IEEE Symposium on, pp. 51–55. IEEE (2009)
52. Rivera, J.E., Durán, F., Vallecillo, A.: Formal Specification and Analysis of Domain Specific Models Using Maude. Simulation **85**(11-12), 778–792 (2009). DOI 10.1177/0037549709341635
53. Rodríguez, A., Durán, F., Kristensen, L.M.: MultEcore webpage (2021). URL `https://ict.hvl.no/multecore/`
54. Rodríguez, A., Durán, F., Kristensen, L.M.: Petri nets experiment resources: MultEcore and Maude files (2021). URL `https://bitbucket.org/phdalejandro/no.hvl.multecore.examples.sosym.petrinets`
55. Rodríguez, A., Durán, F., Rutle, A., Kristensen, L.M.: Executing Multilevel Domain-Specific Models in Maude. Journal of Object Technology **18**(2), 4:1–21 (2019). DOI 10.5381/jot.2019.18.2.a4
56. Rodríguez, A., Macías, F.: Multilevel Modelling with MultEcore: A Contribution to the MULTI Process Challenge. In: Proceedings of MULTI @ MODELS, pp. 152–163 (2019). DOI 10.1109/MODELS-C.2019.00026
57. Rodríguez, A., Rutle, A., Kristensen, L.M., Durán, F.: A Foundation for the Composition of Multilevel Domain-Specific Languages. In: MULTI@ MoDELS, pp. 88–97 (2019). DOI 10.1109/MODELS-C.2019.00018
58. Roldán, M., Durán, F.: Dynamic validation of OCL constraints with mOdCL. Electron. Commun. Eur. Assoc. Softw. Sci. Technol. **44** (2011). DOI 10.14279/tuj.eceasst.44.625
59. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse Modeling Framework. Pearson Education (2008)
60. Strüber, D., Born, K., Gill, K.D., Groner, R., Kehrer, T., Ohrndorf, M., Tichy, M.: Henshin: A Usability-Focused Framework for EMF Model Transformation Development. In: 10th International Conference, ICGT 2017, pp. 196–208 (2017). DOI 10.1007/978-3-319-61470-0_12

61. Syriani, E., Vangheluwe, H., Mannadiar, R., Hansen, C., Van Mierlo, S., Ergin, H.: AToMPM: A Web-based Modeling Environment. In: MODELS-JP 2013, *CEUR Workshop Proceedings*, vol. 1115, pp. 21–25 (2013)
62. Ullman, J.D.: Elements of ML programming. Prentice-Hall, Inc. (1994)
63. The Unified Modelling Language (UML) specification 2.5.1. https://www.omg.org/spec/UML
64. Van Mierlo, S., Barroca, B., Vangheluwe, H., Syriani, E., Kühne, T.: Multi-level modelling in the Modelverse. In: MULTI@ MoDELS, *CEUR Workshop Proceedings*, vol. 1286, pp. 83–92 (2014)
65. Verbeek, H.M.W., Wynn, M.T., van der Aalst, W.M.P., ter Hofstede, A.H.M.: Reduction rules for reset/inhibitor nets. J. Comput. Syst. Sci. **76**(2), 125–143 (2010). DOI 10.1016/j.jcss.2009.06.003
66. Warmer, J., Kleppe, A.: The Object Constraint Language Second Edition: Getting Your Models Ready for MDA. Addison-Wesley Educational Publishers (2003)

A Petri nets multilevel hierarchy

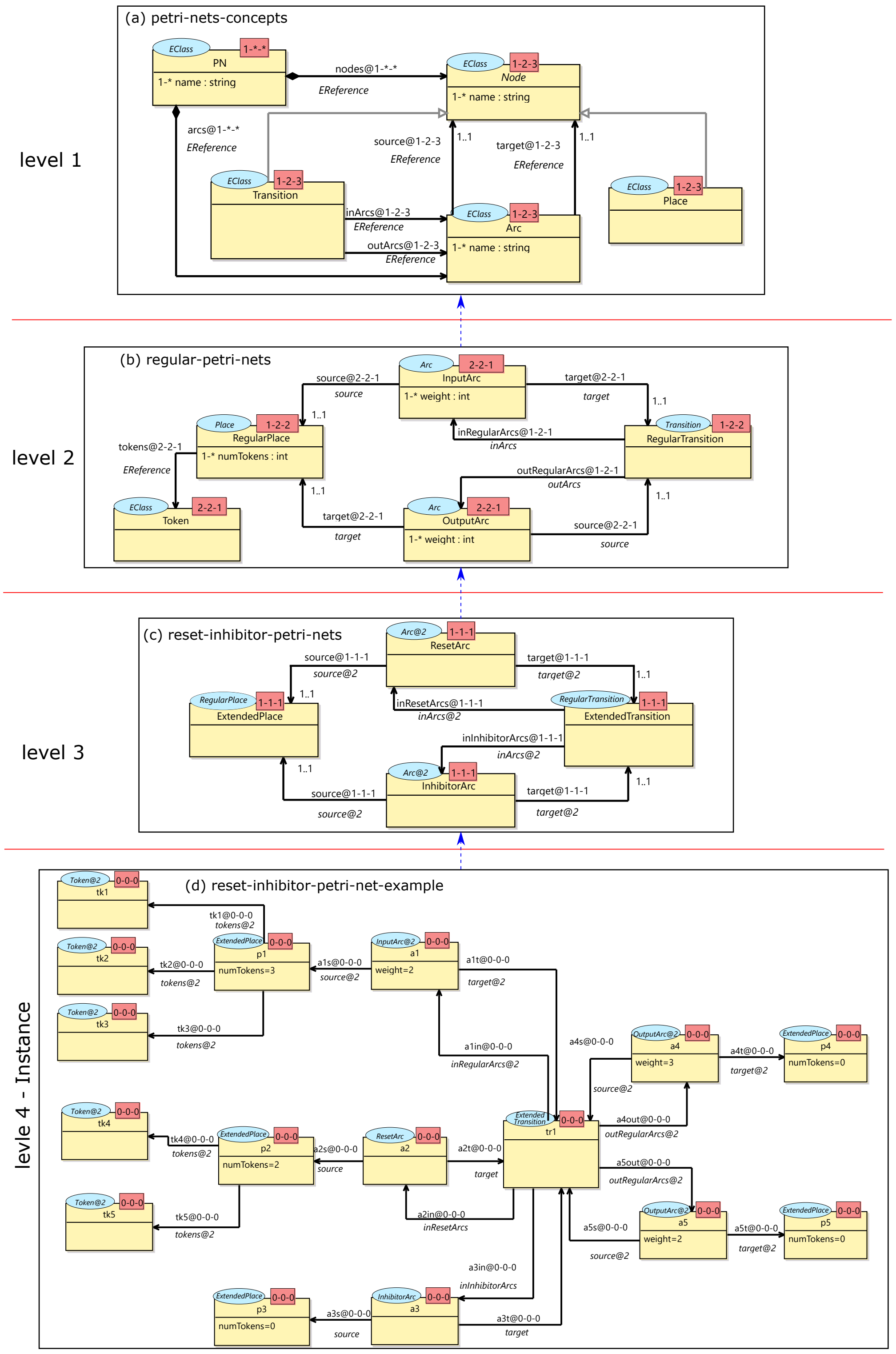


Fig. 19 Petri nets multilevel hierarchy

PAPER F

MULTILEVEL MODELLING WITH MULTECORE: A CONTRIBUTION TO THE MULTI-LEVEL PROCESS CHALLENGE

Alejandro Rodríguez and Fernando Macías

In Enterprise Modelling and Information Systems Architectures Journal **The German Informatics Society**, 2021. (Submitted)

Multilevel Modelling with MultEcore: A contribution to the Multi-Level Process Challenge

Alejandro Rodríguez*,[a], Fernando Macías[b]

[a] Western Norway University of Applied Sciences, Bergen, Norway
[b] IMDEA Software Institute, Madrid, Spain

Abstract. *The MULTI Challenge is intended to encourage the Multilevel Modelling research community to submit solutions to the same, well described problem. This paper presents one solution in the context of process management, where universal properties of process types along with task, artefact and actor types, together with possible particular occurrences for scoped domains, are modelled. We discuss our solution, detailing how we handle each requirement and explain how we use the different features that the MultEcore tool supports to construct the proposed Process case study. We not only focus on the structural dimension of the proposed system where the different models that define the language are provided, but also explore the specification of the static semantics to verify structural constraints and dynamic semantics that refer to the behavioural aspect of the modelled system.*

Keywords. Multilevel Modelling • Semantics • Model Transformations

1 Introduction

Research in Multilevel Modelling (MLM) is continuously increasing and MLM approaches and tools are getting more mature and varied. The MULTI challenge was created to enhance discussion and facilitate the contributions within the MLM community. Encouraging researchers to submit solutions to a common challenge makes it possible to compare them and fosters improvements towards the same set of common goals. In this paper, we use the MultEcore tool (Macías et al. 2016) to create models by applying various multilevel constructions, which are key to fulfil the criteria established for the MULTI Process challenge (João Paulo A. Almeida et al. 2019, 2021). MultEcore enables multilevel modelling through the Eclipse Modelling Framework (EMF) (Steinberg et al. 2008), and therefore allows reusing the existing EMF tools and plugins. The MultEcore tool is available on its webpage[1] and the Eclipse projects which contain all the artefacts of our solution to this challenge can be downloaded from a GitHub repository[2] .

With MultEcore, modellers can create flexible multilevel structures of models that can in turn be composed with each other to include additional aspects. This process is mainly done by defining multilevel hierarchies, where usually the main one is called *application hierarchy* and the additional ones are called *supplementary hierarchies*. Using a parallelism to Software Product Lines, an application hierarchy could be understood as the

* Corresponding author.
E-mail. arte@hvl.no
This work has been partially supported by Comunidad de Madrid as part of the project 49/520608.9/18 (MADRIDF-LIGHTONCHIP) co-funded by ERDF Funds of the European Union.
Note: This work is based on and extends our previous publication to the MULTI 2019 workshop: *Multilevel Modelling with MultEcore: A contribution to the MULTI Process challenge*.

[1] MultEcore website: https://ict.hvl.no/multecore/
[2] GitHub repository with all the artefacts of the MultEcore solution: https://github.com/MultEcore/no.hvl.multecore.examples.emisa.process2021

base language module in the context of a language product line (Méndez-Acuña et al. 2016). Supplementary hierarchies can therefore be used to add new dimensions to the application one, with concepts that are not part of the latter's domain. An application hierarchy can include several supplementary hierarchies which can also be removed without introducing inconsistencies or affecting the integrity of the models.

We also take advantage in this paper of the newest features in MultEcore, some of which are part of our current development efforts. In particular, we discuss the specification of constraints (static semantics) and behaviour (dynamic semantics) of the models by applying MultEcore's model transformation language, which we call Multilevel Coupled Model Transformations (MCMTs) (Macías 2019; Macías et al. 2019; Rodríguez et al. 2019a). The key aspects of our framework which have been applied to solve the challenge are summarised as follows:

- The definition of multilevel hierarchies in a flexible way has allowed us to create tree-like structures where the commonalities of the language are defined once, and the branches can be separately specified and instantiated in a controlled manner by using the notion of *potency*, which restricts the levels at which an element can be used to type other elements. Moreover, we use our three-valued definition of potency, which is able to unify consistently the kinds of potency defined, among others, in Atkinson and Gerbig (2016a) and Lara and Guerra (2010).
- Being able to define supplementary hierarchies helped us with one of the requirements of the challenge (time-stamping nodes).
- The combination of inheritance (i. e. specialisation) and typing (i. e. instantiation) relations, which can be used together consistently in our approach, has been exploited to address certain requirements.
- We benefited from the two main applications of MCMTs to both check the structural correctness of the modelled hierarchies and to specify behavioural descriptions of the bottom-most models.
- An infrastructure that connects MultEcore with the Maude system (Clavel et al. 2007) allowed us to execute our models applying the behavioural MCMT rules.

In this edition, the challenge concerns the domain of process management, which pertains both the particular instantiations of elements (e. g., *process instances*, *task occurrences*), and the universal aspects of the domain (e. g. *process definitions*, *task types*). Note that we use British English throughout the paper, however, we use the original US English of the challenge description for quotations and for the names of our elements in the models. For example the reader may find *artifact* being used instead of *artefact* in some contexts. Respondents to the challenge are required to define, first, universal concepts for process management, and second, an application of such a conceptualisation in the scope of a particular software engineering process. Optionally, they can also capture a different scope for the insurance domain. In order to demonstrate the flexibility of our framework, we have defined a multilevel hierarchy where both domains are included.

The rest of this paper is organised according to the structure recommended on the challenge description, as follows. We describe in Section 2 the technological aspects of MultEcore. In Section 3 we analyse the challenge description and clarify all the assumptions and decisions that we have made in order to fulfil the proposed requirements. We detail in Section 4 all the specific elements which conform the multilevel hierarchies of our solution, presenting both the software engineering and the insurance domains. We also discuss how we handle cross-level constraints and the operational semantics. In Section 5 we discuss each requirement and describe how we addressed it in our solution. In Section 6 we assess the choices we have made and describe how certain asp aspects of our approach facilitate the resolution of the requirements. We discuss in Section 7 related work with respect to other solutions made in past editions of the challenge, both in terms of the approaches used and the solutions developed. Finally, we summarise and conclude the paper in Section 8.

2 Technology

Our solution has been entirely modelled using the MultEcore tool (Macías et al. 2017; Rodríguez and Macías 2019), formally specified in Macías (2019) and Wolter et al. (2019). The MultEcore tool is designed as a set of Eclipse plugins, giving access to its mature tool ecosystem (through integration with EMF) and incorporating the flexibility of MLM. In the MultEcore approach (Macías et al. 2019), the abstract syntax is provided by a set of models that compose the language. The semantics in MultEcore (behaviour and constraints) can be specified by using Multilevel Coupled Model Transformations. Using the MultEcore tool modellers can: (i) define multilevel hierarchies using the model graphical editor; (ii) define MCMTs using the textual DSL for rule edition; and (iii) execute and analyse specific models. The execution of MultEcore models rely on a bidirectional transformation of the models into Maude (Clavel et al. 2007) specifications. When we design a multilevel DSML, we first define its syntax/structure through a multilevel modelling hierarchy and then we specify its semantics via the MCMTs.

2.1 Levels

MultEcore is a level-adjuvant approach (Atkinson et al. 2014; Kühne 2018a) where levels are explicitly used to organise models and the elements inside them. For implementation reasons, MultEcore prescribes the use of Ecore (Steinberg et al. 2008) as root model (graph) at level 0 in all example hierarchies. Models are distributed in *multilevel modelling hierarchies*. A multilevel modelling hierarchy in our context is a tree-shaped arrangement of models with a single root at the top of the hierarchy tree. Levels are indexed with increasing natural numbers starting from the uppermost one, having index 0. All our inter-level relationships between models, nodes and edges are represented via typing relations with the "instance-of" meaning. We use levels as an organisational tool, where the main rationale for locating elements in a particular model is based on how they could potentially define an independent modular artefact. In this regard, we encourage the *level cohesion* principle (Kühne 2018a), that is, we recommend to organise elements that are semantically close (by means of potency and level organisation). On the contrary, we do not promote the *level segregation* principle, which establishes that level organisational semantics should be unique, i. e., aligned to one particular organisational scheme, such as *classification* or *generalisation*. Still, we generally use typing relations with classification semantics, and the typing relation still implies that a node defines which attributes its instances can instantiate and which relations they can have to other nodes. Furthermore, the MultEcore tool checks correct potency and typing safeness. Typing safeness is checked via internal constraints that forbid relations to be circular, reversed or inconsistent neither vertically, i. e., within the same hierarchy, nor horizontally, i. e., if we consider more than one hierarchy.

2.2 Supplementary hierarchies

Frequently, we denote a multilevel hierarchy as the *main* or *default* one and call it *application hierarchy*, since it represents the main language being designed. An application hierarchy can optionally include an arbitrary number of *supplementary hierarchies* which add new aspects to the application one. Adding or removing supplementary hierarchies is made possible by the incorporation or extraction of additional typing chains (see Wolter et al. 2019 for the formal details). For instance, we might have different hierarchies (physically separated, e. g., different projects in the MultEcore tool) that we want to use together. Such a result can be achieved by assigning the role of application hierarchy to one of them and adding the rest as supplementary ones. In this paper, the *Process Hierarchy* acts as application hierarchy and the *Timestamp Hierarchy* is a supplementary one (see Figure 1 and Section 4).

2.3 Instance Characterisation

MultEcore allows for deep characterisation (Atkinson and Kühne 2001) which means that the elements of a model can be instantiated not only in the model immediately below it, but also further down in the hierarchy. It is common in level-adjuvant approaches to use the so-called *potency* mechanism to control the deep instantiation. Potency (Kühne 2018b) is a well-known concept in MLM and it is used on elements as a way of restricting the levels at which this element may be used to type other elements. By using potencies, we can define the degree of flexibility and restrictiveness that we want to allow on the instantiation of the elements of a multilevel hierarchy. Our potency specification is composed by three values on nodes and edges and by two values on attributes. The first two values, *start* and *end*, specify the range of levels below, relative to the current level, where the element can be directly instantiated. The third value, *depth*, is used to control the maximum number of times that the element can be transitively instantiated, or re-instantiated, regardless of the levels where this occurs. Since attributes can be instantiated only once as it does not make sense to create an instance of such instance, the depth on attributes is always 1 and it is not modifiable. Hence, in practical terms, only the first two values (start and end) of the potency are available to the user.

It is worth mentioning that in some parts of this paper, for abbreviation purposes, we use the `X:Y@n` notation to represent that `X` is an instance of `Y`. The optional `@n` represents the `n` number of levels in which `Y` is located above (to which we informally refer as *reverse potency*), with respect to `X`. Note that the default case is @1, which is omitted.

3 Analysis

In this section we discuss our interpretation of the case description, clarifying the assumptions and additions that we have considered to the original description.

First, the challenge description states that '*domain-specific concepts may be defined in their dedicated branches of a hierarchy of models without polluting the general terminology of process management, allowing domain-specific behaviour to be defined for each branch of the hierarchy while allowing for the reuse/enforcement of common structure/behaviour*'. This is precisely the way in which we have organised the required concepts: in a hierarchy with a top model for the generic elements related to processes (Figure 1, top), from which two branches span. The first branch (Figure 1, right) refines these concepts for the (sub)domain of software engineering processes. The second branch (Figure 1, left) does the same for the domain of insurance processes, since we have also included this optional set of concepts in our submission. Based on the suggested refinements (instantiation) of concepts in both branches, the software branch has one more level, since intermediate refinements (e. g. `SEActor`) are required.

Second, the description also requests that '*submitted solutions should include bottom-level instances, at least for key types, exemplifying all attributes mentioned in the challenge description*'. So both of our branches include a bottom-most model which illustrates the instantiation of the nodes, edges and attributes to define a specific state of a process. The result is a five-level, two-branch hierarchy, where each level accounts for a different degree of abstraction in the challenge description. The four user-defined levels (ignoring level 0 with Ecore) are closely aligned with the ones proposed by de Lara et al. in the original process case study (Lara and Guerra 2018, Figure 4).

Third, we assume that when we declare a concept—usually represented with a node—as a meta-concept that will be used to later define actual realisations of it, the former is clearly acting as a type. For instance, the challenge description states the need to define '*actor types*' so that we can define '*actors*'. Hence, we consider that naming the meta-concept `ActorType` is redundant, and therefore we choose to simply name it `Actor`.

Consequently, all the generic elements such as actor type, task type and process type, are named Actor, Task and Process in our solution. It is worth mentioning at this point that we also use a naming convention for relations among nodes, in which verbs are always used in third singular person.

Fourth, we understand that the relation between actors (people) and the duties they can fulfil (roles) is an N:M relationship. That is, one person may have more than one purpose in a process, and one purpose may be shared between several people. If we were to model this situation with actors being connected via a relation to tasks, we would be forced to explicitly model all NxM permutations of people allowed to do tasks, creating too much redundancy. In order to avoid this issue, we created a distinction between actor and role, and created the actor - role - task triangle of concepts, which addresses P5, P9 and P14 (see Section 5). This also allows us to easily apply a composite pattern (Gamma 1995) for combined roles, which are suggested in P15. More importantly, these elements do not affect the general semantics of the models or the alignment with the requirements of the challenge.

Finally, as discussed in Section 5, requirement P19, we chose to create a secondary hierarchy to support the requirement for time stamps in a minimally invasive manner. We also argue that this scenario is a perfect fit to such kind of *aspect orientation* techniques for MLM that are supported in MultEcore.

4 Model presentation

Figure 1 shows the overview of the system architecture we have constructed. We first detail the (main) application hierarchy that captures both domains described in the challenge. This is represented within the dashed central box in the figure, under Process Hierarchy. We describe each level in a subsection and start from level 1 (process).

We also describe how we use the supplementary dimension (dashed box to the right) Timestamp Hierarchy to address one of the requirements of the challenge. Note that supplementary hierarchy is not bound to a specific level but it is orthogonal to the application hierarchy, which means that several models within the Process Hierarchy can make use of the types and attributes defined, in this case, in the unique timestamp model. We further describe this supplementary hierarchy in Section 4.3.3.

We only display the cardinalities on edges in those cases where it is not the default one (0..*). Also, we do not show the Ecore model located at level 0, but start from level 1 as shown in Figure 1.

4.1 Level 1 - Process

The first model in level 1 contains the concepts concerning universal processes (see Figure 2) and corresponds to the process item placed at the top in Figure 1.

A Process contains an arbitrary number of Tasks. As shown in the figure, the type of a node, provided by some element in an upper metamodel, is indicated in a blue ellipse at its top left side, e. g., EClass is the type of Process. Notice the second green ellipse at the right of Process that provides it with a supplementary TimeStamp type. Even

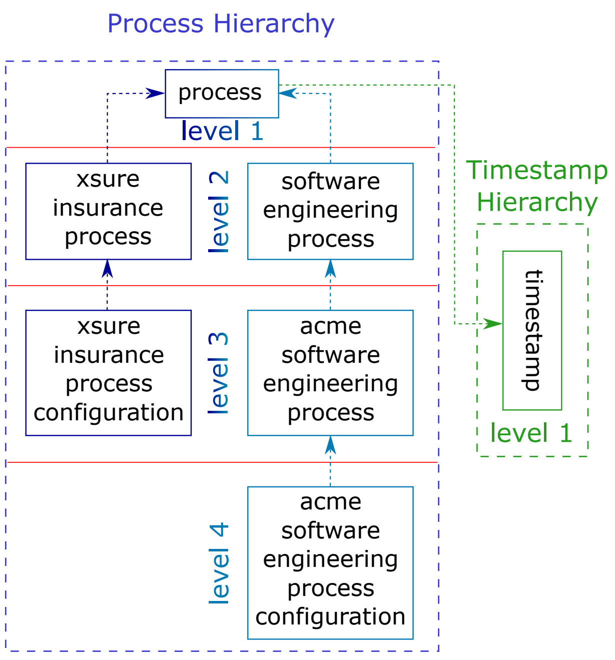

Process Hierarchy
process
level 1
xsure insurance process
level 2
software engineering process
xsure insurance process configuration
level 3
acme software engineering process
level 4
acme software engineering process configuration
Timestamp Hierarchy
timestamp
level 1

Figure 1: High-level system overview

though we further discuss this in Section 5, it is worth mentioning that this second type enables us to instantiate the lastUpdated attribute on any node of the process hierarchy. Note that all the elements within the process hierarchy that are illustrated in this section instantiate the lastUpdated attribute with value 26-Apr-21. Note also that the only elements that have been double-typed are the ones with the second ellipse specified in the process model. The rest of the elements in levels below can still instantiate the attributes since they are ultimately typed by nodes that incorporate the supplementary type at level 1.

The type of an arrow is written near the arrow in italic font type, e. g., EReference for contains. We support attribute declarations that can be typed by one of the four basic Ecore data types, namely Integer, Real, Boolean and String. For instance, Task has declared four attributes, beginDate and endDate of type string, expectedDuration of type int and isCritial as a boolean. Nodes can have at the same time declared and instantiated attributes, as illustrated in Task, that has the four declared attributes commented above, and the lastUpdated instantiated attribute. The annotations displayed as three numbers in a red box at the top right of each node, and concatenated to the name after @ for every edge, specify their potencies. Potency in attributes is displayed as two numbers as an attribute's depth is always 1, since first it is declared, and it can be instantiated only once in a level below. For instance, the potency specified for Task is 1-2-2, which means that an element can be directly instantiated one and two levels below (levels 2 or 3 in the hierarchy), and such instances can be re-instantiated up to 2 additional times. This depth is therefore dependent on the value of the type, and the depth of an element must always be strictly less than the depth of its type.

Each process must have one and only one Initial Task ([1..1] cardinality in initialTask edge) and can have one or more Final Tasks ([1..*] cardinality in finalTask edge). The MultEcore tool allows us to make use of the *inheritance* relation. The inheritance (i. e. specialisation) relation is a special type of arrow among any two nodes within the same level, which imposes on the child node the same typing and potency as the parent node. Moreover, the inheritance relation gives the child node access to the incoming and outgoing arrows of the parent node together with its attributes, while still allowing the child node to define additional attributes or arrows. For instance, InitialTask and FinalTask are children nodes of Task. In MultEcore we can also mark a node, e. g., Gateway, as an abstract node, which cannot be instantiated (indicated by the name in italics). This means that a Gateway must always be instantiated using one of its five children, namely, AndSplit, AndJoin, Sequence, OrJoin or OrSplit (right side of Figure 2). They can connect one or more tasks, depending on the gateway, as indicated in the multiplicity in the source and target relations.

A Task might use and/or produce Artifacts, and such tasks are created and performed by Actors. We define *actor types* as Roles (as discussed in Section 3). The edge hasRole between Actor and AbstractRole models this. We apply to roles the Composite pattern from object-orientation (Gamma 1995). We define AbstractRole as an abstract node. Normal roles are defined as Role and further special roles might inherit from it, for instance, SeniorRole inherits from Role. Furthermore, we use CombinedRole to define roles than can be composed by simple roles (the 2..* cardinality in the includes edge ensures that there are at least two roles combined). Finally, certain roles can perform certain tasks, which is covered by the executes edge from AbstractRole to Task.

4.2 Software engineering process domain

In this section we disclose the domain-specific aspects for the software engineering process which corresponds to the right hand branch of the application hierarchy (see Figure 1).

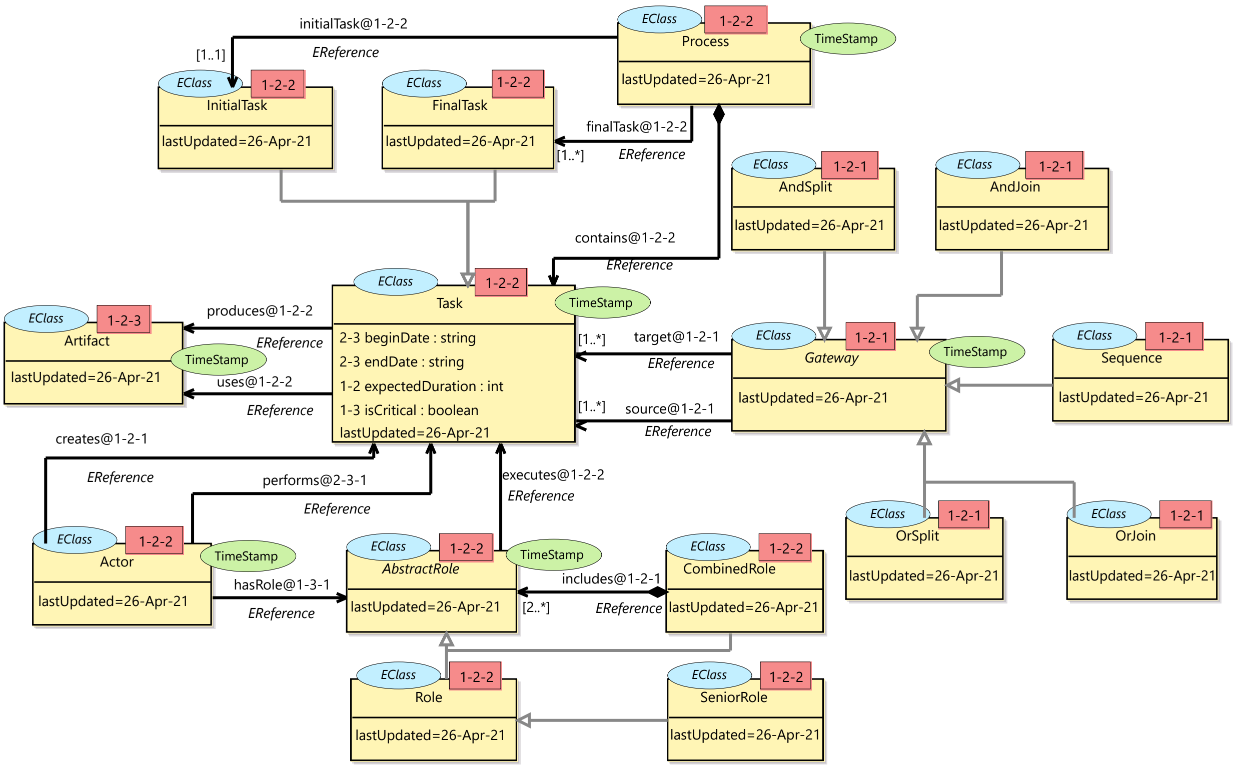


Figure 2: Level 1: Process model

4.2.1 Level 2 - Software engineering process

This level concerns the refinement of concepts from general processes that apply to any software engineering domain. It is represented in Figure 3 and it corresponds to software engineering process in Figure 1.

The creation of this level facilitates the resolution of multiple requirements which are discussed in Section 5. Every software engineering artifact (SEArtifact, which is typed by Artifact (placed at level 1, Figure 2) must have a responsible software engineering actor (responsibleActor relation with multiplicity [1..1] to SEActor). The specification of SEArtifact and SEActor forces the definition of any artifact or actor within the software engineering domain to be typed by SEArtifact or SEActor instead of the generic Artifact or Actor, respectively. Also, each concrete SEArtifact must be assigned a version number (versionNumber attribute). Note the potency 2-2, as the instance level of the software engineering domain is placed at level 4 (acme software engineering process configuration at the bottom of Figure 1).

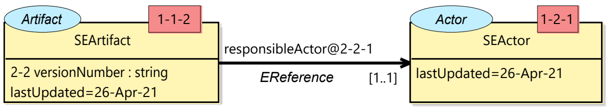


Figure 3: Level 2: Software engineering process model

4.2.2 Level 3 - Acme software engineering process

We now discuss the aspects related to the Acme software engineering process. This model corresponds to the acme software engineering process component in Figure 1. We show in Figure 4 selected parts of the model (right-hand side) in order to compare it with the graphical representation in concrete syntax given in the Challenge description (left-hand side of Figure 4). The complete model that fulfils all the requirements and specifications can be found in Appendix A. From this point, the elements we describe refer to the right-hand side of Figure 4.

At this level, we specify the *types* that belong to the Acme software engineering domain. Note that some of the types of the elements, e. g., InitialTask, Sequence1, RequirementsAnalysis, etc. are allocated two levels above, which is specified, for nodes, in the ellipses where the type is given concatenated with @2, and for the edges, in the

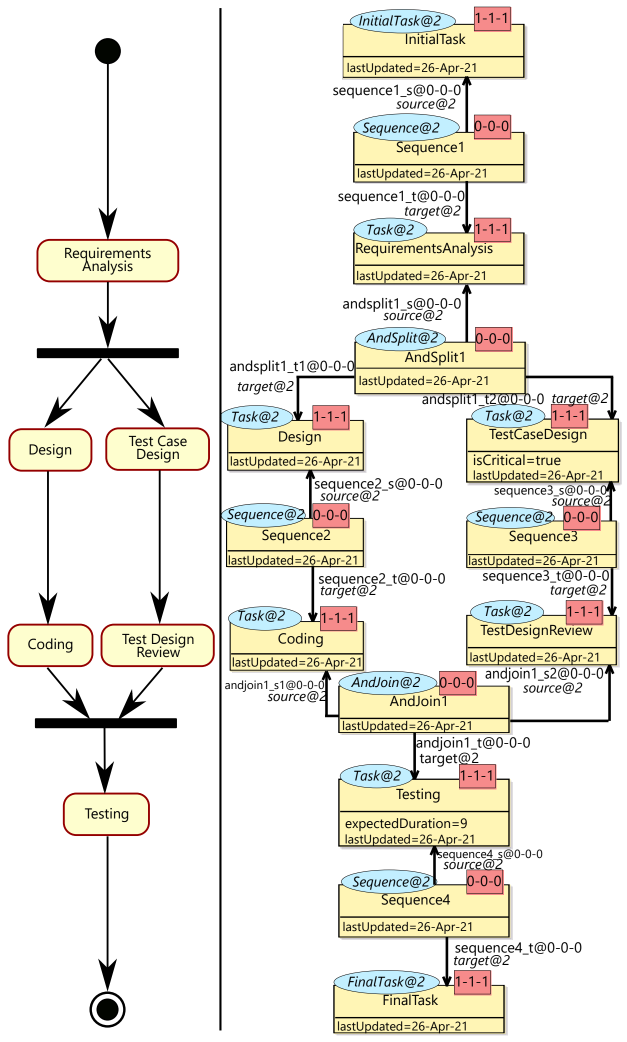


Figure 4: Level 3: Selected parts of the Acme software engineering process model (right-hand side) to compare it with the graphical schema given in the Challenge description João Paulo A. Almeida et al. 2021

types with italic font. One can observe, comparing it with the left-hand side representation, that all the information is accurately reproduced and easy to track.

Note that the potency of some elements, such as Sequence1, AndSplit1, Sequence2, Sequence3, AndJoin1 and Sequence4, is 0-0-0. These values are due to the fact that those elements cannot be further instantiated, which clearly indicates that they belong to this level where the general Acme workflow is represented. Also, notice that we instantiate some attributes here apart from lastUpdated, for example isCritical, which is set to true in TestCaseDesign node, and expectedDuration=9 in Testing node.

4.2.3 Level 4 - Acme software engineering process configuration

In this section we describe the aspects related to a specific application of the concepts defined on the Acme software development process. In this branch (software engineering domain) this model represents a state (i. e. a potential execution) of a process. This model is depicted in Figure 5 and corresponds to the acme software engineering process configuration element in Figure 1. The nodes and relations displayed in this model have been reconstructed using information provided along the software engineering domain requirements (S1, S2, etc.) from the challenge description (João Paulo A. Almeida et al. 2021). Notice that all nodes and relations at this level have as potency 0-0-0, since this is the bottom-most model and cannot be further instantiated. We discuss the elements of this model from left to right on Figure 5.

JohnDoe (typed by SEActor@2) is responsible of the concrete artifacts COBOL (typed by ProgrammingLanguage, with versionNumber=1.3) and COBOLCode. This responsibility is indicated by the incoming cobol_responsibleactor and cobolcode_responsibleactor edges. Also, COBOLCode is written (the type of the edge cobolcode_written) in COBOL. Besides, CodinCOBOL:Coding task uses COBOL and produces

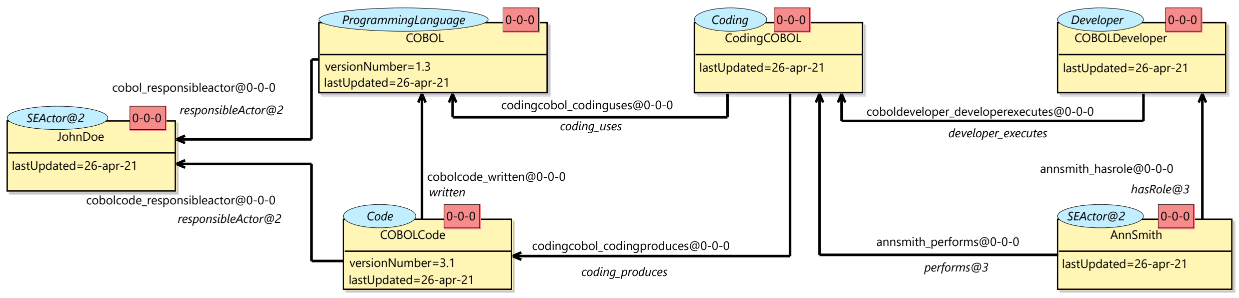


Figure 5: Level 4: Acme software engineering process configuration model

COBOLCode. These two relations are captured by codingcobol_codinguses and codingcobol_codingproduces relations, respectively.

Finally, AnnSmith:SEActor@2 is an actor that has assigned, via the annsmith_hasrole:hasRole@3 relation, the COBOLDeveloper role. AnnSmith performs CodingCOBOL, which the COBOLDeveloper role is allowed to execute. Notice that certain types of the elements aforementioned (e. g., Developer or ProgrammingLanguage) are not explicitly shown in the excerpt on Figure 4, whose full version is detailed in Appendix A.

4.3 Insurance process domain

In the challenge description it is described the so-called *insurance domain*. Even though respondents are encouraged to focus on the software engineering domain, with the insurance part used '*for illustrative purposes only*', we have constructed it in a separate branch and used all the information obtained from analysing the *PX* rules and specifications given in Section 2.2 of the challenge description document. As one can observe in the left-hand side of the Process Hierarchy in Figure 1, the branch is composed by two models (if we ignore process at level 1) rather than three as in the software engineering domain. The demands of this domain do not require the creation of a model that is equivalent to software engineering process model (level 2 in Figure 1). Instead, the level 2 of the insurance branch directly corresponds to the XSure company (xsure insurance process model). This difference demonstrates the flexibility of MultEcore, where the lengths of the different branches of a hierarchy are not required to be equal.

4.3.1 Level 2 - XSure insurance process

The xsure insurance process model, which corresponds to the xsure insurance process element in Figure 1 at level 2, represents the workflow of the ClaimHandling process. For illustrative purposes we show a fragment of the model in Figure 6 and point the reader to Appendix B for the complete model.

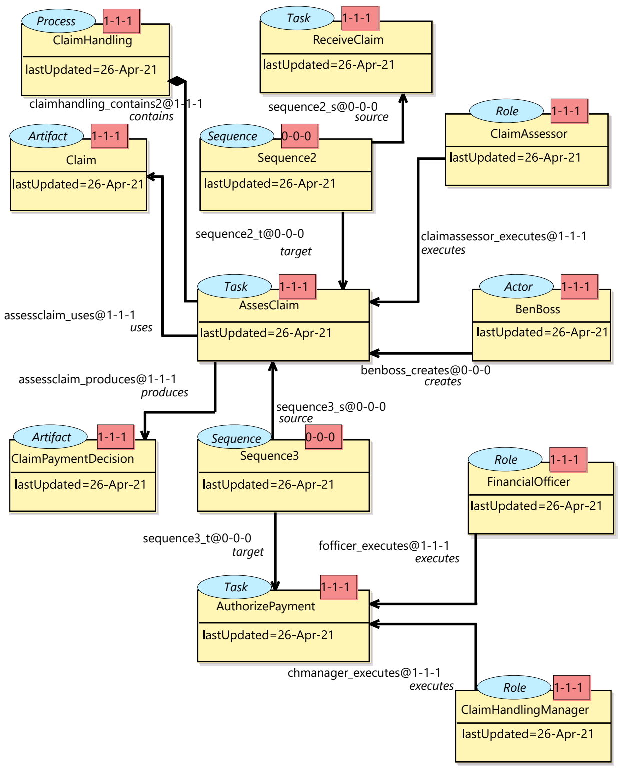


Figure 6: Level 2: XSure insurance process model

We describe the model from top to bottom. A ClaimHandling:Process is composed by all the tasks depicted in the model. To facilitate readability, we only show the containment relation connected to AssesClaim, called claimhandling_contains2 since it is used in the level below that is described in Section 4.3.2. The reader can find all the remaining containment relationships in the appendix in Figure 15. A ReceiveClaim precedes an AssesClaim task, which are connected via Sequence2. To proceed, an AssesClaim uses a Claim:Artifact (to the top left of the figure) and produces a ClaimPaymentDecision. Also, AssesClaim is created (via benboss_creates relation) by BenBoss, who is an Actor.

Furthermore, ClaimAssessor is a Role which is allowed to execute AssesClaim tasks. AssesClaim leads to the following task, AuthorizePayment, connected by the Sequence3 gateway. Finally, both ClaimHandlingManager and FinancialOfficer roles are allowed to execute AuthorizePayment tasks.

4.3.2 Level 3 - XSure insurance process configuration

As stated before, the xsure insurance process configuration model placed at level 3 (see Figure 1) represents the instance level in this particular branch, i. e., represents a concrete scenario and therefore a non-instantiable model (notice the 0-0-0 potencies). The model is depicted in Figure 7.

An instance of the process ClaimHandling, named HandlingClaim123, is defined at this level, and could be interpreted as the concrete implementation of the claim assessment process of the XSure company for a claim with id *123*. It contains, via the handlingclaim123_chcont2 relation, the task AssessingClaim123 that instantiate the two attributes beginDate=01-Jan-19 and endDate=02-Jan-19. Also, XSureAssessor is a ClaimAssessor role that both PaulAlter and JohnSmith actors have assigned. Even though they share that specific role, they have a second role each of them, respectively, XSureManager for PaulAlter and XSureLeader for JohnSmith.

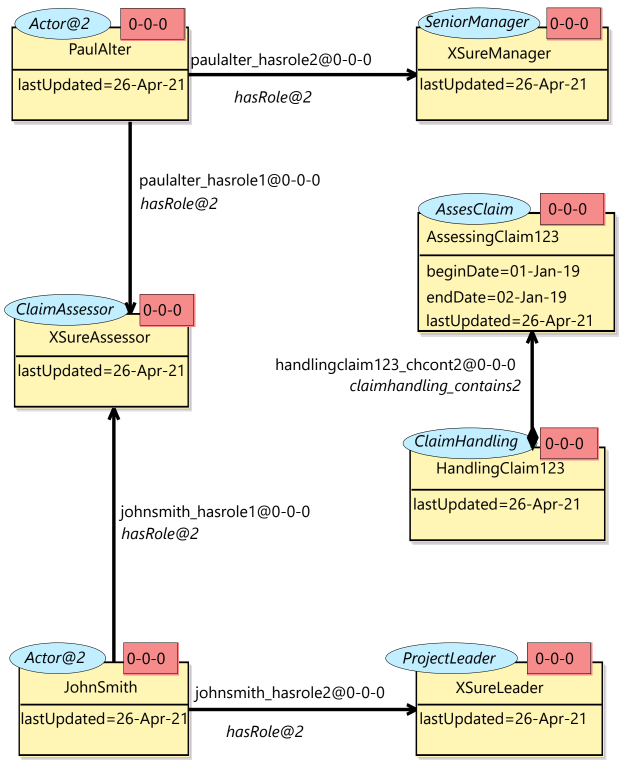


Figure 7: Level 3: XSure insurance process configuration model

This way we display that an actor might have more than one role assigned (as stated by one of the requirements).

4.3.3 Supplementary hierarchies

In this subsection we discuss how we make use of one of the key features that characterises MultEcore. The process hierarchy, which includes both the insurance and the software engineering branches in Figure 1, is the application hierarchy in this case. As mentioned earlier, an application hierarchy can optionally include an arbitrary number of supplementary hierarchies which add new aspects to the application one. The supplementary hierarchy notion has been applied in previous work in different ways: (i) for the runtime verification of properties of an executable workflow (Macías et al. 2018); (ii) to complement a main language with additional non-functional features, for instance, data types (Rodríguez et al. 2018) or additional information to augment the data of a node (Rodríguez and Macías 2019); and (iii) to power up instance elements, where composition of application and supplementary hierarchies could be carried on (Rodríguez et al. 2019c).

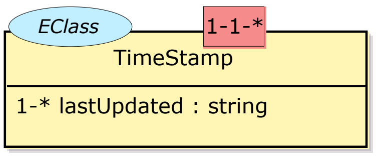


Figure 8: TimeStamp node

As illustrated in Figure 1, we have created a supplementary hierarchy that can provide a *last updated* value to ideally any node defined in the application hierarchy. The supplementary hierarchy, called Timestamp Hierarchy in Figure 1 consists of a single model, called timestamp which has one single node called TimeStamp, as shown in Figure 8. This node has declared the lastUpdated attribute of type string. It is worth reminding that elements from supplementary hierarchies can be used in an orthogonal manner. The advantages of our solution by defining this feature as supplementary is that any node, in any of the models distributed along the Application Hierarchy in both branches can instantiate the instantiate lastUpdated to give it a concrete value (we use the same for all the nodes, lastUpdated=26-Apr-21). The only requirement is to double-type core elements in the process model at level 1, such as Task and Gateway. The result is that any node residing in one of the models depicted in Figures 2, 3, 4 (and its full version in 14), 5, 6 (and its full version in 15) and 7, can instantiate the attribute.

4.4 Cross-level constraints

As introduced in Section 2, MCMTs can be used to specify the dynamic semantics for the definition of behavioural descriptions (as we will see in Section 4.5). In previous work we have shown that this sort of semantics can be executed by using Maude to evolve models with the infrastructure we have built in Rodríguez et al. (2019a). However, the specification and verification of static semantics, i. e., constraints to check some structural correctness of the constructed multilevel hierarchy is explored in this section.

The usual application of the MCMTs when describing behaviour is as endogenous in-place model transformation rules (Mens and Gorp 2006). In this context, the transformation rules represent actions that may happen in the system. These model transformations (MTs) are rule-based modifications of a source model (specified in the left-hand side of the rule) resulting in a new state of such a model (determined by the right-hand side). The left-hand side (LHS) takes as input (a part of) a model and it can be understood as the pattern we want to find in our original model. The right-hand side (RHS) describes the desired modifications that we want to perform in our model and thereby the next state of the system. There is a match when what we specify in the LHS is found in our source model and the execution of the rule represents a single transition in the state space.

These transformations work fine when we want to find a match, and then produce a new state of the model. Still, this mechanism does not completely align with the one we require to define constraints. In order to be able to verify that certain constraints are satisfied we propose a *check mode* that behaves differently than conventional MTs. In this mode, the goal is to find a correspondence in the models through a two-step procedure. Instead of having a model that evolves or change to a new state as it is done for specifying the behaviour (LHS $\rightarrow$ RHS), now, for the model to pass or to be correct with respect to the constraint, both situations (what is being specified in the LHS and the RHS) must be found in the multilevel hierarchy. The fact that the two conditions do not match (or only one of them) results in a constraint violation.

Let us analyse, for instance, the requirement *P17*: '*An actor who performs a task must be authorized for that task. Typically, a class of actors is automatically authorized for certain classes of tasks.*' Figure 9 shows an MCMT rule in check mode to satisfy such a constraint, with a graphical notation instead of the textual one used in the tool to simplify the explanation. The META block allows us to locate types in any level of the hierarchy, and can be used in the FROM and TO blocks (separated by a black horizontal line). It is worth pointing out that the two levels specified (the one for the META and the one for the FROM and TO) in this rule are not required to be consecutive and they would match on levels 1 and 4 of the right-hand branch on Figure 1, respectively. In the case of the insurance domain, this rule would match with levels 1 and 3, respectively, being the rule reusable for both domains.

At the META level, we mirror part of the process metamodel, defining elements like Actor, Task and Role nodes and performs, hasRole and executes edges that are used directly as types in the levels below in the rule. These are constants, which is indicated by underlining the name of the element. A constant in an MCMT rule can only match to an element with the exact name in the corresponding model that has been matched.

For variables (i. e. non-constants), we allow the type on the elements to be indirect, meaning that there can be intermediate types in the actual hierarchy where the MCMT is matched. We see variables in the FROM block, where a first correct match of the rule comes when an element, coupled together with its type, fits an instance of a:Actor that has a relation p:performs to an instance of

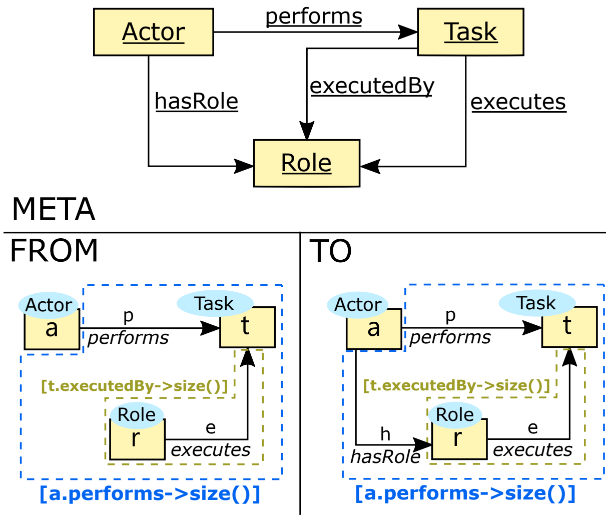


Figure 9: Constraint satisfying requirement P17

t:Task. Also there must exist an instance of r:Role that is linked to t:Task via the e:executes relation. Note that there are two dashed boxes surrounding certain elements. One must take into account that in different scenarios there could an arbitrary number of tasks connected to an actor that can perform them, and that several roles can also be allowed to execute a certain task. To cover all the permutations with a single rule, we use a box-based mechanism that allows us to automatically replicate the contained elements at runtime. Boxes may appear in both sides of the rules, they can be nested, and each of them may have an explicit cardinality specified. Basic support for the Object Constraint Language (OCL) (Clark and Warmer 2003) is incorporated into the MCMTs for: (i) the computation of the cardinality of a box, i. e., the number of times it has to be replicated; (ii) for the manipulation of attribute values (not used in this rule); and (iii) for the specification of conditions (not used in this rule), which greatly improves the expressiveness of the tool. Note that while the support for OCL is very basic, we plan to extend it in future work. For instance, the expression used in the outer box [a.performs->size()] is using the size operation. In OCL, the size() operator calculates the size of the collection it is applied on. The a.performs expression returns the collection of edges whose source is a and its type is performs. Note, however, that the way in which types are used in MLM is a bit different than for standard OCL. This grants transitive typing, which allows for the matching candidate to have any type that is either performs itself or (in intermediate levels) other elements which are ultimately typed by performs. In practical terms, the [a.performs->size()] means that the size of the collection is given by the number of tasks connected to the matched actor (a) via edges of type performs. Similarly, the inner box that encapsulates r:Role and e:executes with the [t.executedBy->size()] expression would count the number of edges (which types ultimately match executedBy) the element t has. Note that executedBy relation is not defined in the process model (Figure 2) since this rule is yet theoretical.

Once all the boxes have been unfolded for the FROM part of the rule, and there has been a match of all the (unfolded) variables, this partial matching is saved and reused in the TO block, to check whether its contents can also be matched. In the case of the TO block, for each task that an actor is performing, and given that a certain role can execute such a task, there must exist a relation h of type hasRole between the actor and some of the roles that are allowed to execute the task. The two consecutive and successful matches would verify that the multilevel hierarchy satisfies the constraint.

4.5 Operational semantics

The challenge description does not comment on the possibility of describing the operational semantics of processes, which can also be done quite naturally through model transformations. We find this fact surprising, given that MT is one of the pillars of Model-Driven Software Engineering in general, and has been already tackled by several authors within the MLM community apart from ourselves (Atkinson et al. 2012, 2009; Kühne and Schreiber 2007; Lara and Guerra 2010; Lara et al. 2015; Rossini et al. 2014). We believe that exploring the possibilities of MT for this challenge could enrich the submissions, debate within the community and further editions of the multilevel challenge. Moreover, the process domain of this challenge is a good candidate for specifying operational semantics through MT rules, since the concept of processes being executed already implies some sort of evolution through time of the models that represent them. So, in this subsection we focus on a simplified proposal of MT rules to model the way in which gateways are triggered to create the next tasks of a process once the previous ones are completed.

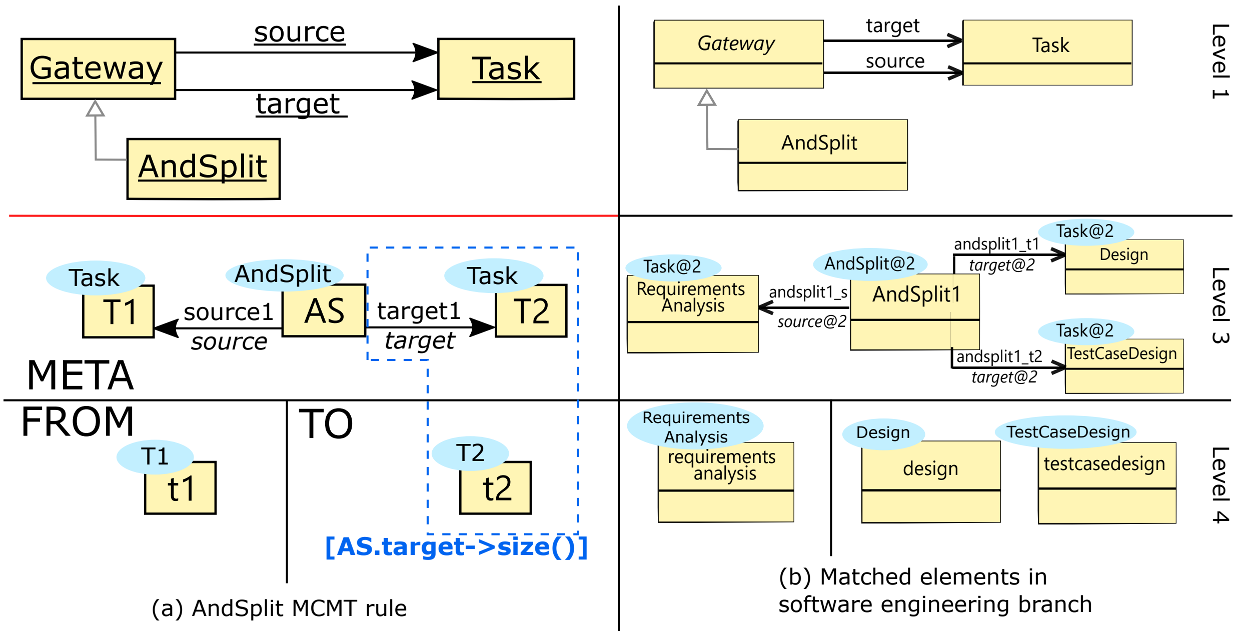
For a more realistic and comprehensive collection of rules that could fully *animate* the models, a new version of the challenge would be required where MTs are taken into account to create a more complete and unambiguous description of the operational semantics of the elements on the domain.

We have explained how MCMTs could be used to specify cross-level constraints that check the structural correctness of the multilevel hierarchy (Section 4.4). Now we describe how MCMTs can be used to specify the behavioural descriptions of the modelled system by means of model transformations. MCMTs have been widely improved since their initial proposal in Macías et al. (2019). While MCMTs are powerful enough to describe many behavioural aspects, it is necessary to have an engine that can execute them against a multilevel hierarchy to have an actual execution mechanism capable of evolving the models. To do so, we rely on the Maude System (Clavel et al. 2007; Durán et al. 2020), a specification language based on rewriting logic (Meseguer 1992), which can naturally deal with states and non-deterministic concurrent computations. A preliminary version of the infrastructure we have implemented (still under development) was presented in Rodríguez et al. (2019b). In that version, the multilevel hierarchy and the set of MCMT rules was transformed into a functional Maude representation that could be executed using the Maude console environment. The results had to be brought back manually one by one, which had some practical and usable limitations. Also, the MCMTs expressive power was rather limited in that version compared to the capabilities they offer nowadays.

Since the goal of this paper is to demonstrate how MultEcore can be used to model the proposed challenge, we do not enter into the Maude specification details. Still, all the produced Maude files can be found in our GitHub repository. The current state of the infrastructure that connects MultEcore with Maude relies on a bidirectional transformation that takes the entire MultEcore representation (both the multilevel hierarchy and the associated MCMT rules) and automatically generates Maude specifications. Then, this transformer takes the XML output files that Maude produces as result of performing execution, and automatically translates them back into MultEcore models that are graphically displayed. The Maude part is handled by a background process which makes the underlying Maude transformations transparent to the user.

To show the potential of the MCMTs we provide now an example of how they can be used to systematically create parts of the models based on the information allocated within the process hierarchy. Therefore, as an illustrative example and to open this line for future Multilevel Modelling challenges, we have sketched five simple MCMT rules that involve the creation of new tasks at the bottom-most levels (level 3 for insurance and level 4 for software engineering) through the information of corresponding gateways connected to the tasks on the levels above. This set of rules handles several cases regarding the different gateways and the initial and final tasks. In the following, we describe one of the rules, but the remaining ones can be found in their textual form with the rest of the artefacts in our solution to the challenge on GitHub.

We show in Figure 10(a) an MCMT rule to create several output tasks from an input task where their types are connected via an *and-split* gateway. Similarly to the rule shown in Figure 9 in Section 4.4, we define at the topmost level of the MCMT constant elements such as AndSplit that inherits from Gateway that is connected to Task via source and target relations. The second META level defines variable elements, such as T1 and T2 of type Task and AS of type AndSplit that connects with the former two via source1 and target1, respectively. These will be matched with elements in level 3 of the software engineering branch, and with elements in level 2 for the insurance branch. In the FROM block, we identify a single t1 task whose type (T1) would be the input of the corresponding and-split gateway. Then, in the TO block, we remove the matched t1 and create the new t2 tasks which types are the outputs of the and-split. Note that each particular process can establish an arbitrary number of output tasks for different instantiations of AndSplit. To make a generic rule that works for any number of output tasks, we define boxes around target1 and T2 from the META block and around t2 from the TO block. Note that the possibility to establish cross-level boxes is a new feature which we have introduced in MultEcore



(a) AndSplit MCMT rule

(b) Matched elements in software engineering branch

Figure 10: (a) MCMT rule to create new tasks based in their types connected via and-join gateways. (b) Matched elements in the Software engineering branch

to be able to handle the current case. The OCL expression [AS.target->size()] counts how many target tasks are connected to the matched and-split gateway. Note that the cross-level box is needed because the gateway information is not given at the instance level, but a level above. Therefore, the three elements must come together into the same box, so when it is unfolded at runtime, the type of each produced t2 is paired correctly with the information located in the level above.

Figure 10(b) shows the corresponding matched elements in the software engineering branch. We can observe in this example the vertical flexibility of the MCMT rules, since the matched elements are distributed within level 1, level 3 and level 4 for the three levels specified in the rule. Even though there exists an intermediate level in the software engineering branch (level 2), it is not a problem for the rule to ignore it and match the appropriate elements in the correct models. At the bottom of Figure 10(b) we see, divided by a vertical line, the two parts of the model that would match the FROM and the TO blocks. In this case, requirementsanalysis would match t1 and design and testcasedesign the two replicated t2 variables.

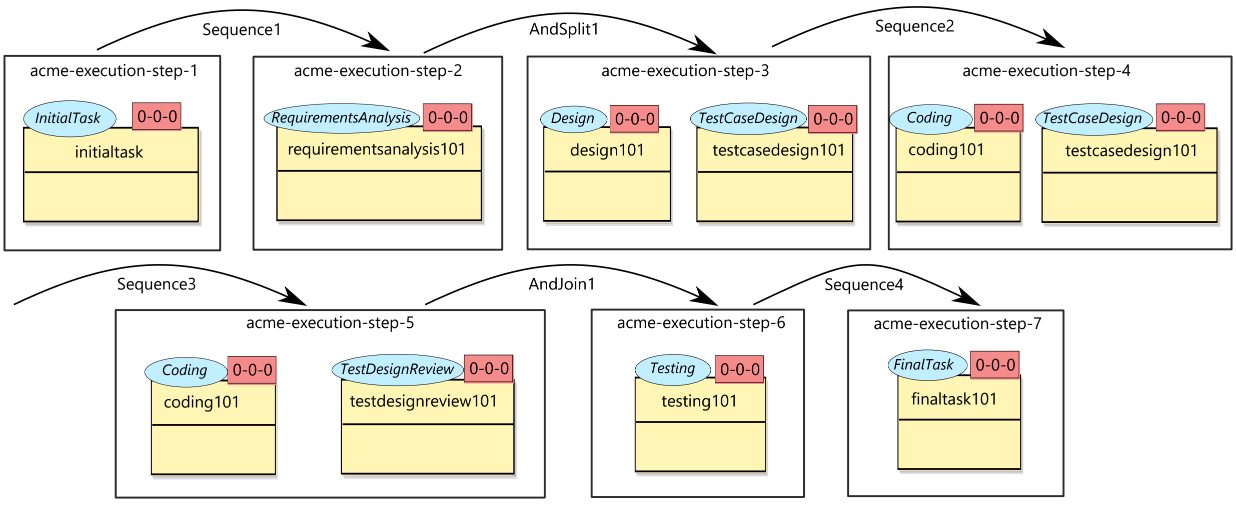


Figure 11: Acme software engineering obtained states by applying subsequent MCMT rules

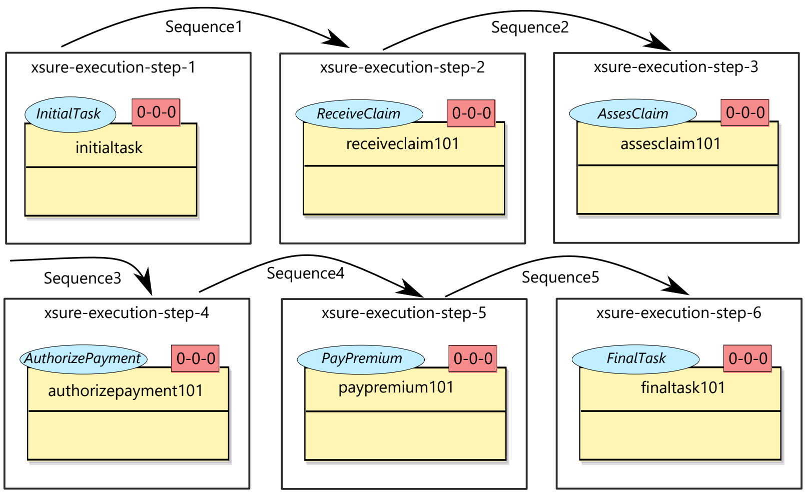


Figure 12: Xsure insurance claim handling process model states obtained by applying subsequent MCMT rules

The Maude integration within MultEcore allows us to use all the available tools for Maude. For execution, we allow the modeller to specify a number of steps to be executed (being each step the application of one of the available rules) or directly customise the execution by stating which rules and in which order should they be applied. To demonstrate the application of different rules, we have created two basic instance models, one for the insurance domain and one for the software engineering domain, each of them with a single element named `initialTask`. The five MCMT rules specified allow us to reach a `finalTask` based on each of the workflows defined in Figures 14 and 15. Note that the executions are only concerned about tasks and gateways, and do not consider other elements such as artefacts or actors.

Figure 11 shows the models for each of the seven execution steps that have been obtained by applying each corresponding MCMT rule on the software engineering domain. At the top left, we have the initial model `acme-execution-step-1` with the `initialTask` node. To its right, obtained by applying the rule that triggers the `Sequence1` gateway we have the `acme-execution-step2` with the `requirmenetsanalysis101` node. Note that the number appended to the name is generated using a *Counter* object that is used in the Maude representation, whose value gets increased every time a new identifier is created. This counter allows us to create fresh elements avoiding name duplication. The name below each curved arrow between model states (instances) does not represent the name of the executed MCMT rule, but the gateway that is matched in the level above where the workflow is represented (see Appendix A). One can observe at the bottom right of Figure 11 how we finally reach an instance `finaltask101` representing the end of the workflow.

Likewise, and demonstrating the horizontal flexibility of the MCMTs, Figure 12 represents six model states produced by the execution engine by applying the same set of rules to the insurance domain. Similarly, the workflow established in the corresponding level above (the XSure insurance claim handling process) is defined in Appendix B. The process is quite similar as for software engineering, where there exists an initial task (top left of Figure 12) in the initial model `xsure-execution-step-1`. Then, by applying rule by rule we get new elements in new model instances, such as `receivclaim101`, `assesclaim101`, . . . and finally `finaltask101`.

5 Satisfaction of Requirements

In this section, we explain how our solution addresses all the requirements in the challenge description. First, we discuss the ones related to the more abstract concepts of processes, tasks, actors and artefacts (João Paulo A. Almeida et al. 2021, Section 2.2.). We preserve their original name format (PX, with X being a number) for easy traceability and reproduce their text for self-containment.

P1

‘*A process type (such as claim handling) is defined by the composition of one or more task types (receive claim, assess claim, pay premium) and their relations.*’

This requirement is addressed with the definition of the nodes `Process` and `Task`, and the containment relationship from the former to the latter. They are contained in model `process` at the top of the hierarchy (see Figures 1 and 2). The suggested instances (*claim handling*, *receive claim*, etc.) have been also used to create the optional insurance (sub)domain in the corresponding branch of the hierarchy, as presented in Section 4.3.

P2

‘*Ordering constraints between task types of a process type are established through gateways, which may be sequencing, and-split, or-split, and-join and or-join.*’

We understand from the way the requirement is written that the set of gateways is fixed and not likely to change. Also, we consider that, semantically speaking, they belong to the same level of abstraction than task, process, etc. This decision is reinforced by the fact that the same gateways are common to all processes. Hence, we choose to define `Gateway` as an abstract node in the process model, and include the four kinds of gateways as children of it—i. e. related via inheritance, exploiting the fact that MultEcore also allows this kind of relation, as explained in Section 4.1. The rationale behind declaring Gateway as abstract is that all processes must use one of its children types for defining sequencing of tasks, but it does not make sense to create an instance of the parent. Finally, it should be noted that, while inheritance is a less flexible construction than typing, adding new kinds of gateways (e. g. xor-split and xor-join) could still be achieved by adding them as new children of Gateway. We refer the reader to Section 4.5 for a discussion on how the operational semantics of these gateways could be easily specified with Multilevel Coupled Model Transformations.

P3

‘*A process type has one initial task type (with which all its executions begin), and one or more final task types (with which all its executions end).*’

This requirement is also addressed using inheritance relations in the process model, following a similar rationale as in the previous one. Therefore, we include the nodes InitialTask and FinalTask, and define specialised containment relations from node Process into them, instead of reusing the one for intermediate tasks. More importantly, these two relations define different cardinalities to enforce a unique initial and at least one final task per process, as per the requirement. We do not define inheritance relations among these different containment relations since that kind of construction is not supported in MultEcore.

P4

‘*Each task type is created by an actor, who will not necessarily perform it. For example, Ben Boss created the task type assess claim.*’

This requirement entails in our solution the definition of the Actor node and the creates relation from it into Task. The other relation performs hinted in this requirement is discussed in the following one. The example instances mentioned in the requirement are also used in the lower model of the insurance branch of the hierarchy (see Section 4.3).

P5

‘*For each task type, one may stipulate a set of actor types whose instances are the only ones that may perform instances of that task type. For example, in the XSure insurance company, only a claim handling manager or a financial officer may authorize payments.*’

First, we include another relation from Actor to Task, called performs. However, this is not enough to model which types of actor can execute which types of tasks. As pointed out before in Section 4.1, we split the concept of actor as an actual person (e. g. *Ben Boss*) and as a specific role that a person may play (e. g. *claim handling manager*) to allow for the flexibility of several people being able to play the same role, and also for the same person to perform more than one role. Therefore, apart from the Actor node discussed in the previous requirement, we create the different Role nodes, some of which appear as a response to following requirements. For the purpose of fulfilling this requirement, the way we model the semantics that an actor is allowed to perform a task, is by checking that it has a role which can execute that task. Therefore, we create the aforementioned nodes, plus the hasRole and executes relations, so that the semantics are encoded in the Actor – Role – Task triangle.

P6

‘*A task type may alternatively be assigned to a particular set of actors who are authorized (e. g., John Smith and Paul Alter may be the only actors who are allowed to assess claims).*’

A naive way to address this requirement could consist of the creation of yet another relation (called assigned or something similar) between Actor and Task. But since we define the Role nodes to fulfil other requirements, we can simply take advantage of the *triangle* mentioned in the previous one, and create a role that is assigned to both actors. In such a way, we cover this requirement without needing to define any new elements.

We argue that, besides being a flexible construction, this way of modelling the requirement makes sense from a semantic point of view: there should be some common ability, permission or status that makes those people suitable to perform the task, and we allow modelling it explicitly. Again, the examples used in the requirement are created as instances on the insurance branch, and we also attach plausible roles to those actors to complete the model.

P7

'*For each task type (such as authorize payment) one may stipulate the artifact types which are used and produced. For example, assess claim uses a claim and produces a claim payment decision.*'

This requirement is tackled by simply defining the Artifact node and the uses and produces relations. Again, the examples mentioned in the requirement are used to construct the optional insurance branch in our solution.

P8

'*Task types have an expected duration (which is not necessarily respected in particular occurrences).*'

We just need to add the expectedDuration attribute of type Integer to the Task node to complete this requirement.

P9

'*Critical task types are those whose instances are critical tasks; each of the latter must be performed by a senior actor and the artifacts they produce must be associated with a validation task.*'

Once again, we can take advantage of the separation of actors and roles to avoid creating a child node of Task for critical tasks. Instead, we add a simple Boolean attribute isCritical to Task, so that a similar constraint to the one defined in Section 4.4 can be used to ensure that critical tasks are only performed by senior actors. But since in our solution the information about what an actor can do is not stored in Actor itself but in Role, we create a child node of the latter called SeniorRole. That is, the actor which performs a task marked as critical, must have at least one senior role, which must be able to execute such task, as indicated by the executes relation. In this case we do not use an attribute to distinguish roles from senior roles since we latter use combined roles through a composite pattern, which is more easily illustrated using inheritance relations.

Regarding the fact that '*the artifacts they produce must be associated with a validation task*', we include an instance of the uses relation which connects TestDesignReview to TestCase. We think that creating an specific child node of Task in process for validation tasks is not a good alternative in this case, as it would pollute that model with software-specific concepts—other kinds of processes may not have validation tasks.

P10

'*Each process type may be enacted multiple times.*'

This requirement is trivially addressed in our solution since we allow for multiple instantiations of a process, as our hierarchy shows.

P11

'*Each process comprises one or more tasks.*'

The contains relation specified for P1 from Process to Task, and the way they are instantiated, already covers this requirement.

P12

'*Each task has a begin date and an end date. (e.g., Assessing Claim 123 has begin date 01-Jan-19 and end date 02-Jan-19).*'

Both attributes have been declared in Task, and are instantiated in the corresponding node, at the lower model of the insurance branch in our hierarchy.

P13

'*Tasks are associated with artifacts used and produced, along with performing actors.*'

This requirement is addressed by creating two new relations in the process model: uses and produces from Task to Artifact. The performs relation that we discuss in earlier requirements is also used to satisfy this one.

P14

'*Every artifact used or produced in a task must instantiate one of the artifact types stipulated for the task type.*'

Thanks to the way we model the structure of this elements in the process model, this requirement is trivially solved by instantiating Artifact, Task and the uses and produces relations between them appropriately. We show how this can be achieved in the example instance models at the bottom of both branches in our hierarchy.

P15
‘*An actor may have more than one actor type (e.g., Senior Manager and Project Leader.)*’

Thanks to the separation of actors and their roles in our solution, this requirement can be easily addressed. The hasRole relation has a 0..* cardinality, so an actor can have several roles by default. But in order to improve the reusability of roles, we choose to include the concept of CombinedRole, which realises the composite pattern (Gamma 1995). In such a way, a combination of roles that several actors share can be defined just once as an instance of CombinedRole and related to several actors. Apart from the aforementioned advantages, using this construction we also remove the need for multiple typing, which is a controversial topic in MLM. Moreover, using multiple typing within the same hierarchy in MultEcore is not allowed. Using MultEcore’s supplementary hierarchies (see Section 2.2) as a means to add additional types did not make sense in this context in any case, since there are no differentiated domains that justify such a construction.

P16
‘*Likewise, an artifact may have more than one artifact type.*’

Same as we did for roles, we can use a composite pattern to address this requirement. In such a way, instead of using multiple typing (which, as argued before, is neither desirable nor a possible alternative in MultEcore), we could join simple artifacts into combined artifacts, and use the latter as replacements of multiple typing. We choose not to include this construction in our solution to keep the process model as simple as possible. Besides, none of the other requirements entails defining an instance of Artifact with multiple types.

P17
‘*An actor who performs a task must be authorized for that task. Typically, a class of actors is automatically authorized for certain classes of tasks.*’

Once again, the triangle construction of actors, tasks and roles allows us to fulfil this requirement without adding any new elements to the models, but defining a constraint over them. First, we explicitly represent in our models that an actor performs a task and also has a series of roles, some of which are allowed to execute certain types of tasks. Then, when instances of Role and Task are created in lower levels, it can be checked that they instantiate the corresponding relations in order to verify this requirement, using the constraint from Figure 9.

P18
‘*Actor types may specialize other actor types in which case all the rules that apply to instances of the specialized actor type must apply to instances of the specializing actor type. For example, if a manager is allowed to perform tasks of a certain task type, so is a senior manager.*’

The nature of inheritance in MultEcore allows us to easily model this requirement. Since we use distinction between actors and roles, this requirement actually affects the latter in our solution, according to our understanding: a role can specialise another role, but it does not make sense for an actual person to inherit from another in this context. That is, an actor can have a role, and a separate actor a second role which inherits from the first. In such a way, the specialising role (child) would inherit its executes relation to a task from the specialised role (parent), and any actor having the specialising role would be allowed to perform that task too. However, since this requirement also mentions the possibility of “senior” versions of the different roles (initially mentioned in P9), we also include a specialisation of Role into SeniorRole, which can be directly instantiated in order to recognise those specialised roles involving seniority. The consequence here is that an X:Role can be specialised into a

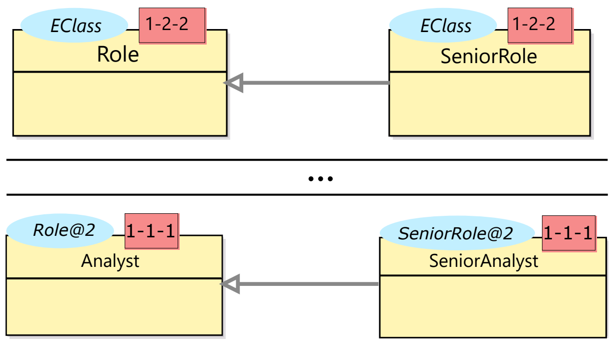


Figure 13: Fragment of the process multilevel hierarchy showing the P18 requirement fulfilment

Y:SeniorRole, which is allowed by MultEcore. In general, a node can inherit from another as long as their potencies match and their types are the same, or alternatively the specialising node's type is itself a specialisation of the specialised node's type. An excerpt of our models illustrating this scenario is shown in Figure 13. Finally, the constraint that enforces that only actors with the right role might execute a particular task also ensures that this requirement is fulfilled (Figure 9).

P19

'*All modeling elements, at all levels, must have a last updated value of type time stamp. This feature should be defined as few times as possible, ideally only once. Respective definitions are exempt from the requirement to have a last updated value.*'

The key to fulfilling this requirement is defining a lastUpdated attribute with the loosest potency possible (1-*) in such a way that it can be instantiated no matter how the hierarchy grows—be it in depth, in width or in number of elements in a model—without forcing the modeller to add neither more definitions of that same attribute, nor typing or inheritance relations to previously defined elements, nor any other mechanism that entails accidental complexity (Atkinson and Kühne 2008). With this goal in mind, we considered four alternative constructions that were possible in our solution. First, we could naively add a copy of the attribute to every node without a parent in the process model. But this solution would forbid the actual elements in that model from instantiating the attribute, so it is not a perfectly valid option. Second, we could add an extra model on top of process (displacing all models in the hierarchy one level lower) for the definition of a single node TimeStamp which contains the aforementioned attribute definition, and type every node in the process model by it. This alternative would give us the desired effect, but does not make any sense semantically. Besides, it is an ad-hoc solution which could cause trouble if we eventually need that level on top for other purposes. The last two options are based on the use of our supplementary typing mechanism to separate concerns, since we can consider time-stamping an aspect that could be included in many domains without being an integral part of any of them. In such a away, we can define a supplementary hierarchy with a single model (two, if we count Ecore on level 0), which contains the TimeStamp node with the lastUpdated attribute with 1-1 potency and 1..1 multiplicity, since the requirement states that nodes *must* instantiate it. So, the third option we considered consisted of adding this TimeStamp as a supplementary type to every other node in the application hierarchy. However, this option is far from ideal, since every new node that we add to the hierarchy needs to be double-typed with this supplementary type to be able to instantiate the attribute. So, finally, the fourth option which we actually implement in our solution is an improvement of the previous one: we change the attribute potency to 1-* and only add TimeStamp as a supplementary to Process, Task, Gateway, AbstractRole, Actor and Artifact; since the rest of the nodes in model process are children of one of them. With this construction, we only define the attribute once, we "link" it six times (through supplementary typing) and it is already available everywhere thanks to inheritance (in the process model) and potency (in the rest of the hierarchy). Moreover, it would still be available in any new branches, any new models in the existing branches and any new node that we define in the existing models using the types of process. It would only be required to add TimeStamp as a supplementary type by hand if we were to instantiate

EClass. So we believe that our solution contains a nearly-optimal solution for this requirement.

To sum up the discussion of the PX requirements, Table 1 summarises whether they have been tackled in our solution and how.

Secondly in this section, we discuss the requirements which are specific for software engineering processes (Section 2.3. in the challenge description). We begin by reproducing in MultEcore the diagram shown in Figure 1 in the description, both of which are included in our Figure 4 for a side-by-side comparison[3] . Using this figure as a starting point, we add the different nodes and edges that we require to fulfil those requirements. The full model is depicted in the appendix, in Figure 14. Again, we refer to them with their original names of the form SX, and summarise the following discussion at the end of this section, in Table 2.

S1

‘*A requirements analysis is performed by an analyst and produces a requirements specification.*’

RequirementsAnalysis is present in the description’s figure and is therefore included already in the initial version of the model in Figure 4. To that same model—acme software engineering process, which we refer to as just acme process in the remainder—we add the nodes Analyst:Role and RequirementsSpecification:SEArtifact, plus the corresponding relations, according to the process model (Figure 2). The usage of SEArtifact instead of Artifact as type in the latter node is due to requirement S10, and we refer the reader to that discussion for a justification of this choice. This same remark also applies to some of the following requirements.

S2

‘*A test case design is performed only by senior analysts and produces test cases.*’

We have simplified the wording of this requirement while maintaining its meaning. To fulfil it, we include SeniorAnalyst:SeniorRole and the corresponding instance of the executes relation in model acme process. We indicate that SeniorAnalyst is an specialised version of Analyst through an inheritance relation, as an example of the construction discussed in P18. We also include TestCase:SEArtifact and instantiate the produces relation to indicate that it is a product of test case design. Note that the senior analyst role does not need to be connected to an actor for this model to be correct. Hence, we choose not to overload the models with additional details beyond the ones enforced by the requirements, in order to simplify their description and visualisation in this paper.

S3

‘*An occurrence of coding is performed by a developer and produces code. It must furthermore reference one or more programming languages employed.*’

To address this requirement we add to acme process the nodes Developer:Role, and two instances of SEArtifact: Code and ProgrammingLanguage. We connect these nodes to the coding task by instantiating, respectively, the relations executes, produces and uses.

S4

‘*Code must reference the programming language(s) in which it was written.*’

In order to represent that a code is written in a programming language, we create a relation written between these two artifacts. Since this relation only pertains two software-specific artifacts, it does not have a type in the process model. For such scenarios, MultEcore always allows to create direct instances of an EClass or EReference through potency, and in this case we use the latter as the type of written. Although this construction differs conceptually from linguistic extensions (Atkinson and Kühne 2001), its practical usage is quite similar to it.

S5 and S6

‘*Coding in COBOL always produces COBOL code.*’ ‘*All COBOL code is written in COBOL.*’

We group together these two requirements since they pertain the same part of the model and have common elements. Coding in COBOL, as an

[3] Note that each instance of Sequence is depicted as a node, plus the two arrows which indicate its source and target tasks.

Table 1: Summary of PX requirements

Req.	Addressed?	Comments
P1	+	—
P2	+	Using inheritance
P3	+	Using inheritance
P4	+	—
P5	+	Modelled as a triangle between the nodes Actor, Role and Task
P6	+	Does not require new modelling constructs
P7	+	—
P8	+	—
P9	+	Using a constraint, and creation of ValidationTask dismissed
P10	+	Trivially addressed
P11	+	Addressed in P1
P12	+	—
P13	+	Partially addressed earlier
P14	+	Trivially addressed
P15	+	Trivially addressed, but improved with composite pattern
P16	+	Using composite pattern, but not actually modelled
P17	+	Using a constraint
P18	+	Using inheritance and an existing constraint
P19	+	Using a supplementary hierarchy

instance of the Coding task, belongs naturally in a level below the model acme process, since the latter deals with coding as a generic concept, as the original figure in the challenge description shows. Therefore, the new node CodingCOBOL:Coding is declared in the bottom-most model of the software branch of our hierarchy: acme software engineering process configuration, which we call acme configuration for short and is depicted in Figure 5. This same reasoning can be applied to COBOLCode:Code and COBOL:ProgrammingLanguage. To complete the model, the relevant instances of the relations coding_uses, coding_produces and written are used to connect those three nodes to each other, modelling the semantics of both requirements.

S7

'*Ann Smith is a developer; she is the only one allowed to perform coding in COBOL.*'

The fulfilment of this requirement implies creating an instance of Actor in the acme configuration model. Due to the refinement of Actor (from process model) into SEActor (from software engineering process model, called software process for short) that S10 entails, the node AnnSmith that we create is an instance of SEActor. Note that our implementation allows us to create instances of SEActor in both levels 3 and 4 of the software branch of the hierarchy. Hence, we include AnnSmith:SEActor@2 in the bottom model, and instantiate the performs relation from process (which relates actors to tasks) to indicate that she carries out the task coding in COBOL. As already explained, our solution distinguishes between actors (as actual people) and the roles they perform, so we also create a special type of developer that is allowed to code in COBOL, i. e. COBOLDeveloper:Developer. Ann Smith is connected to this role via an instantiation of the hasRole relation. Finally, even though there is already an instantiation of executes between Developer and Coding in the acme process model, we think that it is appropriate to instantiate it again in this model, between COBOLDeveloper and

CodingCOBOL. We believe that this repetition provides clarity to the model and simplifies the definition of the constraint presented in 4.4.

S8

‘*Testing is performed by a tester and produces a test report.*’

The node Testing is already present, so we add the nodes Tester:Role@2 and TestReport:SEArtifact to the model acme process. We instantiate the relations executes and produces (from two levels above) in order to connect each node to Testing, respectively.

S9

‘*Each tested artifact must be associated to its test report.*’

At first glance, it could be argued that this requirement can be satisfied in the model software process in level 2 of the software branch. However, we believe that it actually pertains to model acme process, since it is only related to Testing, which is declared on that model. Moreover, Testing may not be defined or used in the same way in different software processes which could be defined in other hypothetical software companies. With this choice, we also avoid the need for a constraint that would check that Testing is associated with only some specific instances of SEArtifact. Therefore, we include in model acme process a node TestReport:SEArtifact that is connected to the existing nodes Testing and Code via two relations isTested:EReference and testing_produces:produces@2, respectively. As explained in S4, we exploit the fact that MultEcore allows creating direct instances of EReference anywhere for the typing of isTested. We choose to only create isTested for Code, but there is no obstacle if one wants to create more relations like it from other instances of SEArtifact to other test reports—or even the same one, if one wanted to model a test report that contains info about several tested artefacts.

S10

‘*Software engineering artifacts have a responsible actor and a version number. This applies to requirements specification, code, test case, test report, but also to any future types of software engineering artifacts.*’

We hinted in the discussion of previous requirements that this one entails, to our understanding, the creation of the intermediate model software process for the software branch, that does not have a counterpart in the insurance branch. In this new model, we need to refine generic artefacts into software engineering artefacts which contain more information. Hence, in the model in level 2 we create Artifact:SEArtifact, which defines the required attribute versionNumber of type String that should be instantiated in the bottom-most level of the branch—i. e. model acme configuration in level 4. The potency that we require for the attribute is therefore 2-2 (recall that the depth for attributes is always 1 and consequently not displayed). The cardinality of this attribute, not displayed, is 1..1, so that the attribute *must* be instantiated, according to the requirement. In such a way, any X:Y:SEArtifact in model acme configuration needs to instantiate the attribute, as COBOL and COBOLCode illustrate. To fulfil the rest of the requirement, we also need to model that instances of SEArtifact have a special relation to actors. Since MultEcore does not allow for cross-level relations, we need to create a corresponding SEActor:Actor in software process so that we can then define responsibleActor:EReference among them. Using the same rationale as for the attribute, the potency of responsibleActor is 2-2-1 and its cardinality is 1..1. Examples of instances of this relation are those connecting COBOL and COBOLCode to JohnDoe in model acme configuration.

S11

‘*Bob Brown is an analyst and tester. He has created all task types in this software development process.*’

We interpret that ‘*this software development process*’ refers to a specific instance of a software process. That is, the model in Figure 1 in the challenge description, which corresponds to model acme process in our solution. Hence, we include in that model a node BobBrown:SEActor and instantiate the creates relation from it towards every instance of Task in this model, e. g. Design. This includes the initial and final tasks that every process must have. It is worth pointing out again that our solution allows for the creation of direct instances of actors in two different levels, both in the insurance branch (as instances of Actor) and in the software branch (instantiating SEActor). This construction is necessary since the two lower levels in both branches of our hierarchy (levels 2 and 3 on insurance branch; 3 and 4 in software) may need to define actors in order to adhere to the requirements, e. g. Bob Brown needs to appear in model acme process and Ann Smith in acme configuration. In contrast, roles can be simply instantiated and re-instantiated in those levels, since the domain naturally requires so, e. g. COBOLDeveloper:Developer:Role@2. While this construction for actors might seem undesirable at first, we argue that it removes the need for cross-level relations and that it neither requires any additional elements to be defined nor enforces an artificial re-instantiation of actors—which would be done in a similar manner as we do for roles. The only shortcoming that we see in our solution is that the same actor may appear twice in two models in adjacent levels. For example, if Ann Smith would be responsible for creating tasks that appear in acme process, she would also have to appear there along Bob Brown, and hence would be a duplicate of the Ann Smith that is already present in acme configuration. However, if some practical application of our models—like code generation—were affected by such duplication, we could simply identify both nodes based on the fact that they share the same name, type and potency.

S12

'*The expected duration of testing is 9 days.*'

Testing in model acme process instantiates the expectedDuration integer attribute to 9 to fulfil this requirement.

S13

'*Designing test cases is a critical task which must be performed by a senior analyst. Test cases must be validated by a test design review.*'

We instantiate the isCritical Boolean attribute to true in TestCaseDesign, in model acme process. We connect the node SeniorAnalyst to that instance of Task with an instance of executes. For the sake of simplicity, we do not relate this role to any actor, although it would be reasonable to do so eventually. The fact that a test design review validates the test case design is already represented in the original workflow as a sequence of the two tasks.

6 Assessment of the Modeling Solution

In this section, we discuss the advantages and shortcomings of the choices we made in our solution to the challenge. We also point out whether we were forced to make any compromises or whether our solution presents any deficiencies.

6.1 Basic modelling constructs

MultEcore is graph-based from a theoretical point of view, and this fact reflects on the EMF-based implementation. All models use nodes and relations as the basic building blocks, which are contained in models. Attributes are formally nodes, as explained in Macías (2019), but in practise they behave as commonly expected: they are defined inside a node and instantiated in the instances of that node. The rationale for the separation of these elements in different models is to make them as independent from each other as possible, so that they can be connected to each other only by typing relations. This eases the addition and removal of intermediate models. For example, software process could be removed from our hierarchy, and the types of the elements in the models below just be replaced by the type of the removed types, e. g. SEActor to Actor and responsibleActor to EReference. To achieve this separation, potency plays an important role, as discussed later in this section. Furthermore, combining inheritance with typing also allows us to choose whichever

Table 2: Summary of SX requirements

Req.	Addressed?	Comments
S1	+	—
S2	+	—
S3	+	—
S4	+	Creating a direct instance of EReference through potency
S5	+	Addressed in new model acme configuration
S6	+	Addressed in new model acme configuration
S7	+	Using roles
S8	+	—
S9	+	Addressed in acme process, not in software process
S10	+	Addressed in new model software process
S11	+	May entail actor duplication
S12	+	—
S13	+	—

construction is more flexible, understandable and aligned with the requirements, e. g. the composite pattern that we present for roles.

6.2 Levels

Levels are used as an organisational tool in MultEcore, as explained in Section 2.1. This rationale entails that the typing relations—from nodes to nodes and from relations to relations—have the meaning of *my type defines my structure*, in the sense of which relations can be defined and to which other nodes, which attributes can be instantiated, which nodes can inherit from which other nodes, etc. Due to potency, these typing relations can jump over levels, but still levels serve as a default organisation of models and the elements they contain. We also mentioned already in Section 2.1 that these typing relations among levels do not necessarily adhere to classification with all its implications, since we prioritise flexibility and conciseness, but is in general quite aligned with the concept.

6.3 Number of levels

As stated before, hierarchies in MultEcore are unbounded, so the hierarchy we present could grow downwards as much as necessary. We chose to add an intermediate level for refinements related to software processes (e. g. SEActor), which could perhaps have been done with inheritance in model process. However, this alternative would pollute the model, which is supposed to be generic and unaffected by the particularities of any subdomain. Conversely, we did not force a similar intermediate level in the insurance branch just to keep the hierarchy symmetric since it was not necessary, but of course it could be included if required at a later point in time. To sum up, we designed our solution to be as flexible as possible, and used levels to create, to our understanding, clearly-defined partitions of the domain: processes in general, software processes, the software process of a particular company, and the state of such process at a specific point in time (and a similar partition for the insurance subdomain).

Actually, any of the models in the intermediate levels can be considered a DSML which is used to define the level(s) below it, using the types they define in a structurally coherent manner and satisfying the given constraints. The bottom-most models represent a specific state of the process, e. g. *Ann Smith, who is a COBOL Developer, is using COBOL version 1.3 to implement version 3.1 of a particular piece of COBOL code*. These bottom-most models could be used for different purposes, like logging the different tasks performed by the actors and the generated artefacts, or for monitoring purposes, by representing the current state of the process. If the models were enhanced with further details, one could even consider the execution of simulations prior to the actual enactment of the process in the real world. In such a way, it would be possible to asses whether the specified process, task distribution, workload, etc. are likely to succeed or will probably lead to time and budget overruns.

6.4 Cross-level relationships

Cross-level relations go against modularity, and therefore would disfavour some benefits of our approach, e. g. flexibility and reusability. Moreover, they are purposely not supported by our current formalisation (Wolter et al. 2019). Hence, our solution does not employ them, but we have not identified any case in which they are more desirable than an alternative construction.

6.5 Cross-level constraints

The expressive power of MCMTs allows us to use them to define different kinds of semantics, which have been illustrated in this paper. We can specify dynamic semantics to describe the behavioural aspect of the modelled system and also define static semantics that check the structural correctness of the multilevel hierarchy. As discussed in Section 4.5, the dynamic semantics are applied following the traditional in-place model transformations rules manner where the match of the left-hand side of the rule leads to the modifications specified on the right-hand side. The cross-level constraints would be executed in a so-called *check mode* where the left-hand side and the right-hand side specify two multilevel sub-hierarchy patterns that have to be found for the constraint to be satisfied (see Section 4.4).

6.6 Integrity mechanisms

This discussion is twofold: integrity mechanisms which prevent incorrect constructions and repairing mechanisms if such a construction is made. For the first group, both the formalisation and the implementation of MultEcore has mechanisms to avoid cyclic inheritance, cyclic typing, potency-violating typing, invalid inheritance, multiplicity violations for relations and attributes, duplication of elements and incorrect typing relations for all kinds of elements. Repairing actions like the ones required for the co-evolution of models, metamodels and MTs are not part of MultEcore. However, the tool does include some basic repairing mechanisms, e. g. fixing the potency of an element to `0-0-0` if any of the three values becomes `0` or correcting depth of an element to the depth of its type minus one, if a higher or equal value is specified. Additionally, more advanced repair mechanisms are planned in future releases, such as changing the type of an element to the type of its type if the former is removed.

6.7 Deep characterisation

Our solution makes intensive use of potency, with no element using MultEcore's default potency of `1-1-*`. The reasons for this are two. First, the presented scenario clearly defines a bottom-most level for model instances (i. e. enactments) of specific processes, and it does not make sense to create further instances of such models. Hence, the value of depth is always bounded. And second, the way in which elements most in the top models, especially `process`, are expected to be used, forces us to use end values higher than `1`. In some cases, even the start value differs from `1` to prevent them from being instantiated in the level below, e. g. the `performs` relation in `process`.

6.8 Generality

We believe that our solution performs very well regarding its generality, and the reusability that it entails. We have managed to create a solution with minimal redundancy in most cases, the only exceptions being the potential duplication of actor in two adjacent levels and the several supplementary typing relations in `process` to enable the instantiation of the attribute `lastUpdated` in all nodes. Moreover, we illustrated the reusability of the `process` model and the related MCMT rules by modelling the optional insurance domain, and including an example execution of this process in our solution. In general, we believe that the software process model can be used for other software-related companies that may implement different processes than Acme. Likewise, the acme process model one level below could both be instantiated for other points in time of the same enactment (as illustrated by our MCMT-based execution) or enacted differently for other departments of the same company that adhere to the same process.

6.9 Extensibility

Our solution already illustrates how some extensions can be performed when new requirements appear. For example, the discussions regarding senior actors and validation tasks in P9. Similar extensions through inheritance are always available and simple to perform, since they only add new information to the models, therefore not compromising their integrity or semantics. Moreover, we have shown how model (or level) insertion can be performed by introducing an intermediate level in the software branch (motivated by S11) which has no counterpart in the insurance branch of our hierarchy.

We finalise this section by discussing two of the topics that are recommended in the challenge description.

First, we would like to remark that MultEcore is not only the supporting tool for our approach that we have used to fully create the models and MTs presented in this challenge. The approach also includes a detailed formalisation based on Graph Theory and Category Theory which provides a framework of reference for the tool's behaviour (Wolter et al. 2019).

And second, although we already stated that the verification of our models can be performed through integrity mechanisms (Section 6.6), there are additional checks. For example, the standard validators of EMF for Ecore models and their XMI instances can be still used with MultEcore, since the tool reflects multilevel changes on both facets for the models in each level, using a mechanism called *sliding window* presented in Macías (2019, Section 4.1). This validator can be used to check that obligatory attributes are correctly instantiated or that the multiplicities of relations are respected, among others. However, these checks have some caveats due to the way in which multilevel aspects are represented in those models, so a full integration that does not display multilevel constructions as errors is still a matter of future work.

7 Related Work

The MULTI challenge has received several responses by the community in order to bring insights on how MLM can be applied to solve the suggested scenarios. We first discuss other solutions related to the Process Challenge in 2019 (João Paulo A. Almeida et al. 2019).

Jeusfeld's solution (see Jeusfeld 2019b) is implemented in DeepTelos (Jeusfeld 2019a; Jeusfeld and Neumayr 2016) that extends Telos and that allows to define hierarchies of level objects (called *most-general* instances). DeepTelos is developed by just creating the DeepTelos objects with additional rules/constraints in ConceptBase (Jarke et al. 1995). Note that the core idea of DeepTelos is to exploit the powertype pattern (Odell 1994) and therefore is a level-blind approach (Henderson-Sellers et al. 2013), which means that it does not express an explicit notion of level, even though they are intuitively derived by analysing the solution implementation. This powertype-based solution allows them to naturally deal with cross-level relationships, feature that we do not support in MultEcore. On the other hand, Jeusfeld argue that certain requirements, such as P17 can no be completely fulfilled as they would have to extend their specification by Telos rules. Conversely, our multilevel transformation language (MCMTs) allows us to specify multilevel constraints. The most-general instances idea replaces the well-known potency mechanism present in level-adjuvant approaches. In concrete, our three-value potency specification allows us to be both generic and precise depending on the particular needs. Such level of precision remains a bit blurry to us regarding the solution in Jeusfeld (2019b), where they also have to make the explicit separation between Task and TaskType, which we deem unnecessary in a multilevel context.

Somogyi et al. (Somogyi et al. 2019) also contributed with their solution by using their tool DMLA (Theisz et al. 2019; Urbán et al. 2018). DMLA is a self-validating metamodelling formalism relying on gradual model constraining through its interpretation of the classical instantiation relation. DMLA is self-described, and it also provides so-called *fluid metamodelling*, which means that it is not required to instantiate all entities of a model at once. Models in DMLA are stored in tuples, referencing each other, and thus, forming an entity graph. It is also a level-blind approach that naturally supports the specification of cross-level relationships. Being a level-blind approach where all entities can reference any other entity (the fluid nature), it is easier for the modeller to construct invalid models, which is more difficult in other approaches where the hierarchy of models is clearly constructed, like ours. Furthermore, the sanity checks that potency gives facilitates the modeller to always be sure that the model under construction is correct. Also, DMLA does not explicitly supports some features, such as inheritance (even though authors argue that it can be simulated). While MultEcore naturally supports inheritance, Somogyi et al. had to simulate inheritance which resulted in an artificial workaround to solve some of the requirements.

The two solutions discussed above were the only ones published along with our MultEcore response in 2019. Even though in 2018 there was another challenge case, namely the Bicycle Challenge[4] , we find interesting to discuss the work presented by Lange and Atkinson (Lange and Atkinson 2018). Note that Mezei et al. (Mezei et al. 2018) also presented a solution using DMLA, for which we do not enter into more details as the relevant aspects have already been discussed in the previous paragraph.

[4] Bicycle Challenge 2018: https://www.wi-inf.uni-duisburg-essen.de/MULTI2018/wp-content/uploads/2018/03/MULTI2018-BicycleChallenge.pdf

Lange and Atkinson's solution (Lange and Atkinson 2018) was constructed using the mature tool Melanee (Atkinson and Gerbig 2016b). Melanee is one of the most advanced tools based in OCA (Atkinson and Kühne 2005) for deep modelling which supports modelling through deep, multi-format, multi-notation, user-defined languages. The Melanee solution is closer to what our solution with MultEcore looks like as it is a level-adjuvant approach that also distribute models according to the ontological classification of its elements and uses (a different form of) potency. Like in MultEcore, Melanee does not allow cross-level relationships so models are organised into clear abstraction levels. While this has some advantages, it also has some drawbacks, for instance, the creation of additional nodes in certain levels to make the connections. An example reflected in our solution is the fact that an actor may appear in two different abstraction levels. If we take as a reference the right branch of the multilevel hierarchy depicted in Figure 1, while an actor can create tasks in level 3 of this branch (see, for example, BobBrown on the right side of Figure 14) it can also perform concrete tasks such as CodingCOBOL, performed by AnnSmith (see bottom right of Figure 5).

Finally, regarding our own submission to the challenge in 2019 (see Rodríguez and Macías 2019) we have made improvements and extensions both to the solution and to the MultEcore tool. The multilevel hierarchy presented in the previous work was modelled so it was symmetric, i. e., both branches (insurance and software engineering) had the same length. This forced us to include an intermediate model for the insurance domain that did not really capture any of the requirements stated in the challenge description. In the current version presented in this article this model has been avoided, which helped us demonstrate a flexibility aspect of MultEcore where the different domains do not need to have the same length as they are fully independent from each other. Moreover, as demonstrated, the MCMT rules are still applicable to both domains due to their vertical and horizontal flexibility (for more details on this, we refer the reader to Rodríguez et al. 2019b, Section 4.2). The composite pattern was already implemented for roles in the solution submitted in 2019. MultEcore's facilities such as the use of inheritance and the potency customisation capabilities allowed us to exploit such a pattern within the multilevel context. Thus, we have also used this construction to discuss how to model the artefact situation (P16), and the scenario on Figure 13. Furthermore, we explore in this paper the operational semantics of process challenge. We have shown in Section 4.5 how we can execute models and evolve them by applying MCMT rules that describe the behaviour. This part was not examined in our submission in Rodríguez and Macías (2019). We have also included some minor enhancements which we previously overlooked, like the values of some potencies or making the node `Gateway` abstract. Finally, we have improved the way in which supplementary attributes can be instantiated to develop a nearly-optimal solution for requirement P19.

8 Conclusions and Future work

In this paper, we have presented an extended solution based on our initial contribution (Rodríguez and Macías 2019) to the Process Challenge proposed at the MULTI workshop (João Paulo A. Almeida et al. 2021). Our multilevel modelling hierarchy has a total of five abstraction levels, two branches and 7 models (more if we take into account all the model states generated during the execution, shown in Section 4.5). Such hierarchical distribution covers the generic domain of process description and its refinement for the software engineering and the insurance domains. Each level can be understood as a potential candidate for the generation of software artefacts, like domain-specific editors (graphical and/or textual) to specify processes at any level of abstraction, or for the simulation of processes through model transformations at the bottom levels. Our solution is based on the MultEcore tool and the infrastructure that connects its to Maude which allows us to perform simulation/execution. MultEcore is built on top of EMF which allows us to use all the EMF capabilities boosted with multilevel capabilities. For instance, this facilitates the usage of the rich ecosystem of EMF such as using editors with Sirius for graphical results and Xtext for custom specification languages.

From a more conceptual standpoint, one of our ambitions with respect to MultEcore is to make it an approach that enhances flexibility and reusability. This has allowed us to create an elegant, concise and correct multilevel hierarchy for the given domain of process modelling where, for example, the branches are independent between them and their lengths are different. We believe that this solution can be an interesting contribution for the challenge and be used to foster fruitful discussions within the MLM community. Furthermore, we have gone one step ahead by exploring behavioural aspects, and we believe that including this dimension as part of future challenge proposals would bring engaging results from the MLM community.

We have presented preliminary results regarding execution by showing some examples of model evolution by applying operational semantics via MCMT rules. Currently we are actively working on MultEcore-Maude infrastructure to improve the execution and further verification of the specified multilevel hierarchies. Also, we are studying how to improve the MCMTs flexibility, by taking advantage of inheritance to reuse some MCMT rules with common behaviour. While MCMTs are flexible with respect to horizontal and vertical extensions, we identify a key point of improvement as being able to reuse `META` levels on MCMTs into other rules. Another important aspect that we plan to work on, is the implementation in Maude and the integration into MultEcore of the check mode of MCMTs for the validation of the multilevel hierarchy with respect to structural constraints (as shown in Section 4.4).

We conclude this paper by answering the questions that the challenge description explicitly asks respondents to address.

'*Does the submission address the established domain as described in Section 2 and demonstrate* *the use of multi-level features?*' We believe that our solution contains all the required concepts and constructions required in the challenge description. In most cases, these constructions do not require workarounds or additional concepts, and we discuss and justify our choices in the few cases where we need them. Furthermore, our solution prominently makes use of multiple levels, three-valued potency specification and double typing (through a supplementary hierarchy). All of these concepts are important multilevel features that this submission showcases.

'*Does it evaluate/discuss the proposed modeling solution against the criteria presented in Section 3?*' The whole Section 6 in this paper is dedicated precisely to the discussion of those criteria, in the same order that they are enumerated in João Paulo A. Almeida et al. 2019, so that we can make sure that this question is properly addressed. We also included the recommended discussion aspects suggested by the challenge description.

'*Does it discuss the merits and limitations of the applied MLM technique in the context of the challenge? Authors may suggest further requirements that clearly demonstrate the utility of their chosen approach.*' We have thoroughly discussed the advantages of MultEcore and the few scenarios where we found limitations all throughout Sections 2, 5, 6 and 7. We have also suggested including new requirements regarding the operational semantics of the challenge's domain for upcoming editions in Section 4.5.

References

Almeida J. P. A., Rutle A., Wimmer M., Kühne T. (2019) The MULTI Process Challenge. In: Burgueño L., Pretschner A., Voss S., Chaudron M., Kienzle J., Völter M., Gérard S., Zahedi M., Bousse E., Rensink A., Polack F., Engels G., Kappel G. (eds.) 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019. IEEE, pp. 164–167

Almeida J. P. A., Rutle A., Wimmer M., Kühne T. (2021) The MULTI Process Challenge. In: Enterprise Modelling and Information Systems Architectures Available at https://bit.ly/3b3cQZV

Atkinson C., Gerbig R. (1st Jan. 2016a) Flexible Deep Modeling with Melanee. In: Betz S., Reimer U. (eds.) Modellierung 2016. LNI Vol. 255. Gesellschaft für Informatik, Bonn, pp. 117–122

Atkinson C., Gerbig R. (1st Jan. 2016b) Flexible Deep Modeling with Melanee. In: Betz S., Reimer U. (eds.) Modellierung 2016. LNI Vol. 255. Gesellschaft für Informatik, Bonn, pp. 117–122

Atkinson C., Gerbig R., Kühne T. (2014) Comparing multi-level modeling approaches. In: Atkinson C., Grossmann G., Kühne T., de Lara J. (eds.) Proceedings of the Workshop on Multi-Level Modelling co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2014), Valencia, Spain, September 28, 2014. CEUR Workshop Proceedings Vol. 1286. CEUR-WS.org, pp. 53–61

Atkinson C., Gerbig R., Tunjic C. (2012) Towards Multi-level Aware Model Transformations. In: Hu Z., de Lara J. (eds.) Theory and Practice of Model Transformations - 5th Intl. Conf., ICMT 2012. LNCS Vol. 7307. Springer, pp. 208–223

Atkinson C., Gutheil M., Kennel B. (2009) A Flexible Infrastructure for Multilevel Language Engineering. In: IEEE Trans. Software Eng. 35(6), pp. 742–755

Atkinson C., Kühne T. (2001) The Essence of Multilevel Metamodeling. In: Gogolla M., Kobryn C. (eds.) «UML» 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings. Lecture Notes in Computer Science Vol. 2185. Springer, pp. 19–33

Atkinson C., Kühne T. (2005) Concepts for Comparing Modeling Tool Architectures. In: Briand L. C., Williams C. (eds.) Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005, Proceedings. Lecture

Notes in Computer Science Vol. 3713. Springer, pp. 398–413

Atkinson C., Kühne T. (2008) Reducing accidental complexity in domain models. In: Software & Systems Modeling 7(3), pp. 345–359

Clark T., Warmer J. (2003) Object Modeling With the OCL: The Rationale Behind the Object Constraint Language Vol. 2263. Springer

Clavel M., Durán F., Eker S., Lincoln P., Martí-Oliet N., Meseguer J., Talcott C. L. (eds.) All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic. Lecture Notes in Computer Science Vol. 4350. Springer

Durán F., Eker S., Escobar S., Martí-Oliet N., Meseguer J., Rubio R., Talcott C. L. (2020) Programming and symbolic computation in Maude. In: J. Log. Algebraic Methods Program. 110 https://doi.org/10.1016/j.jlamp.2019.100497

Gamma E. (1995) Design patterns: elements of reusable object-oriented software. Pearson Education India

Henderson-Sellers B., Clark T., Gonzalez-Perez C. (2013) On the Search for a Level-Agnostic Modelling Language. In: Salinesi C., Norrie M. C., Pastor O. (eds.) Advanced Information Systems Engineering - 25th International Conference, CAiSE 2013, Valencia, Spain, June 17-21, 2013. Proceedings. Lecture Notes in Computer Science Vol. 7908. Springer, pp. 240–255

Jarke M., Gallersdörfer R., Jeusfeld M. A., Staudt M. (1995) ConceptBase - A Deductive Object Base for Meta Data Management. In: J. Intell. Inf. Syst. 4(2), pp. 167–192

Jeusfeld M. A. (2019a) DeepTelos Demonstration. In: 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019. IEEE, pp. 98–102

Jeusfeld M. A. (2019b) DeepTelos for ConceptBase: A Contribution to the MULTI Process Challenge. In: 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019. IEEE, pp. 66–77

Jeusfeld M. A., Neumayr B. (2016) DeepTelos: Multi-level Modeling with Most General Instances. In: Comyn-Wattiau I., Tanaka K., Song I.-Y., Yamamoto S., Saeki M. (eds.) Conceptual Modeling - 35th International Conference, ER 2016, Gifu, Japan, November 14-17, 2016, Proceedings. Lecture Notes in Computer Science Vol. 9974, pp. 198–211

Kühne T. (2018a) A story of levels. In: Proceedings of MODELS 2018 Workshops: ModComp, MRT, OCL, FlexMDE, EXE, COMMitMDE, MDETools, GEMOC, MORSE, MDE4IoT, MDEbug, MoDeVVa, ME, MULTI, HuFaMo, AMMoRe, PAINS co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), Copenhagen, Denmark, October, 14, 2018., pp. 673–682 http://ceur-ws.org/Vol-2245/multi%5C_paper%5C_5.pdf

Kühne T. (2018b) Exploring Potency. In: Wasowski A., Paige R. F., Haugen Ø. (eds.) Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018, Copenhagen, Denmark, October 14-19, 2018. ACM, pp. 2–12

Kühne T., Schreiber D. (2007) Can programming be liberated from the two-level style: multi-level programming with DeepJava. In: ACM SIGPLAN Notices 42(10), pp. 229–244

Lange A., Atkinson C. (2018) Multi-level modeling with MELANEE. In: Hebig R., Berger T. (eds.) Proceedings of MODELS 2018 Workshops, Copenhagen, Denmark, October, 14, 2018. CEUR Workshop Proceedings Vol. 2245. CEUR-WS.org, pp. 653–662 de Lara J., Guerra E. (July 2010) Deep metamodelling with MetaDepth. In: Objects, Models, Components, Patterns. LNCS Vol. 6141. Springer, pp. 1–20

de Lara J., Guerra E. (2018) Refactoring Multi-Level Models. In: ACM Transactions on Software Engineering and Methodology (TOSEM) 27(4), p. 17

de Lara J., Guerra E., Sánchez Cuadrado J. (2015) Model-driven engineering with domain-specific meta-modelling languages. In: Software & Systems Modeling 14(1), pp. 429–459

Macías F. (2019) Multilevel modelling and domain-specific languages. PhD thesis, Western Norway University of Applied Sciences and University of Oslo

Macías F., Rutle A., Stolz V. (2016) MultEcore: Combining the Best of Fixed-Level and Multilevel Metamodelling.. In: MULTI@ MoDELS, pp. 66–75

Macías F., Rutle A., Stolz V. (2017) Multilevel Modelling with MultEcore: A Contribution to the MULTI 2017 Challenge. In: Proceedings of MULTI @ MODELS, pp. 269–273 http://ceur-ws.org/Vol-2019/multi%5C_9.pdf

Macías F., Rutle A., Stolz V., Rodríguez-Echeverría R., Wolter U. (2018) An Approach to Flexible Multilevel Modelling. In: Enterprise Modelling and Information Systems Architectures 13, 10:1–10:35

Macías F., Wolter U., Rutle A., Durán F., Rodriguez-Echeverria R. (2019) Multilevel Coupled Model Transformations for Precise and Reusable Definition of Model Behaviour. In: Journal of Logical and Algebraic Methods in Programming

Méndez-Acuña D., Galindo J. A., Degueule T., Combemale B., Baudry B. (2016) Leveraging Software Product Lines Engineering in the development of external DSLs: A systematic literature review. In: Computer Languages, Systems & Structures 46, pp. 206–235

Mens T., Gorp P. V. (2006) A Taxonomy of Model Transformation. In: Electron. Notes Theor. Comput. Sci. 152, pp. 125–142

Meseguer J. (1992) Conditioned Rewriting Logic as a United Model of Concurrency. In: Theor. Comput. Sci. 96(1), pp. 73–155

Mezei G., Theisz Z., Urbán D., Bácsi S. (2018) The bicycle challenge in DMLA, where validation means correct modeling. In: Hebig R., Berger T. (eds.) Proceedings of MODELS 2018 Workshops, Copenhagen, Denmark, October, 14, 2018. CEUR Workshop Proceedings Vol. 2245. CEUR-WS.org, pp. 643–652

Odell J. (1994) Power Types. In: J. Object Oriented Program. 7(2), pp. 8–12

Rodríguez A., Durán F., Rutle A., Kristensen L. M. (2019a) Executing Multilevel Domain-Specific Models in Maude. In: Journal of Object Technology 18(2), 4:1–21

Rodríguez A., Durán F., Rutle A., Kristensen L. M. (2019b) Executing Multilevel Domain-Specific Models in Maude. In: Journal of Object Technology 18(2), 4:1–21

Rodríguez A., Macías F. (2019) Multilevel Modelling with MultEcore: A Contribution to the MULTI Process Challenge. In: Proceedings of MULTI @ MODELS, pp. 152–163

Rodríguez A., Rutle A., Durán F., Kristensen L. M., Macías F. (2018) Multilevel modelling of coloured petri nets. In: Hebig R., Berger T. (eds.) Proceedings of MODELS 2018 Workshops, Copenhagen, Denmark, October, 14, 2018. CEUR Workshop Proceedings Vol. 2245. CEUR-WS.org, pp. 663–672

Rodríguez A., Rutle A., Kristensen L. M., Durán F. (2019c) A Foundation for the Composition of Multilevel Domain-Specific Languages. In: MULTI@ MoDELS, pp. 88–97

Rossini A., de Lara J., Guerra E., Rutle A., Wolter U. (2014) A formalisation of deep metamodelling. In: Formal Aspects of Computing 26(6), pp. 1115–1152

Somogyi F. A., Mezei G., Urbán D., Theisz Z., Bácsi S., Palatinszky D. (2019) Multi-level Modeling with DMLA - A Contribution to the MULTI Process Challenge. In: 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019. IEEE, pp. 119–127

Steinberg D., Budinsky F., Merks E., Paternostro M. (2008) EMF: Eclipse Modeling Framework. Pearson Education

Theisz Z., Bácsi S., Mezei G., Somogyi F. A., Palatinszky D. (2019) By Multi-layer to Multi-level Modeling. In: 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019. IEEE, pp. 134–141

Urbán D., Theisz Z., Mezei G. (2018) Self-describing Operations for Multi-level Metamodeling. In: Hammoudi S., Pires L. F., Selic B. (eds.) Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018, Funchal, Madeira - Portugal, January 22-24, 2018. SciTePress, pp. 519–527

Wolter U., Macías F., Rutle A. (Nov. 2019) The Category of Typing Chains as a Foundation of Multilevel Typed Model Transformations. 2019-417. University of Bergen, Department of Informatics

A Complete model of Acme software engineering process

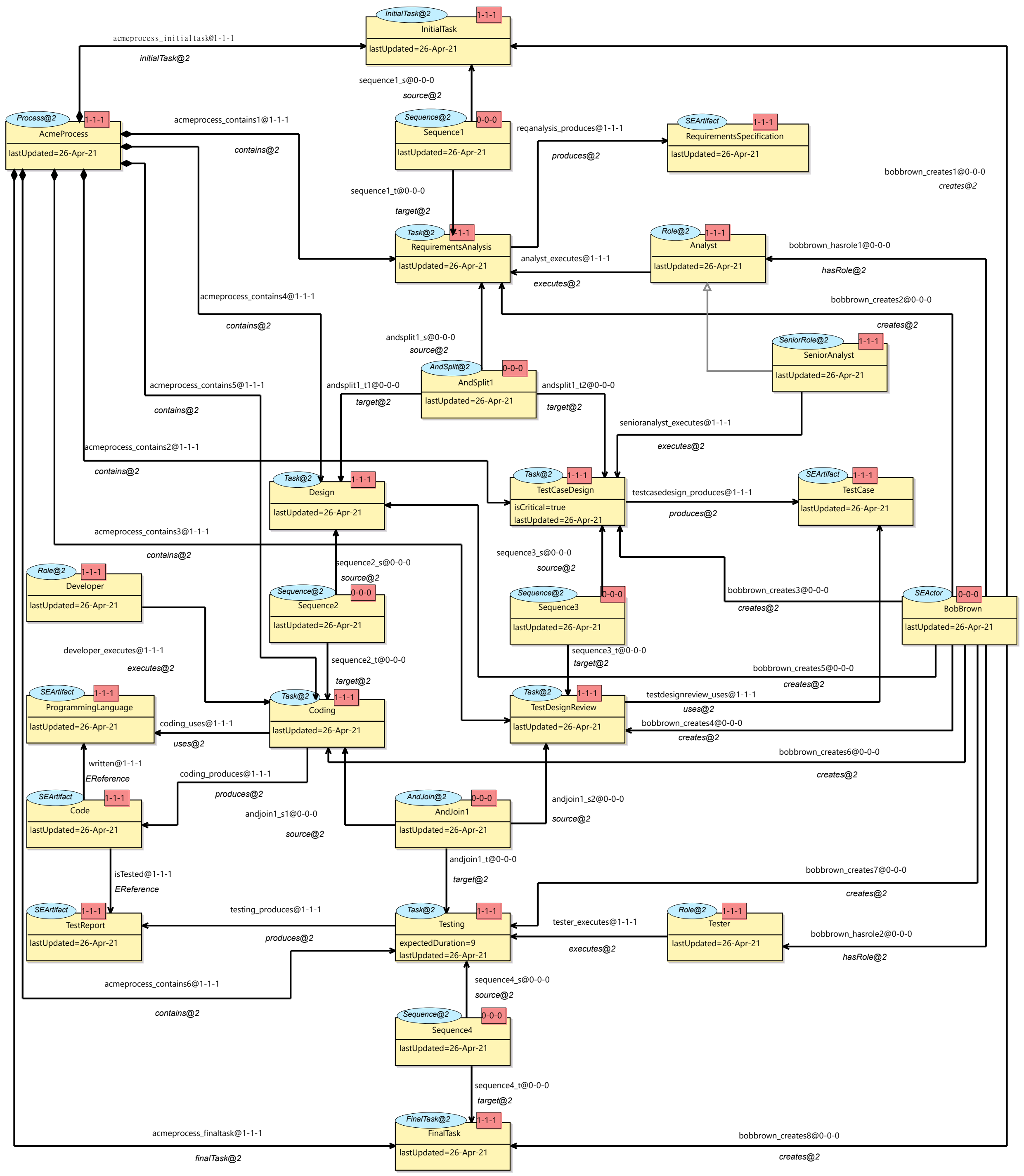


Figure 14: Level 3: Complete Acme software engineering process model

B Complete model of XSure insurance Claim Handling process

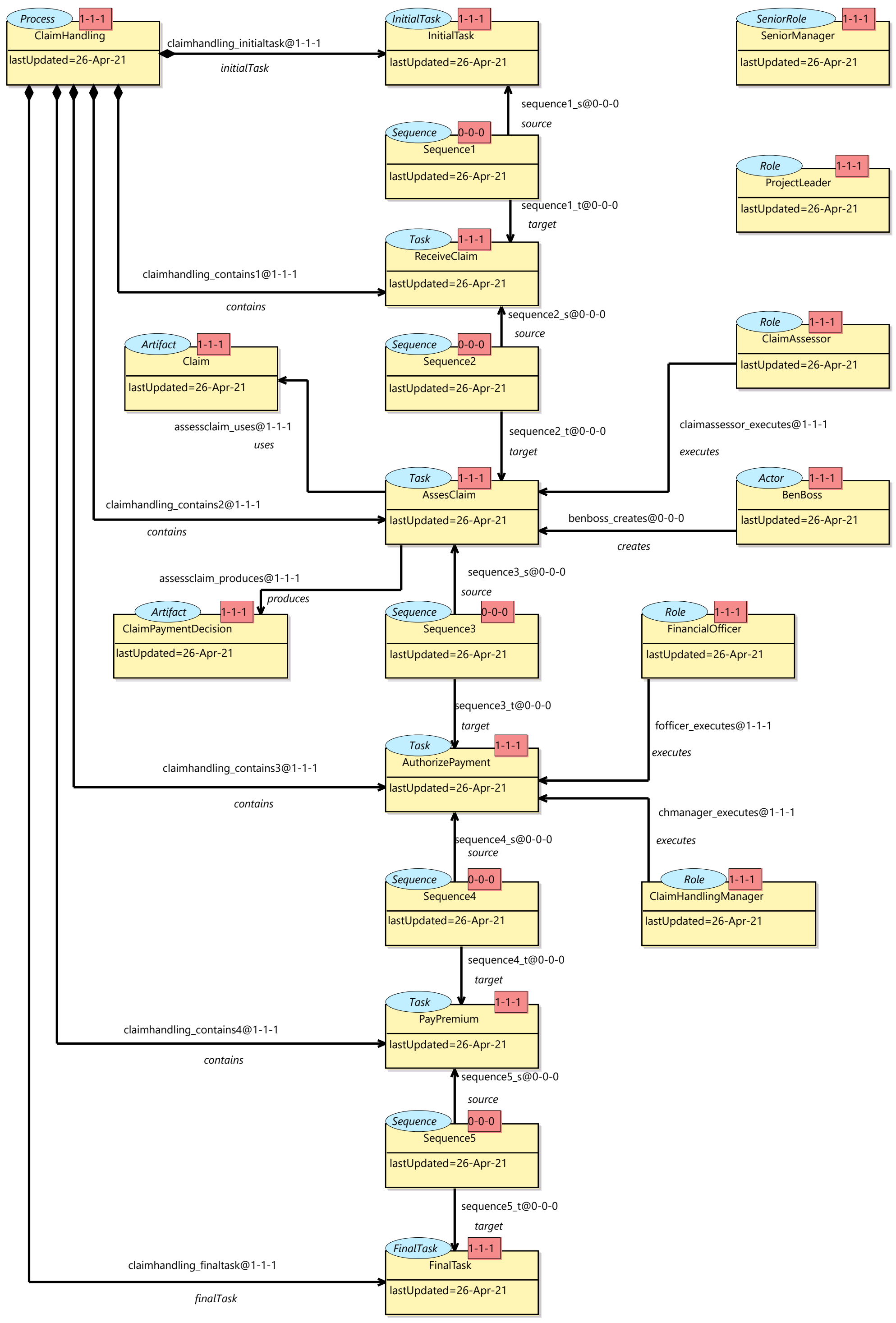

Process 1-1-1 ClaimHandling lastUpdated=26-Apr-21
InitialTask 1-1-1 InitialTask lastUpdated=26-Apr-21
SeniorRole 1-1-1 SeniorManager lastUpdated=26-Apr-21
claimhandling_initialtask@1-1-1
initialTask
sequence1_s@0-0-0
source
Sequence 0-0-0 Sequence1 lastUpdated=26-Apr-21
Role 1-1-1 ProjectLeader lastUpdated=26-Apr-21
sequence1_t@0-0-0
target
Task 1-1-1 ReceiveClaim lastUpdated=26-Apr-21
claimhandling_contains1@1-1-1
contains
sequence2_s@0-0-0
source
Artifact 1-1-1 Claim lastUpdated=26-Apr-21
Sequence 0-0-0 Sequence2 lastUpdated=26-Apr-21
Role 1-1-1 ClaimAssessor lastUpdated=26-Apr-21
assessclaim_uses@1-1-1
uses
sequence2_t@0-0-0
target
claimassessor_executes@1-1-1
executes
Task 1-1-1 AssesClaim lastUpdated=26-Apr-21
Actor 1-1-1 BenBoss lastUpdated=26-Apr-21
claimhandling_contains2@1-1-1
contains
benboss_creates@0-0-0
creates
assessclaim_produces@1-1-1
produces
sequence3_s@0-0-0
source
Artifact 1-1-1 ClaimPaymentDecision lastUpdated=26-Apr-21
Sequence 0-0-0 Sequence3 lastUpdated=26-Apr-21
Role 1-1-1 FinancialOfficer lastUpdated=26-Apr-21
sequence3_t@0-0-0
target
fofficer_executes@1-1-1
executes
Task 1-1-1 AuthorizePayment lastUpdated=26-Apr-21
claimhandling_contains3@1-1-1
contains
chmanager_executes@1-1-1
executes
sequence4_s@0-0-0
source
Sequence 0-0-0 Sequence4 lastUpdated=26-Apr-21
Role 1-1-1 ClaimHandlingManager lastUpdated=26-Apr-21
sequence4_t@0-0-0
target
Task 1-1-1 PayPremium lastUpdated=26-Apr-21
claimhandling_contains4@1-1-1
contains
sequence5_s@0-0-0
source
Sequence 0-0-0 Sequence5 lastUpdated=26-Apr-21
sequence5_t@0-0-0
target
FinalTask 1-1-1 FinalTask lastUpdated=26-Apr-21
claimhandling_finaltask@1-1-1
finalTask

Figure 15: Level 2: XSure insurance Claim Handling process model