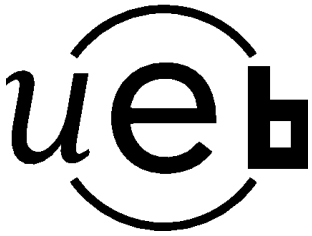
ANNÉE 2015



HABILITATION À DIRIGER DES RECHERCHES
UNIVERSITÉ DE RENNES 1

sous le sceau de l'Université Européenne de Bretagne

Mention : Informatique

École doctorale Matisse

présentée par

Benoit COMBEMALE

préparée à l'unité de recherche IRISA – UMR6074
Institut de Recherche en Informatique et Système Aléatoires / ESIR

Towards Language-Oriented Modeling

HDR soutenue à Rennes
le 4 décembre 2015

devant le jury composé de :

Franck BARBIER
Professor, University of Pau, France / *Rapporteur*
Lionel BRIAND
Professor, University of Luxembourg / *Rapporteur*
Betty H.C. CHENG
Professor, Michigan State University, USA / *Rapporteur*
Oscar NIERSTRASZ
Professor, University of Bern, Switzerland / *Rapporteur*
JOERG KIENZLE
Professor, McGill University, Canada / *Examinateur*
PETER D. MOSSES
Professor, Swansea University, United Kingdom / *Examinateur*
BENOIT BAUDRY
Research Scientist, INRIA, France / *Examinateur*
JEAN-MARC JÉZÉQUEL
Professor, University of Rennes 1, France / *Examinateur*

In memory of Prof. Robert B. France,
A mentor, an academic father and far beyond.
Rest in peace, Robert.

Foreword

In the French academic system, the 'habilitation à diriger des recherches' (HDR) is an additional degree after the PhD. This degree is required to officially supervise PhD students, to apply for full professorship, and more generally to run an independent research activity. The manuscript of an HDR is supposed to overview the research activities of the candidate, and demonstrate a clear vision leading to consistent contributions. In that sense, it is not meant to be a fully polished, self contained document like a PhD thesis.

This document constitutes the manuscript of my HDR. I overview a decade of research work in the fields of model driven engineering (MDE) and software language engineering (SLE). The contributions are organized so as to highlight the vision as well as the scientific method. Of course, these contributions would not have been achieved alone. This document also aims to present the complementarity of the research work conducted with the various students I have been pleased to supervise, as well as the place of the various collaborations with researchers and practitioners, in the elaboration of the resulting vision I defend in this document.

This document is meant to be first dedicated to the referees of my HDR who evaluate and report on my research activities, the breakthroughs addressed, and the resulting contributions. This document also offers a brief overview of the state of the art in the fields of MDE and SLE. Finally, this document would be also of interest for everyone willing to learn how to turn domain knowledge into added value and coordinated DSMLs.

Contents

Chapter 1

Introduction

After a brief introduction to the application and scientific context of my research work (Section 1.1), this first chapter introduces the overall objectives and challenges addressed in the contributions presented in this habilitation (Section 1.2). Then I present an overview of my contributions (Section 1.3) and the scientific and technological breakthroughs achieved (Section 1.4). Section 1.5 describes the research methods followed during my research activities. Since the contributions are the result of a collaborative effort, I list in Section 1.6 the various collaborations with students, researchers and software engineers who contributed to the results, and Section 1.7 presents the research grants, industrial contracts and collaborative projects which supported the overall research activities. Finally, Section 1.8 describes the organization of this document.

Contents

1.1 Context

1.1.1 Application Context

Software is increasingly pervasive in, among others, cyber-physical systems, systems of systems, and in the Internet of things. It supports numerous aspects of our daily lives (from online tax payments to smart orchestrated airplane controls, to autonomic cars and smart cities). For

increasingly broad and diverse application domains, the engineering of complex software-intensive systems involves:

- Multiple stakeholders, each with a specific domain expertise in a particular engineering area (e.g., hardware, mechanical, civil, electrical);
- Some forms of domain-specific modeling supporting expert analysis, design and early validation & verification processes;
- Software as the integration layer among all those domains and areas of expertise.

These characteristics have major impacts on software development processes:

- The involvement of multiple stakeholders in the development process requires separating concerns so that each stakeholder can focus on her tasks, without losing the big picture.
- Each of these domain specific concerns must be addressed with dedicated tools and methods to leverage the expertise of domain experts. Building such specific tools is still difficult and costly.
- At some point, these concerns must be integrated, whether at the binary level (e.g., components on automotive platforms), the framework level (e.g., OSGi), the programming language level (Aspect weaver in e.g. AspectJ), the model level, or a combination of these techniques.

Simulation and analysis based on the composition of early, concern-specific models are becoming an important technique for product quality and development efficiency. Scaling up the definition of domain-specific tools and methods, as well as the various integration techniques to face the increasing complexity of software-intensive systems are still daunting challenges by themselves.

1.1.2 Scientific Context

Model-Driven Engineering (MDE) aims at reducing the accidental complexity associated with developing complex software-intensive systems [166]. A primary source of accidental complexity is the large gap between the high-level concepts used by domain experts to express their problem statements and the low-level abstractions provided by general-purpose programming languages [79]. Manually bridging this gap, particularly in the presence of changing requirements, is costly in terms of both time and effort. MDE approaches address this problem through the use of modeling techniques that support separation of concerns and automated generation of major system artifacts (*e.g.,* test cases, implementations) from models. In MDE, a model describes an aspect of a system and is typically created for specific development purposes. Separation of concerns is supported through the use of different *modeling languages*, each providing constructs based on abstractions that are specific to an aspect of a system. For example, Generalized Stochastic Petri Nets can be used to create performance models [9], while the notation provided by the Simulink[1] tool is adapted to simulation models. MDE technologies also provide support for manipulating models; for example, there exists tool support for querying, transforming, merging, and analyzing (including executing) models. As such, modeling languages are "the heart and soul" of MDE.

Incorporating domain-specific concepts and best practices development experience into modeling languages can significantly improve software and systems engineer productivity and

[1] Cf. http://www.mathworks.com/products/simulink

system quality. To address this challenge, the modeling community is starting a technology revolution in software development, and the shape of this revolution is becoming more and more clear. Little, *domain-specific modeling languages* (DSMLs) are increasingly being developed to continuously leverage business or technical domain expertise of various stakeholders, and then used as formalisations of the domains to define relationships among them and support the required integration activities. A DSML provides a bridge between the (problem) space in which domain experts work and the implementation (solution) space. DSMLs are usually small and simple languages, focused on a particular problem or aspect of a (software) system. Domains in which DSMLs have been developed and used include those for automotive, avionics, and cyber-physical systems. Hutchinson et al. recently provided some indications that DSMLs can pave the way for wider industrial adoption of MDE [204]. This leads to a *language-oriented modeling*[2] which emerges in various guises (e.g., metamodeling, model transformation, generative programming, compilers, etc.), and in various shapes (from API or fluent API, to internal or external DSMLs).

Although there are many examples of the use of DSMLs to overcome the semantic gap between problem space and solution space, it has only been recently recognized that the development of a DSML is itself a significant software engineering task. Indeed, many DSMLs were designed in an ad-hoc way and without proper language engineering principles with clearly identified phases (e.g., decision, analysis, design, implementation, deployment, maintenance, evolution and adaptation) and artefacts. The development of DSMLs conceptually follows a unified partially ordered process: identification of the abstract syntax (i.e., the concepts and structures that constitute a DSML of interest); specification of the concrete syntax (i.e., the symbols and notations to be used by stakeholders); mappings from abstract to concrete syntax; specification of the semantics (which captures the meaning of the concepts and structures in the DSML, e.g., using mathematics, simulation, transformation); elaboration of an integrated development environment that allows stakeholders to write models in the DSML, check that models are well-formed, and support simulation, code generation, etc. While conceptually all approaches for DSML development generally follow this process, in practice they all use different techniques and technologies, all resulting from specific foundations. In the modeling community, *metamodeling* foundations have been defined in the last two decades. The core idea is that the very same notion of model is used to formalize models. We call such a special kind of model a metamodel. This vision has led to work, starting in the late nineties, on language workbenches that support the development of DSMLs and associated tools (e.g., model editors and code generators) [66].

Research on systematic development of DSMLs has produced a technology base that is robust enough to support the integration of DSML development processes into large-scale industrial system development environments. Current DSML workbenches support the development of DSMLs to create models that play pivotal roles in different development phases. Workbenches such as Microsoft's DSL tools[3], MetaCase's MetaEdit+[4], JetBrains's MPS[5], Eclipse Modeling Framework (EMF)[6], MontiCore[7] and the Generic Modeling Environment (GME)[8] support the specification of the abstract syntax, concrete syntax and the static and dynamic semantics of a DSML. These workbenches address the needs of DSML developers in a variety of application domains.

While DSMLs have been found useful for structuring development processes and providing

[2] By analogy with Language-Oriented Programming [202]
[3] Cf. http://www.microsoft.com/en-us/download/details.aspx?id=2379
[4] Cf. http://www.metacase.com/fr/mwb/
[5] Cf. https://www.jetbrains.com/mps
[6] Cf. http://www.eclipse.org/modeling/emf
[7] Cf. http://www.monticore.de
[8] Cf. http://www.isis.vanderbilt.edu/projects/gme/

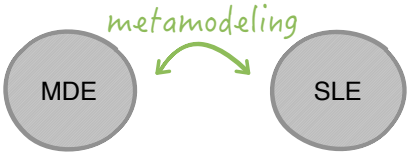


Figure 1.1: Research Domains

abstractions to stakeholders [97], their ultimate value has been severely limited by their user-understanding ambiguity, the cost of tooling and the tendency to create rigidity, immobility and paralysis (the evolution of such languages is costly and error-prone). The development of DSMLs is a challenging task also due to the specialized knowledge it requires. A language engineer must own not only quite solid modeling skills but also the technical expertise for conducting the definition of specific artifacts such as grammars, metamodels, compilers, and interpreters. *"Software languages are software too"* [73] and, consequently, languages development inherits all the complexity of general software development; concerns such as maintainability, re-usability, evolution, user experience are recurring requirements in the daily work of language engineers. As a result, there is room for application of software engineering techniques that facilitate the DSML construction process. This results in the emergence of what we know as *Software Language Engineering* (SLE) that is defined as the application of systematic, disciplined, and measurable approaches to the development, use, deployment, and maintenance of software languages [110].

The research work presented in this document are situated in the context of *metamodeling*, and aims to cross-fertilize MDE and SLE (cf. Figure 1.1). Complementarily, the research work leverages on background in language design and implementation, typing, software reuse, software product line, validation & verification techniques, and concurrency.

1.2 Objectives and Challenges

The adoption of DSML has major consequences on the industrial development processes. This breakdowns the development process into two complementary stages (see Figure 1.2): the development, adaptation or evolution by *language engineers* of one or several DSMLs, each capitalizing the knowledge of a given domain, and the use of such DSMLs by *language users* to develop the different system concerns.

Figure 1.2 depicts the two interdependent processes that continuously drive each other's. The main objective of the language engineering process is to produce a DSML which tackles a specific concern encountered by systems engineers in the development of a complex software-intensive system, together with its tooling. In particular, we focus in this habilitation on the tooling required for early validation and verification of the system design. Once an appropriate DSML is made available to systems engineers, it is used to express the solution to this specific concern in the final system. However, by definition, DSMLs are bounded to evolve with the domain they abstract. Consequently, systems engineers need to be well aware of end users' expectations in order to report their new requirements to the language engineers. A new evolved DSML is then produced by the language engineers, which is in turn used by systems engineers and so on and so forth. It is worthwhile to note that, although this is unlikely in large companies, these roles can be alternatively played by the same people in smaller organizations.

To support such a language-oriented modeling, it is worthwhile to provide tools and methods which help language engineer to give value in the DSMLs by leveraging on the knowledge of the language users about the domains and their integration altogether.

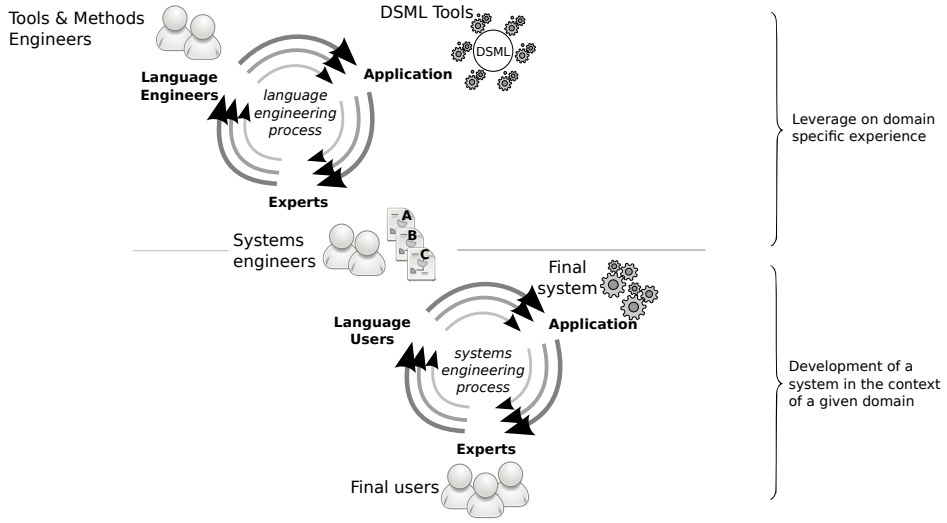


Figure 1.2: A big picture of *Language-Oriented Modeling*

This appealing vision raises many challenges. MDE encompasses the identification of added-value domain knowledge, and SLE covers the technical activities to develop DSMLs, but the interplay between both it is still challenging: how to turn application-level and business knowledge into added-value DSMLs, and then to scale-up with their multiplication due to the various stakeholders and application domains of interest? In a context where DSML development and evolution become daily activities for various software and systems engineers, it is now crucial to bridge the gap between MDE and SLE. The challenge is twofold: to leverage on domain knowledge in DSML specifications while making them reusable and customizable for other contexts; and to support the separation of concerns through the use of multiple DSMLs while ensuring their correct integration and consistency.

1.3 Overview of the Contributions

The development of new DSMLs as well as the evolution of existing ones become daily activities for various software and systems engineers. To support those activities, I first present foundational concepts and engineering facilities which help to capture the core domain knowledge into the various heterogeneous concerns of DSMLs (aka. *metamodeling in the small*, see left part of Figure 1.3), with a particular focus on executable DSMLs to automate the development of dynamic V&V tools.

As a major consequence of the adoption of MDE and SLE in industrial processes, there is a growing number of DSMLs to address the increasing number of application domains of interest, and the various stakeholders involved in each application domain. To address this challenge, I introduce relevant language interfaces atop a DSML implementation, and the associated composition operators to reuse and integrate multiple DSMLs (aka. *metamodeling in the large*, see right part of Figure 1.3).

The research work[9] aims to place DSMLs as a key pivot for the socio-technical coordination

[9] A seminar has been recently organized about the various activities about SLE for MDE I am involving in the DiverSE team: http://people.irisa.fr/Benoit.Combemale/sleseminar2015. All the slides are available on the webpage of the seminar, and provide an up-to-date content on the past results and ongoing work.

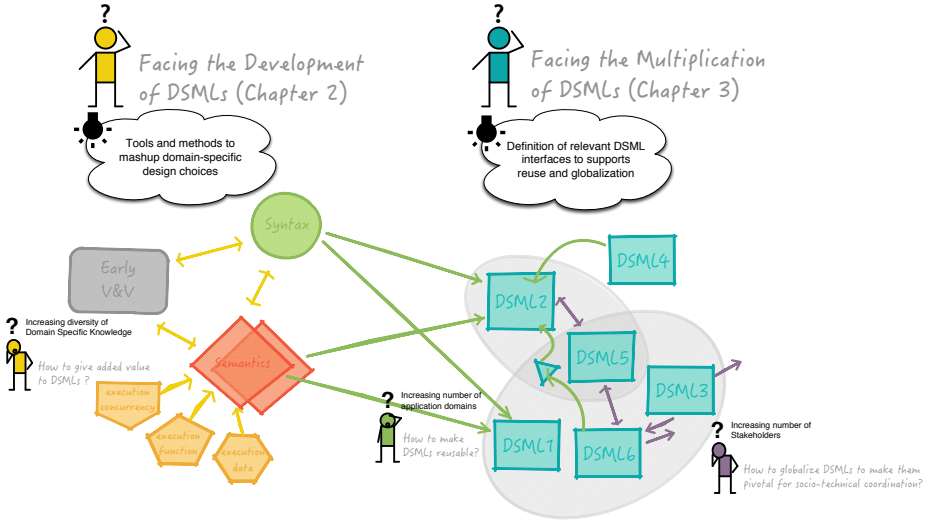


Figure 1.3: Challenges to support a *Language-Oriented Modeling*

in the development of complex software-intensive systems. The social coordination refers to the elicitation of the domains of expertise, as well as their social translucence over the various stakeholders which involve in the development processes. The technical coordination refers to the coordination of the various heterogeneous artefacts build by the stakeholders. Their coordination must be supported in various way for analysis, deployment, execution, etc.

1.4 Scientific and Technological Breakthroughs

In these research activities I explore various breakthroughs in terms of modularity and reusability of DSMLs. Based on model typing, I introduce new metamodeling foundations on top of the existing foundations. In particular, I propose a clear separation between the implementation of a language and its structural interface. This interface is then used to *type* the models, but also to define subtyping relations between them to ensure a safe substitutability of the corresponding language implementations. Inspired by software product lines, I also propose a complete approach and the corresponding facilities to define language families and manage syntactic and semantic variation points. From such a family, one can configure a particular DSML and then derive its complete implementation.

To ease the development of V&V tools for each DSML, we propose original patterns and a methodology for implementing executable DSMLs as well as generative and generic tools which automate the development of V&V tools (e.g., model checker, omniscient debugger). I also propose an original approach which bridges the gap between the concurrency theory and the algorithm theory, to integrate a formal concurrency model into the execution semantics of DSMLs. This concurrency model reflects the concurrency constraints in a particular application domain or for the deployment on specific modern platforms. Generic simulation and analysis tools are provided to reason over such a concurrency model.

Finally, I define an explicit behavioral language interface extracted from the concurrency model, used to define coordination patterns between heterogeneous executable DSMLs. Such

patterns are later used to automatically generate the coordination between specific conforming models.

All the contributions have been implemented in software platforms – the language workbench *Melange* and *the GEMOC studio* – and experienced in real-world case studies to assess their validity. In this context, I also founded *the GEMOC initiative*, an attempt to federate the community – both from industry and academia – on the grand challenge of *the globalization of modeling languages*.

1.5 Research Methods

The research work reported in this document is a combination of foundations and engineering facilities. As such, I also followed a combination of scientific methods and validation techniques as relevant as for each kind of research work.

I combine both deductive and analytical methods. While conceptual models are founded in well-defined theories (e.g., typing, substitutability, concurrency theory, etc.), engineering facilities are motivated and tailored with regard to real-world case studies and empirical observations. Consequently, the research results are experienced by, as appropriate, formal, empirical and experimental validations.

An important effort in my research work lies in the consolidation of the results in concrete and integrated tools. In particular, the research activities result in both the Melange language workbench, and the GEMOC studio. These two software environments make the research results concrete, and allowed the realization of concrete experimentations on real-world case studies. Though, they contribute to the dissemination of the underlying concepts and principles.

Finally, my research activities are strongly grounded in many collaborations. Most of the results are acknowledged to the various Master and PhD students, software engineers and post-doctoral researchers that I have been pleased to supervise (cf. Section 1.6). Also, the vision I developed has been motivated and experienced on case studies provided by industrial partners which are essential to keep focus and problem-driven the research activities (cf. Section 1.7). Finally, most of the ideas result from various discussions and collaborations with colleagues around the world, either within collaborative projects or through informal discussions in scientific events (visits of universities, workshops, seminars and conferences). I also enjoyed to involve in the organization of many scientific events, and to act as a founding member of the GEMOC initiative that aims to federate the community – both in academia and industry – on the chalenge of the globalization of modeling languages. All these community building activities reflect my way to conduct the research activities as a collaborative and dynamic process.

1.6 Supervision

The work presented here and the emergence of the overall vision result from collaborations I have had with many researchers all over the world, my colleagues in the DiverSE team, as well as students I supervised during their Masters and PhD thesis, and software engineers I worked with on specific projects.

Table 1.1 gives the list of PhD I co-supervised: it provides the amount of co-supervision work I took care of, the period, the defense date, the funding, and the topic. In addition to the PhD students I officially supervised at University of Rennes 1, France, I also enjoyed to closely work with various other PhD students in their research projects. This is particularly the case for Thomas Degueule (Univ. Rennes 1), Moussa Amrani (Univ. Luxembourg), Mounira

Name	Rate (%)	Period	Defense	Funding	Topic
Clément Guy	80	2010-2013	10/12/13	MESR Grant	Model Typing
Emmanuelle Rouillé	40	2010-2014	16/04/14	CIFRE, Sodifrance	Software Process
Erwan Bousse	80	2012-2015	03/12/15	MESR Grant	Trace Management
David M. Acuña	80	2013-2016		Project VaryMDE	DSL Variability
Marcelino R. Cancio	50	2015-2018		Project Clarity	DSL Adaptation

Table 1.1: PhD co-supervision from 2009 to 2015

Kezadri (Univ. Toulouse), Florent Latombe (Univ. Toulouse), Mattias Vara (Univ. Nice Sophia Antipolis) and Peter Wuliang Sun (Colorado State University, USA).

Also, I have been pleased to work with several post-doctoral researchers in the DiverSE team who contributed to both the research work and the supervision of the students. José A. Gailndo worked on variability in modeling languages, and Cédric Bouhours worked on model typing.

The research work presented in this thesis would not have been done without extensive engineering efforts to both experiment and validate the scientific contributions. Moreover, an additional engineering effort has been put in research platforms with long term support and evolution to disseminate worldwide the research results. This engineering effort results in two platforms that will be presented in this document, namely Melange and the GEMOC studio. To support this engineering effort, I have been pleased to work with various software engineers, including Didier Vojtisek who is research engineer at Inria and helped me a lot during the realization of the vision presented in this thesis, as well as various software engineers founding with projects, including François Tanguy and Dorian Leroy on the GEMOC studio and Fabien Coulon on Melange.

1.7 Grants, Contracts and Projects

The research work presented in this document has been supported by various research grants, bilateral contracts with industry, as well as international and national collaborative projects. They provided the necessary funding to realize the research work, including the research staff (internships, PhD students, post-doctoral researchers and software engineers) and scientific environment. These collaborations also provided great opportunities to motivate, challenge, experiment, validate and transfert our solutions in industrial settings.

Among others, the FUI project TOPCASED[10] (2005-2009) and the ITEA2 OPEES[11] (2009-2012) initially supported the development of the research work related to the execution semantics of modeling languages for early validation and verification. Then, the EU FP7 Marie Curie ITN (Initial Training Network) RELATE[12] (2011-2014), the CNRS PICS project MBSAR[13] (2013-2015, PI), and the Inria-Thales bilateral contract VaryMDE[14] (2011-2015, co-PI) supported the research work related to the reuse and variability management of modeling languages.

While the former activities are currently developed in the context of the ANR project GEMOC[15] (2012-2016, scientific coordinator) to support the coordination of execution semantics,

[10] TOPCASED (*Toolkit in OPen source for Critical Applications & SystEms Development*). Cf. http://topcased.org

[11] OPEES (*Open Platform for the Engineering of Embedded Systems*). Cf. http://opees.org

[12] RELATE. Cf. http://www.relate-itn.eu/

[13] MBSAR *(Model-Based Security Analysis at Runtime)*. Cf. http://gemoc.org/mbsar

[14] VaryMDE *(Variability in Model Driven Engineering)*. Cf. http://varymde.gforge.inria.fr/

[15] GEMOC. Cf. http://gemoc.org/ins

the latter is currently investigated through the LEOC project CLARITY[16] (2014-2017) with industrial partners such as Thales, Airbus and Areva, and the Inria-DGA bilateral contract FPML[17] (2014-2018, co-PI). I am the French representative in the management committee of the COST Action MPM4CPS[18].

The complete list of grants, contracts and projects is available at http://people.irisa.fr/Benoit.Combemale/projects.

1.8 Organization of the document

In the rest of this document I first present our contributions to face the development of added-value DSMLs (Chapter 2), and then our contributions to face the multiplication of DSMLs in industrial development processes (Chapter 3).

The message of Chapter 2 is twofold. First, we claim that language engineering techniques for designing and implementing disposable DSMLs are close to maturity. However, some challenges to help language engineers to properly leverage on their own application-level knowledge still need to be addressed. Hopefully, decades of research in software engineering already paved the way and software language engineering should leverage these facilities in order to tackle these challenges.

Second, we claim that the common view on software language design should fundamentally evolve. Rather than abstract syntax trees, metamodels, type checkers, parsers, code generators, compilers..., we need to model and represent a software language as the composition of a set of language design decisions, concerning, among others, the existing language-units solutions, variation points, features and usage scenarios that are needed to satisfy the requirements. Once we are able to represent software languages, in several phases of the lifecycle, in terms of the aforementioned concepts, changing and evolving software languages as well as developing the tooling is considerably simplified.

Chapter 3 investigates how to scale up with multiple DSMLs. First we investigate the multiplication of application domains in which DSMLs are used. While each domain requires some specificities and customizations, we observe that recurrent paradigms, patterns, model transformations would benefit to be reused from one domain to another. Then, we investigate the multiplication of stakeholders in the development processes of complex software-intensive systems. In particular, we explore how DSMLs can play a pivotal role in the socio-coordination of the various stakeholders. This requires the support of the coordinated use of various DSMLs. This leads to the concept of the globalization of DSMLs, that is, the use of multiple DSMLs to support coordinated development of diverse aspects of a system. For both the reuse and the globalization of DSMLs, we leverage on the contributions of the previous chapter to define relevant DSMLs interfaces, and the required composition operators between them.

Chapter 4 concludes this document with a wrap-up of the research activities conducted during the last decade and the main outcomes. Finally, Chapter 5 introduces a broader vision that pushes forward the use of DSMLs, with long term perspectives related to the adaptability and globalization of DSMLs.

[16] CLARITY *(éCosytème pour la pLAte-foRme d'Ingénierie sysTème melodY)*. Cf. http://www.clarity-se.org

[17] FPML *(Domain-Specific Metamodeling for Policy Filtering)*.

[18] MPM4CPS (*Multi-Paradigm Modelling for Cyber-Physical Systems*). Cf. http://www.cost.eu/COST_Actions/ict/Actions/IC1404

Chapter 2

Metamodeling in the Small: Facing the Development of DSMLs

In this chapter, I present an overview of my contributions to help language engineers developing application-level DSMLs that provide business added value and support early validation and verification of the conforming models. I present foundational concepts and engineering facilities which help to capture the core domain knowledge into the various heterogeneous concerns of DSMLs, as well as to check and automate some activities of the DSML development.
I first present a modular approach to implement DSMLs (Section 2.1) which comes with one dedicated meta-language per language concern and a specific composition operators which ensure a statically checked consistency between the various concerns. This approach also leverages the modularity of the DSML specification to provide efficient structural model checking (Section 2.2), based on model and metamodel slicing to properly scope the actual analysis. Then I present a metamodeling pattern which provides guidelines to design and relate the various concerns of executable DSMLs (Section 2.3), which is further refined to introduce formal and explicit concurrency model in the execution semantics of DSMLs (Section 2.4). The DSML concerns are supported by generic and generative approaches to automate the development of advanced language tooling based on execution traces (Section 2.5), such as omniscient debugging. Finally, Section 2.6 offers a wrap-up of the overall approach by presenting a step-by-step implementation of the UML Activity Diagram language, and describing the resulting environnement for model edition, execution, debug and animation.

Contents

2.1 Mashup of (Domain-Specific) Meta-Languages

The content of this section is an adapted excerpt from the following publication:

Jean-Marc Jézéquel, Benoit Combemale, Olivier Barais, Martin Monperrus, and François Fouquet. Mashup of metalanguages and its implementation in the Kermeta language workbench. *International Journal on Software and Systems Modeling (SoSyM)*, 14(2):905–920, 2015. [102]

In the model-driven terminology, DSMLs are generally defined using metamodels and consequently, DSML "programs" are generally referred to as "models". For instance, a model-driven development process may use a DSML to simulate the system in prototyping phases, another DSML to define its software architecture, and yet another one to specify the set of valid inputs so as to allow test-case generation. The goal of using multiple DSMLs is to improve various aspects of software: such as improving consistency with requirements, reducing development costs or reducing the number of bugs [166].

However, model-driven development is no silver bullet. One of its drawbacks is that industry, instead of relying on a small number of general-purpose languages, now needs many modeling environments of production-level quality. In other terms, language design and implementation goes far beyond an activity for a few experts only and becomes a challenging task for thousands of software or systems engineers and domain experts, which we call *language engineers* [97].

Language engineers who are responsible for designing and implementing a tool-supported DSML can of course use general-purpose programming languages such as Java. However, implementing a DSML and its respective tooling is complex. It requires orchestrating various heterogeneous concerns, as different as the definitions of abstract and concrete syntaxes, static

semantics (including the well-formedness rules), behavioral semantics, as well as extra-functional issues such as compile-time or run-time performance, memory footprint, etc.

This is why researchers and practitioners proposed *language workbenches* [85, 78, 135, 197]. A language workbench provides language engineers with languages, libraries or tools to ease the design and implementation of DSMLs. Centaur [17] is an early contribution in this field, more recent approaches include Metacase's MetaEdit+ [187], Microsoft's DSL Tools [42], Clark et al.'s Xactium [30], Krahn et al's Monticore [115] Kats and Visser's Spoofax [106] and Jetbrain's MPS [199].

In this section, we present the Kermeta language workbench designed for specifying and designing DSMLs [102]. In a nutshell, the Kermeta workbench involves one different meta-language per DSML implementation concern: one meta-language for the abstract syntax[1] (aligned with EMOF [150], cf. Section 2.1.1); one for the static semantics (aligned with OCL [153], cf. Section 2.1.2) and one for the behavioral semantics (The *Kermeta Action Language* that extends Xtend, cf. Section 2.1.3)[2]. The Kermeta workbench uses an original modular compilation scheme to compose the three different meta-languages responsible for mashing-up the different DSML concerns into a standalone implementation (cf. Section 2.1.4). Throughout the section, we illustrate all these features by presenting the implementation of fUML [155]. Finally, we evaluate the use of the Kermeta language workbench from the end user point of view (here, the language engineer) on the basis of this case study (cf. Section 2.1.5).

2.1.1 Concern #1: Abstract Syntax Definition

First of all, to build a DSML in Kermeta, one defines its abstract syntax (i.e., the metamodel), which specifies the domain concepts and their relations. The abstract syntax is expressed in an object-oriented manner, using the OMG meta-language EMOF (Essential Meta Object Facility) [150]. EMOF provides the following language constructs for specifying a DSML metamodel: package, classes, properties, multiple inheritance and different kinds of associations between classes. The semantics of these core object-oriented constructs is close to a standard object model that is shared by various languages (*e.g.,* Java, C#, Eiffel). *We chose EMOF for the abstract syntax because it is a* de facto *standard allowing interoperability with other tools.*

In practice, we use in Kermeta the *de facto* standard implementation Ecore provided by the Eclipse Modeling Framework (EMF) [182], and aligned with the standard EMOF.

Figure 2.1 shows the excerpt of the fUML metamodel, and depicts those concepts as a class diagram. In our Kermeta-based fUML design, we reuse the abstract syntax standardized and provided by the OMG. In practice, OMG provides the fUML metamodel in terms of EMOF and we automatically translate it into an Ecore-based metamodel. Since the abstract syntax is expressed as an object-oriented metamodel, a concrete fUML model (equivalent to a DSML progam) is composed of instances of the metamodel classes.

2.1.2 Concern #2: Static Semantics Definition

The static semantics of a DSML is the union of the well-formed rules on top of the abstract syntax (as invariants of domain classes) and the axiomatic semantics (as pre- and post conditions on operations of metamodel classes). The static semantics is used to statically filter incorrect DSML models before actually running them. It is also used to check parts of the correctness of

[1] We will also use the term "metamodel" to refer to it. This is one definition in the community. For some researchers, "metamodel" sometimes referred to abstract syntax plus static semantics.

[2] The concrete syntax is achieved thanks to a full compatibility with all EMF-based tools for concrete syntax, such as the de facto standards Xtext (see http://www.eclipse.org/Xtext) and Sirius (see http://www.eclipse.org/sirius)

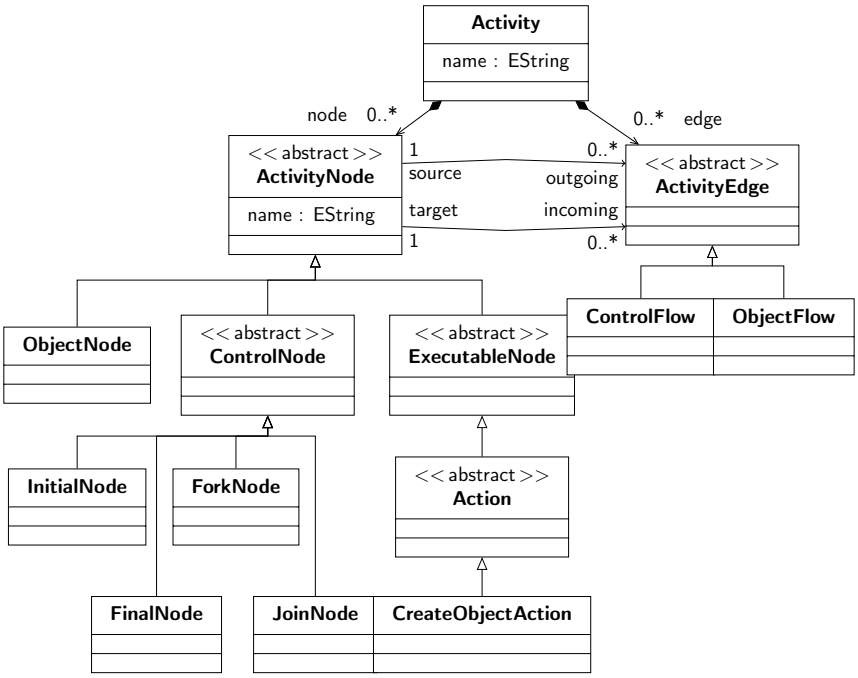


Figure 2.1: Excerpt of the fUML Metamodel

a DSML model's execution either at design-time using model-checking or theorem proving, or at run-time using assertions, depending on the execution domain of the DSML. Kermeta uses a dedicated active annotation (@Invariant) within Xtend to express the static semantics, directly woven into the metamodel using the active annotation @Aspect. The syntax of Xtend is close to the OMG Object Constraint Language (OCL) [153], and Kermeta statically checks that the invariant is side effect free.

As an example, the additional constraint *fUML_is_class*, defined in Listing 2.1, is applied over the specific action *CreateObjectAction*, and tells us that it can only be linked to an instance of *Class* (i.e. one can not create, say activities at run-time).

Listing 2.1: Weaving the Static Semantics of fUML into the Standard Metamodel

```
1 @Aspect(className=CreateObjectAction)
2 class CreateObjectActionAspect {
3 // The given classifier must be a class.
4   @Invariant
5   def boolean fUML_is_class(){ _self.classifier instanceof Class}
6 }
```

In the Kermeta workbench, the abstract syntax and the static semantics are conceptually and physically (at the file level) defined in two different modules. Consequently, it is possible to define different semantic variants for the same domain, i.e. to have a single Ecore metamodel shared by different static semantics, *e.g.,* to cope with language variants.

2.1.3 Concern #3: Behavioral Semantics Definition

EMOF does not include concepts for the definition of the behavioral semantics and OCL is a side-effect free language. To define the behavioral semantics of a DSML, we have created the

Kermeta Action Language, an extension of the action language Xtend that is used to express the behavioral semantics of a DSML. The extension provided to Xtend is inspired of the facilities of the initial in-house Kermeta action language [147]. It can be used to define either a translational semantics or an operational semantics of DSMLs. A translational semantics would result in a compiler while an operational semantics would result in an interpreter [33]. In this section, for sake of clarity, we will only present operational semantics. However, the language is exactly the same in both cases.

Xtend is imperative, statically typed, and includes classical control structures such as blocks, conditionals, loops and exceptions. Xtend also implements traditional object-oriented mechanisms for handling multiple inheritance and generics. Xtend also provides a powerful extension mechanism through the use of active annotations. Using this mechanism, the Kermeta action language complements Xtend with annotations for all EMOF constructs that must have a semantics at run-time such as containment and associations. First, if a reference is part of a bidirectional association, the assignment operator semantics handles both ends of the association at the same time. Second, if a reference is part of a containment association, the assignment operator semantics unbinds existing references if any, so that one object is part of another one. Finally, for multiple inheritance, Kermeta borrows the semantics from the Eiffel programming language [140].

Using the Kermeta action language, an operational semantics is expressed as methods of the classes of the abstract syntax [147]. Listing 2.2 is an excerpt of the operational semantics of fUML. Using the `@Aspect` annotation, a method "execute" is added to the metamodel class "Activity". The body of the method imperatively describes what is the effect of executing an activity. In this case, it consists of i) creating an activity node activation group and activating all the activity nodes in the activity, and ii) copying the values on the tokens offered by the output parameter nodes to the corresponding output parameter.

2.1.4 Composition Operators for the Mashup of Meta-Languages

As introduced above, mashing-up all DSML concerns in the Kermeta workbench is achieved through the annotation `@Aspect`.

In Kermeta, all pieces of static and behavioral semantics are encapsulated in metamodel classes. For instance, in Listing 2.2, the behavioral semantics is expressed in the metamodel classes "Activity". The `@Aspect` annotation enables language engineers to relate the language concerns (abstract syntax, static semantics, behavioral semantics) together. It allows language engineers to reopen a previously created class to add some new pieces of information such as new methods, new properties or new constraints. It is inspired from open-classes [32].

2.1.5 Evaluation of the Mashup From the Language Engineer Viewpoint

With the Kermeta-based design of fUML, all fUML concerns (abstract syntax, statics semantics and behavioral semantics) are separated in different units and the fUML runtime environment is the result of the mashup. In other terms, the Kermeta-based design clearly separates the three concerns of DSML implementation, and thus loyally reflects the structure of the specification.

For us, the driving motivation of having three different modules for implementing a DSML is: 1) allowing that each part of a DSML is done by different stakeholders (e.g., the abstract syntax by the standardization body and the compiler by the tool vendors), possibly in parallel; 2) supporting different variation points (either syntactic or semantic). The main con is that there is one language engineer, responsible for the meta-language integration, who must understand the different modules.

Listing 2.2: Operational semantics of fUML Activity with Kermeta

```
@Aspect(className=Activity)
class ActivityAspect inherits ExecutableAspect{
// the semantics of executing an activity
def void execute(runnable : Runnable)
  // Creation of an activity node activation group
  runnable.execute()
  var group : ActivityNodeActivationGroup init
      ActivityNodeActivationGroup
        .new()
  group.execution := runnable
  runnable.group := group

  // Activation of all the activity nodes in the activity
  runnable.group.activate(self.node,
    self.edge)
  var outputNodeActivations :
    OrderedSet<ActivityParameterNode>
    init runnable.group.
  fumlGetOutputParameterNodeActivations()

  // Copy the values on the tokens offered by output parameter nodes
      to the corresponding output parameters
  outputNodeActivations.each {outputNodeActivation |
    var parameterValue : ParameterValue init ParameterValue.new()
    parameterValue.parameter := (outputNodeActivation.asType(
        ActivityParameterNode)).parameter
    var tokens : Set<Token> init outputNodeActivation.
          fumlGetTokens()
    tokens.each { token |
      var val : Value init (token.asType(ObjectToken)).val
      if (val != void) then
        parameterValue.values.add(val)
      end
    }
    runnable.fumlSetParameterValue(parameterValue)
  }
}
```

Jezequel et al. [101] presented a reference manual on how to design and implement a DSML with the Kermeta language workbench. Beyond *how to*, we present in the rest of this section the advantages of our approach from the viewpoint of the language engineer (who plays the role of the end-user in our context).

2.1.5.1 Are concerns designed in different modules?

Our mashup approach provides two dimensions of modularity:

- modularity of domain concepts (metamodel concepts)
- modularity of language engineering concerns (parsing, static semantics, interpretation, compilation, etc.)

The first one of course does not bring anything new with respect to a Java based approach (e.g. the fUML reference implementation): this is just the usual class-based modularity found in OO languages, including MOF. Still it is very helpful when one wants to slightly change an existing concept, or even add or remove one into the meta-model.

So the originality of our approach lies in the modularity of language engineering concerns. It is accepted that the design of a modeling language deals at least with [91]: the abstract syntax (called a metamodel in the MDE terminology), the set of constraints on the abstract syntax (called static semantics), and the behavioral semantics. Using our approach, all these concerns are implemented in different modules:

- the abstract syntax is defined as an EMOF metamodel [150]. Technically speaking, it is completely defined in an Ecore file (e.g. `fuml-metamodel.ecore`). Please refer to [182] for more details about this file format. This module is standalone and does not depend from others modules.

- the static semantics is defined in a module dedicated to invariants, pre- and postconditions. When using OCL, this means creating a file, say `fuml.ocl`. This module imports the language metamodel (the Ecore file aforementioned) and has no other dependencies.

- the behavioral semantics (also known as execution semantics) is defined in a dedicated module using the Kermeta language, say `fuml.xtend`.

There is an exact one-to-one mapping between the abstract concerns of language design and the concrete design modules. The design of the modules are clearly layered so that there are no more spurious design dependencies than logical dependencies.

This architecture also enables language engineer to get rid of certain heavyweight design patterns, such as the design pattern *Visitor* which is often used to inject the semantics. Not using such patterns has two advantages. First, the DSML design is easier to understand and maintain. Second, at run-time, our DSML architecture requires less communication between objects (delegates and proxy calls), which contributes to a better efficiency. We refer the reader to [102] for a detailed evaluation of the approach.

2.1.5.2 Are concerns designed using appropriate meta-languages?

The research on aspect-oriented software development has shown [175] that not all languages are equal in term of implementing aspects. Certain concerns are well-suited to be implemented using domain-specific languages, sometimes called domain-specific aspect languages.

Using our approach, all concerns are designed using *meta*-DSMLs. The abstract syntax is designed using Ecore, the static semantics is designed using OCL and the operational semantics is designed using the Kermeta language (based on Xtend). Let us now review the advantages of certain particular *meta*-DSMLs for the related concerns.

- Metamodeling with EMOF: We argue that EMOF is especially appropriate to design DSML metamodels:

  - it is based on object-oriented modeling. Thus, any engineer who is fluent with object-oriented thinking is able to intuitively design a language metamodel with EMOF.

  - it is the result of years of discussion between DSML experts: it contains a lot of constructs that are known to be useful for metamodeling (*e.g.,* association, containment and multiple inheritance).

  - the tool-support for EMOF is good: there are several vendors and mature tools; it is possible to express EMOF metamodels using different textual and graphical editors.

- Static Semantics with OCL: OCL is well-suited for defining a DSML static semantics for the same reasons as those presented in the previous paragraph (maturity and tool support). Furthermore, since OCL is side-effect free, it is impossible for language engineers to accidentally introduce some pieces of behavioral semantics in the static semantics definition. In other words, the DSML design itself participates to ensuring the separation of concerns. On the contrary, using Java/AspectJ or another general-purpose programming language for expressing the static semantics would open the door to concern tangling.

- Operational Semantics with the Kermeta Language: As already stated, Kermeta is a workbench as well as a language. As a language (*Kermeta Language*), it has been specifically designed to express the operational semantics of languages. Let us now review a couple of examples that show the power of the Kermeta language with this respect.

  - *Manipulating collections of objects:* The operational semantics of DSMLs often deals with manipulating collections of objects. For instance a DSML for state machines would at some point traverse all transitions starting from a given state. Kermeta includes useful functions based on lambda expressions to manipulate collections: e.g. *collect*, *select* or *reject* (similar functions can be found in OCL and in some general purpose languages such as Smalltalk). Those constructions give a natural way to navigate through models compared to iteration over Java collections.

  - *Manipulating metamodel concepts within the operational semantics:* When implementing a behavioral semantics, one often manipulates concepts of the metamodel. If the language used to implement operational semantics does not natively support metamodeling concepts, the operational semantics is bloated with workarounds to approximate the metamodeling concepts. For example, Figure 2.2 represents an excerpt of the fUML implementation where the *Pin* class inherits from both *ObjectNode* and *MutiplicityElement*, and *isReady* is a method of *Pin* that expresses a piece of operational semantics. Let us compare how to handle two excerpts of this method in Java and Kermeta.

    This listing emphasizes one important characteristics of our approach. The Kermeta language enables direct manipulation of the concepts of the language without having to use special wrapper methods. For instance, on the right-hand side listing, `lower` directly refers to the field *lower* of the metamodel class *MutiplicityElement*. On the contrary, in Java, the language engineer always has to master the simulation of the semantics of multiple inheritance, association, containment, etc. In other words, our approach lowers the representational gap between the code of the operational semantics and the metamodel concepts (*i.e.* of the abstract syntax). Also, writing the operational semantics with the Kermeta language avoids bloating (code generation, annotations, etc.) due to embedding the semantics of the metamodeling language (EMOF) into a programming language that does not support it by default (*e.g.,* Java). Instead, the Kermeta language relies on the extension mechanism provided by Xtend, namely *active annotation*, to extend it with MDE-specific facilities and semantics.

```java
/****** In Java *********/
class Pin extends ObjectNode {
  // simulates multiple inheritance using
       delegation
  public multiplicityElement = new
       MultiplicityElement();
}
// elsewhere
class InputPinActivation extends PinActivation {
  public boolean isReady() {
    // The language engineer has to know
    // all low-level implementation choices,
    // here the use of delegation to simulate
    // multiple inheritance
    int minimum = this.node.multiplicityElement.
         lower;
  }
}
```

```
/****** In Kermeta *********/

@Aspect(className=InputPinActivation)
class InputPinActivationAspect {
  operation isReady() : Boolean is do
    // "lower" from MultiplicityElement
         on node
    var minimum : Integer init self.node.
         lower
  end
}
```

Figure 2.2: Comparison of the Java Implementation against the Kermeta Implementation.

2.2 Efficient Structural Model Checking

The content of this section is an adapted excerpt from the following publication:

Wuliang Sun, Benoit Combemale, Robert B. France, Arnaud Blouin, Benoit Baudry, and Indrakshi Ray. Using Slicing to Improve the Performance of Model Invariant Checking. *Journal of Object Technology (JOT)*, page 28, 2015. [184]

In MDD, models must conform to the well-formedness rules of the metamodel. Such well-formedness rules can be thought of as invariants of the metamodel. One needs to check the models to ensure that the invariants of the metamodel are satisfied using automated tools such as Eclipse OCL[3], so that the developers can identify potential problems during design time before they are used to generate code. However, the existing tools are inefficient for invariant checking on large models. For example, checking model instances consisting of hundreds of thousands of elements against a metamodel that includes 345 elements would take more than two hours [184]. Thus, there is a need for techniques that support invariant checking for large models and metamodels.

Slicing techniques [203] produce reduced forms of artifacts that can be used to support, for example, analysis of artifact properties. Slicing techniques have been proposed for different software artifacts, including programs (e.g., see [82, 203]), and models (e.g., see [5, 16, 68, 103, 114]). In the MDD area, model slicing techniques have been used to support a variety of modeling tasks, including model comprehension [5, 16, 114], analysis [100, 120, 121], and verification [68, 172, 173].

In model slicing techniques, *slicing criteria* are input data used to determine the elements that are included in slices. Model slicing techniques typically proceed in two steps: (1) The dependency between model elements of interest (e.g., elements satisfying a *slicing criterion*) and the rest of the model is analyzed using heuristics related to a model's properties (e.g., the structure of a model); and (2) a fragment of the model consisting only of elements satisfying a slicing criterion, is extracted from the model.

We introduce the model slicing technique to the invariant analysis process [184]. The approach aims to improve the size of the model that can be checked using invariant checking tools. The approach is not intended to improve the existing invariant checking algorithms. Instead, the approach aims to rely on the modular approach introduce in the previous section to reduce the size of the checking inputs to make the analysis more efficient. It means our approach

[3]Cf. http://projects.eclipse.org/projects/modeling.mdt.ocl

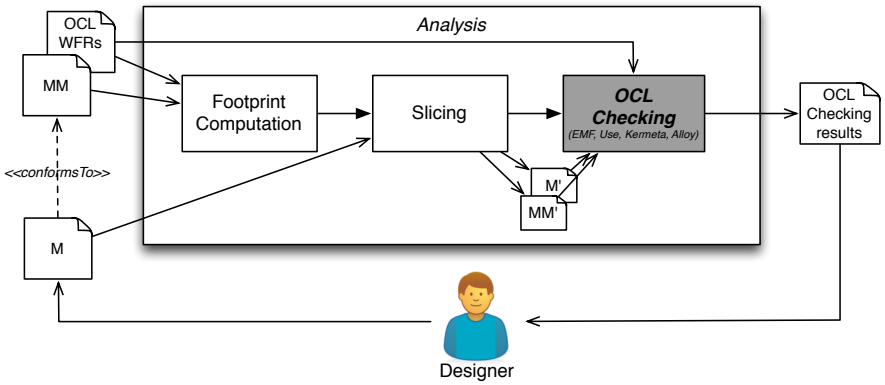


Figure 2.3: Approach overview

preprocesses the input of the invariant checking process, and thus is agnostic to the checking technologies the software developers are working with.

We have developed a framework that provides: (1) an implementation of the model slicing technique; (2) an implementation for checking models against invariants defined in the metamodels. The framework was implemented using Java and the Eclipse Modeling Framework (EMF) [182]. Even though the evaluation framework builds upon Java and Eclipse, the slicing technique is not bound to a particular technological space, and it can be implemented using any language and framework. We have evaluated our technique to check whether (1) the slicing improves the efficiency of the invariant checking, and (2) the invariant checking results for the sliced models are the same as the unsliced models. We have evaluated our approach with the Java metamodel and 73 models produced by reverse engineering Eclipse plugins. The evaluation we performed provides evidence that the proposed slicing technique can significantly reduce the time to perform the invariant checking while preserving the checking results. We also show that the invariant checking approach described in the paper can offer similar performance gains on small manually built models (e.g. hundreds of elements).

2.2.1 Approach Overview

Figure 2.3 shows an overview of the proposed invariant analysis approach (we refer the reader to [184] for all the details about the approach). The input of the checking includes a metamodel (*MM*), a model (*M*), and one or many OCL invariants (*Well-Formedness Rules*). First, the approach computes a footprint of the OCL invariants on the metamodel. A footprint refers to part of a metamodel that contains all elements that affect the outcome of an operation [100]. In this paper a footprint refers to all metamodel elements that are directly referenced by the input OCL invariants. Second, the footprint serves as slicing criterion, and is used to generate a sliced metamodel (*MM'*) from the input metamodel. The sliced metamodel (*MM'*) includes (1) all the metamodel elements from the footprint, and (2) all the subclasses of the classes in the footprint. Third, the sliced metamodel (*MM'*) is used to generate a sliced model (*M'*) from the input model. The sliced model (*M'*) contains only model elements that are instances of metamodel elements in *MM'*. Finally, the sliced metamodel and model with the invariants are fed into the tools for invariant checking.

2.2.2 Evaluation Results

We conducted an evaluation which aims to answer the following research questions [184]:

- RQ1: Can the slicing technique significantly improve the efficiency of the invariant checking? Our experiments revealed that the proposed slicing technique can significantly reduce the time to perform the invariant checking, achieving checking speedup ranging from 1.5 to 36.0.

- RQ2: Is the slicing technique ensured to preserve the invariant checking results? We show that checking an invariant in the sliced model is equivalent to checking it in the unsliced model, and the proposed sliced models are sufficient to the invariant checking.

2.3 The xDSML Pattern: A Metamodeling Pattern for Model Execution

The content of this section is an adapted excerpt from the following publications:

Benoit Combemale, Xavier Crégut, and Marc Pantel. A Design Pattern to Build Executable DSMLs and associated V&V tools. In *19th Asia-Pacific Software Engineering Conference (APSEC 2012)*, Hong Kong, December 2012. IEEE. [35]

Benoit Combemale, Xavier Crégut, Pierre-Loïc Garoche, and Xavier Thirioux. Essay on Semantics Definition in MDE. An Instrumented Approach for Model Verification. *Journal of Software (JSW)*, 4(9):943–958, November 2009. [33]

Model executability is now a key concern in MDE, especially to support early validation and verification (V&V) in the development process, such as debugging [43], model checking [33] and runtime verification [126].

Recently, several ways have been explored to implement the execution semantics of DSML. Basically, they map the abstract syntax, defined by the metamodel, to a semantic domain [91]. Most proposals translate models into an existing semantic domain in order to reuse available tools (*e.g.,* simulators or model-checkers). Such a semantics, called *translational semantics*, is used for instance by the group pUML in order to formalize some UML diagrams [29]. Even if more expressive languages like Maude [161] or FIACRE [15] may be used to ease the writing of the translation between the DSML high level concepts and the formal language low level ones, this approach may require complex transformations to implement the semantic mapping. Furthermore, execution results are only obtained in the target domain. Getting back the results in the source language is difficult and usually requires to extend its abstract syntax in order to model these results.

Other approaches propose to weave executability into metamodels using an action language (e.g., Kermeta [147], xOCL [31] or even Java with the EMF API). Similarly, in-place model transformations, including graph transformations [163], were widely investigated to give a declarative specification of the execution semantics. For example, Markovic *et al.* use QVT [152] to express rewriting rules that gradually compute the values of an OCL expression [130]. Kuske et al. [118] have used graph transformation to define the executable semantics for some UML diagrams. These approaches allow a more intuitive definition of executable DSMLs. The semantic domain is an extension of the abstract syntax, and the semantic mapping is defined using an action language. Thus, the language engineer has only to deal with concepts of the DSML and not with another language and an explicit mapping. Nevertheless, such approaches require to implement for each DSML all the execution-based tools.

In all cases, the definition of DSMLs is facing today hard methodological problems for the specification of tool supported execution semantics. Such DSMLs, called *executable DSML* (xDSML), are often empirically defined without any uniformity and underlying best practices [28]. For example, the information capturing the state of a model being executed, a key part of the semantic domain, is often scattered in a tool-specific way, without any explicit relation to the abstract syntax. Thus, different tools such as simulators, model checkers or code generators may easily be inconsistent, and not interoperable as they rely on slightly different semantic domains. In the same way, no methodology to define an executable DSML provides the flexibility to associate different semantics to the same DSML, to combine different models of computation (*e.g.,* multi-modeling), and to easily weave time and communication models; nor the evolvability to manage semantics changes.

Consequently, semantics-based tools (*e.g.,* simulators and graphical animators) are most of the time redefined without any capitalization (*e.g.,* dynamic execution related information, execution engine, etc.), and without any guidances to ease this error prone and time consuming development task.

In this section we present a general, reusable and tool-supported approach, to assist a language engineer in the definition of an execution semantics and the related tools [35][4]. It relies on capturing the different concerns involved in the definition of an executable DSML. These concerns are reified, in a structural design pattern to support executability into DSML: the xDSML pattern[5]. It addresses several common use cases relying on execution semantics, especially model V&V. Based on this pattern, generic and generative approaches are proposed to partially or totally automate the definition of DSML tools for V&V.

2.3.1 Motivation

The designer of a model that describes a system behavior usually needs to simulate and animate it to check whether it behaves as expected. Unfortunately, the metamodel does not generally describe all the information that has to be managed at execution time (i.e. the semantic domain). For example, a language for State Machines would define the concepts of `Region`, `State`, `Transition`, `Event`, etc. but would lack the notions of active states in a region, or of fireable transitions.

Also, no elements are available to store the sequence of events received by a state machine. For instance, events injected into a particular state machine will trigger fireable transitions and change the current states of the regions. Obviously, the way the system reacts to the stimuli defines its execution semantics. This reaction updates the execution-related data according to the current state of the model and the received stimulus.

In the end, complete execution traces must be stored to be able to analyse particular execution of a given state machine (e.g., simulation, model checking, etc.).

We have highlighted that model execution requires the extension of a DSML metamodel with: i) the definition of information managed during execution, ii) the definition of the stimuli that trigger the evolution of the model, iii) the organization of stimuli as scenarios, iv) the definition of an execution semantics (or transition function) that describes how the model state evolves when a stimulus occurs.

An *executable DSML* (xDSML) is a DSML which defines the execution of its conforming models for a particular purpose. Therefore, an executable DSML at least includes the definition

[4] This work has been initially applied in Topcased [72], an open-source MDE toolkit for safety critical application design, and more recently in the context of the ANR project GEMOC, see `http://gemoc.org/ins`.

[5] We use the term *pattern* in metamodeling, similarly to its use in modeling (*e.g.,* in [83]). In this section, we follow the common design pattern description format used in [83].

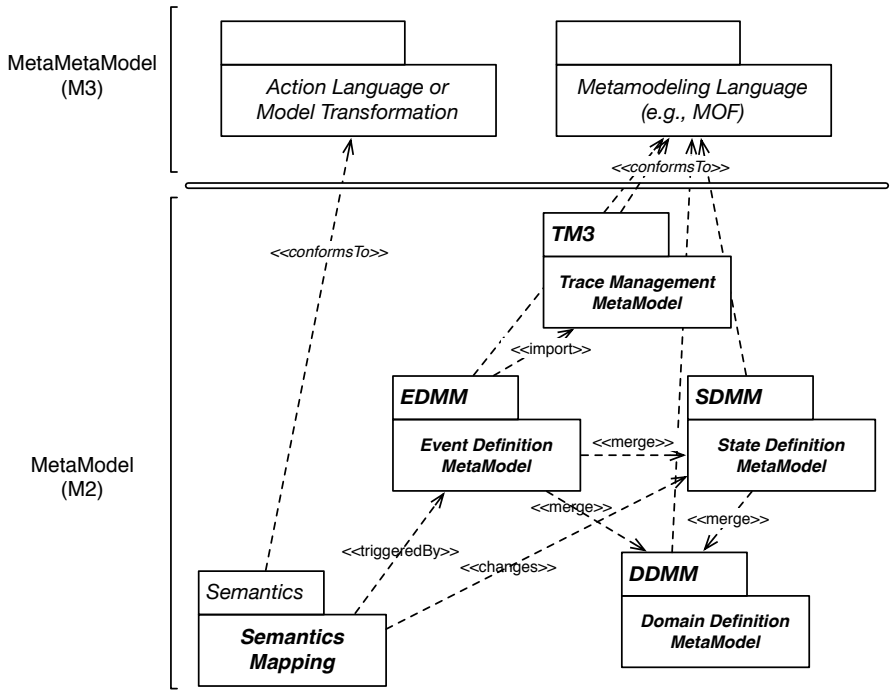


Figure 2.4: The xDSML pattern

of its language abstract syntax, and its execution semantics (including semantic domain and semantic mapping related information).

We propose to reify execution related elements to make them explicit and manageable. We aim to provide flexibility, evolvability and interoperability in the semantics definition. Furthermore such elements must ease the development of tools related to model execution, for example V&V tools.

2.3.2 Structure

Figure 2.4 shows the structure of the proposed xDSML pattern. It is built from four structural parts (detailed in the next subsection) that are woven together using the «`merge`» and «`import`» predefined package operators of MOF [150]. These parts organize the data related to the DSML and its execution semantics. A fifth part called *Semantics* provides the execution semantics itself relying on the previous four parts (i.e., the semantic mapping based on the previous reification of the semantic domain information). Because it is a pattern to organize data at the metamodel level (i.e., a *metamodeling pattern* [28]), the structure shows dependencies between packages that represent parts of a metamodel. This pattern is architectural like *MVC* or *3-tiers*. It emphasizes the common structure that a metamodel for an xDSML should use in order to define the language semantics. In addition to provide guidelines in language definition, the purpose is to be able to define generic and generative tools relying on that architecture.

2.3.3 Participants

Domain Definition MetaModel (DDMM) It is the usual metamodel used by standardization bodies to define the modeling language. It provides the key concepts of the language (representing the considered domain) and their relationships. For instance, the UML metamodel defined by the OMG is a DDMM. Usually, the DDMM does not contain all the execution-related information. For instance, the UML DDMM does not formalize the notions of *active state* nor *event queue*. Thus, even if a model describes the implicit potential behavior of a system, it does not usually provide explicitly the elements for its execution.

State Definition MetaModel (SDMM) During the execution of a model, additional data is usually mandatory for expressing the execution itself (aka. dynamic information). Such data must be manipulated and recorded (in the form of metaclass instances). For example, each active UML region must have one active state and a state machine must store the sequence of received events. These execution related data make up the SDMM, and are related to the semantic domain: the data required to express the execution semantics. Thus the SDMM is built on top of the DDMM. For example, the UML State Machines SDMM may add a reference from `Region` to `State` (both defined in the DDMM) to record the active state of one region.

Event Definition MetaModel (EDMM) The EDMM of a given DSML specifies the concrete stimuli (called runtime events) that drive the execution of a model that conforms to this DSML. These stimuli are not only concrete system hardware events, but also more abstract software events like storage events for reading or writing, communication events for sending or receiving, clock events as ticks, function events like computation results given parameters, etc. Concrete stimuli define properties of events related to the formal execution semantics to be supported.

As an illustration, the runtime event we consider for the UML State Machine stores an UML event in a state machine queue. When the UML event in the queue is handled by the state machine, it fires the transitions that it triggers.

Trace Management MetaModel (TM3) The TM3 is specific to a particular model of computation (MoC) and is reused for all DSMLs using this MoC. As an example, Figure 2.5 shows a simplified TM3 dedicated to discrete-events system modeling [210]. It defines three main metaclasses called `Trace`, `Scenario` and `RuntimeEvent`. `RuntimeEvent` is an abstract metaclass which reifies the concept of stimulus. It is an abstraction for any kind of semantic related stimulus defined in the EDMM. To this end, `RuntimeEvent` is imported in the EDMM, and all the concrete runtime events must inherits from it. This metaclass has executability-related features, like (partially ordered) dates of occurrence (i.e., symbolic representation of the time when the runtime event occurs). Any `RuntimeEvent` that triggers a semantic action involving a state change should have a reference to its source and target states information in the SDMM. `RuntimeEvent` instances fall into two categories, which are modeled by the `RuntimeEventKind` enumeration. Exogenous runtime events are injected by the environment, while endogenous runtime events are produced internally by the system in response to another runtime event (cf. `cause` in Figure 2.5). As stated by the OCL constraint in Figure 2.5, a scenario is made of exogenous runtime events whereas a trace corresponds to one possible execution of a scenario and is thus composed of any kind of runtime events. A more sophisticated trace management metamodel or a "standard" one (like the UML Testing Profile [154]) may be integrated in our pattern.

In Section 2.5, I present recent work that benefits from the xDSML pattern to automatically generate an efficient TM3 and the corresponding runtime manager.

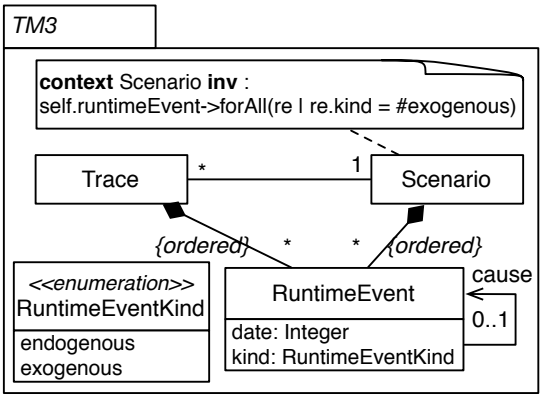


Figure 2.5: A simplified TM3 for Discrete-Events Modeling

Semantics The last and key participant is the package *Semantics*. It abstracts both the semantic mapping [91] (DSML-specific part) and the interactions with the environment (MoC-specific part). It describes how the running model (SDMM) evolves according to the stimuli defined in the EDMM. An important point in applying the pattern is to define the content of the package *Semantics* that depends on the application context. On the one hand the semantic mapping may be explicitly defined as a transition function and thus conforms to an action language (a.k.a. operational semantics). In this case, the four previous participants correspond to the semantic domain. On the other hand, the semantic mapping may be implicitly defined thanks to a translation to another language (a.k.a. translational semantics). Consequently SDMM and EDMM do not correspond to the semantic domain but help in defining the mapping, and in getting results back.

Finally, the semantic mapping may be expressed as predicate functions that specify the state of the model (a.k.a. axiomatic semantics). In such a case, and similarly to the operational semantics, the SDMM and EDMM are used as the semantic domain.

2.3.4 Consequences

According to the xDSML pattern, an xDSML is supported by an executable metamodel MM_x composed of three DSML-specific parts (DDMM, SDMM, and EDMM) and one MoC-specific part (TM3):

$$MM_x = \{DDMM, SDMM, EDMM\} \cup \{TM3\}$$

MM_x reifies the elements involved in model execution. The DDMM is the starting point. It is usually standardized and cannot be changed in order to preserve interoperability. The TM3 is shared by any DSMLs relying on the same MoC. Thus, a semantics is defined by a triplet (SDMM, EDMM, *Semantics*). The SDMM and the EDMM introduce the needed information to express the execution semantics (i.e. the semantic domain) whereas the package *Semantics* implements the semantic mapping. These three different parts should not be defined independently in order to reduce the risks of inconsistencies. Any change in this triplet entails a new semantics. In order to reduce these risks, we propose through the use of this pattern to reify the various aspects linked to the definition of the execution semantics in order to allow systematic specification, analysis and validation of an executable DSML metamodel.

Applying this pattern produces several consequences, both for the definition of the semantics, and for the definition of the execution-related tools.

2.3.4.1 Definition of the Semantics

The pattern allows a modular implementation of the execution semantics (i.e., an implementation that is separated out, encapsulated, and easily replaceable) with respect to the core language metamodel, the DDMM. The specification of the DSML semantics is split in two parts: first, a generic MoC based on the TM3, and shared with other DSMLs; and then DSML specific elements based on the SDMM and EDMM. This strong property provides several benefits described here after.

It favors the evolvability of the semantics during the DSML lifetime thanks to the separation of concerns involved in the definition of an execution semantics.

It eases the factorization of commonalities. The pattern favors the definition of a family of semantics for a single language as well as the semantics of a family of languages. For example, semantic variation points (like in UML) lead to different but similar semantics definitions. In most cases, SDMM and EDMM are the same and only the package *Semantics* has to be adapted.

It provides flexibility in the association of semantics to a given DSML in order to define several purpose driven semantics for the same DSML. Obviously, runtime information (SDMM), concrete runtime events (EDMM) and the package *Semantics* are dependent on the user purpose during the execution of models. For instance, the user may prefer to carry out more abstract execution with fewer runtime events and/or runtime information that demonstrates one aspect of the system under assessment or the user may want to define a fine-grained semantics that exhibits most aspects of the system. Each semantics will have its own set of events in the EDMM and states in the SDMM.

No specific method is enforced to apply the pattern. Nevertheless, we have proposed in [34] a method for the definition of DSML execution semantics dedicated to verification activities. It advocates a property driven approach: only runtime information and events required to evaluate properties of interest to the end user are described. In doing so, the EDMM and SDMM are a minimal mandatory subset of data to express the semantics relevant for the user, as advocated by the substitutability principle [146].

The definition of the package *Semantics* is postponed. The pattern is mainly an architectural pattern that helps in structuring information required to make a DSML executable while ensuring interoperability between tools based on this DSML. Thus, the semantic mapping and the interaction with the environment are not described in the pattern (as discussed in Section 2.3.3). According to the purpose of empowering a DSML with execution, the content of the package *Semantics* may be detailed. In most cases, the architecture of the MM_x eases the definition of the package *Semantics*. However, for scalability, efficiency, and some time readability purposes, it might be useful to introduce a new metamodel not relying on the standard DDMM. For example, the use of matrices to encode Petri nets instead of graphs is mandatory to allow the execution of huge models. This is also true in the case of General Purpose Modeling Languages (GPML) whose standard metamodel (DDMM) and semantics can be extremely complex. The introduction of purpose specific metamodels allows to ease the definition of the semantics for a subset of the language that the end user wants to assess.

Semantics is discrete event oriented. The EDMM part of the pattern stresses the use of discrete events to represent system stimuli. It may not be well-suited for all systems, like

continuous one. Nevertheless, we can notice that when one wants to observe a continuous system, a discretization (on events or time) is performed. Thus, the pattern is still applicable as this is done in PTOLEMY II [124] for example. Time may be managed continuously as part of the MoC or discretized as runtime events.

2.3.4.2 Definition of the Execution-Related Tools

The formalization of pattern elements favors the definition of generic and generative execution-based tools. Section 2.5 introduces a combination of generative and generic approaches to support execution traces in general, and omniscient debugging in particular.

Several models of computation (MoCs) may be used to support symbolic execution semantics. The description of the EDMM and TM3 might give the impression that the semantics is restricted to a discrete event MoC. In fact, these parts of the pattern define the discrete observations and interactions between the user/environment and the system, but any MoC can be used, including continuous ones. Our aim is to describe systems that in the end will be managed by either discrete software or human end users. Both can only handle a finite discrete history of the system. The MM_x architecture is strongly based on the user point of view: observation of the interaction between the model and its environment (depicted by the model state) at some key points in time represented by the runtime events. However, the package *Semantics* can implement any MoC or abstract the translation to an existing one (Section 2.4).

Cosimulation and models at runtime can be integrated. The package *Semantics* can also be implemented as a wrapper over, either real physical systems in which sensors and actuators are mapped to MM_x directly or through software layers, or existing softwares and execution engines. Several DSMLs can also be integrated through shared data in their MM_x and synchronization/cooperation in their packages *Semantics*.

It favors interoperability between the various semantics-related tools for a given DSML. Different kinds of tools may be based on the same executable DSML (*e.g.,* model simulator and graphical animator, *model-checking* based verification tool). The separation between MM_x and the package *Semantics* makes possible to share data between tools (i.e., a counter example provided by a verification tool can be analyzed using a graphical animator). However, this relies only on structural similarities and thus requires to assess the compatibility of both packages *Semantics* (i.e., by checking the bisimilarity of the transition relations).

2.4 Weaving Concurrency in Modeling Languages

The content of this section is an adapted excerpt from the following publications:

Florent Latombe, Xavier Crégut, Benoît Combemale, Julien Deantoni, and Marc Pantel. Weaving Concurrency in eXecutable Domain-Specific Modeling Languages. In *8th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2015)*, Pittsburg, USA, October 2015. ACM. [122]

Julien Deantoni, Papa Issa Diallo, Ciprian Teodorov, Joël Champeau, and Benoit Combemale. Towards a Meta-Language for the Concurrency Concern in DSLs. In *Design, Automation and Test in Europe Conference and Exhibition (DATE 2015), PI*,France, March 2015. [53]

Benoit Combemale, Julien Deantoni, Matias Vara Larsen, Frédéric Mallet, Olivier Barais, Benoit Baudry, and Robert France. Reifying Concurrency for Executable Metamodeling. In *6th International Conference on Software Language Engineering (SLE 2013)*, LNCS, USA, 2013. Springer-Verlag. [38]

Benoit Combemale, Cécile Hardebolle, Christophe Jacquet, Frédéric Boulanger, and Benoit Baudry. Bridging the Chasm between Executable Metamodeling and Models of Computation. In *5th International Conference on Software Language Engineering (SLE 2012)*, LNCS, Germany, 2012. Springer. [39]

The emergence of modern concurrent systems (e.g., Cyber-Physical Systems and Internet of Things) and highly-parallel platforms (e.g., many-core, GPGPU and distributed platforms) call for Domain-Specific Modeling Languages (DSMLs) where concurrency is of paramount importance. Such DSMLs are intended to propose constructs with rich concurrency semantics, which allow system designers to precisely define and analyze system behaviors. However, implementing the execution semantics of such DSMLs is a particularly difficult task. Most of the time the concurrency model remains implicit and ad-hoc, embedded in the underlying execution environment.

The lack of an explicit concurrency model prevents: the precise definition, the variation and the complete understanding of the DSML's semantics, the effective usage of concurrency-aware analysis techniques, and the exploitation of the concurrency model during the system refinement (e.g., during its allocation on a specific platform).

In this section, we introduce a *concurrency-aware executable metamodeling approach* (Section 2.4.1), which supports a modular definition of the execution semantics (package *Semantics* in the xDSML pattern introduced in the previous section), including the concurrency model, the semantic rules, and a well-defined and expressive communication protocol between them. The protocol supports both *the mapping* of the concurrency model to the semantic rules, and *the feedback*, possibly with data, from the semantic rules to the concurrency model.

The concurrent executable metamodeling approach also comes with a dedicated meta-language to define the concurrency model and the protocol (Section 2.4.2), and an execution environment to simulate and analyze behavioral models.

2.4.1 Reifying Concurrency in xDSMLs

2.4.1.1 Background Knowledge

Current metamodeling environments support defining a modeling language through the specification of the concrete and the abstract syntaxes as well as the mapping from the syntactic domain to the semantic domain. Over the last 50 years, the language theory community has studied the mapping between the syntactic domain and the semantic domain extensively. This has led

to three primary ways of defining semantics: *operational semantics*, where a virtual machine uses guard(s) on the execution state to drive the evolution of the models expressed in the language [159, 104, 12, 111]; *axiomatic semantics*, where predicates on the execution state allow reasoning about the models expressed in the language and its correct evolution [92, 87, 208]; and *translational semantics* [80] that defines an exogenous transformation from the syntactic domain to an existing language (either an existing computer language or a mathematical denotation, *i.e.,* a denotational semantics [168]). A drawback of such approaches is that none of them supports the specification of concurrency in a manner that would allow systematic reasoning (chapter 14 of [208]). Even if these approaches could support the definition of concurrency, the concurrency model would be scattered through the semantic specification, making it difficult to understand and analyze the properties related to concurrency (*e.g.*, deadlock freeness, determinism).

In most language implementations, the concurrency semantics is implicitly embedded in the underlying execution environment used to execute the conforming models. For instance, some executable models supporting concurrent execution rely on the Java concurrent model. On one hand, the concurrency of the model depends on the Java concurrency and on the other hand it does not guarantee similar execution/analysis on platforms with different parallelism possibilities (*e.g.*, single core vs. many cores, processor arrays).

Work on formal and explicit models of concurrency has been the focus of some research programs since the fifties. Early work in this area resulted in three well-known contemporary approaches: CCS [144], CSP [93] and Petri Nets [158]. Unlike the approaches from language theory, these solutions focus on concurrency, synchronizations and the, possibly timed, causalities between actions. In these approaches, the focus is on concurrency and, thus, the actions are opaque and abstract away details on data manipulations and sequential control aspects of the system. Such models have proven useful for reasoning about concurrent behavior, but they are not tailored to support the description of a *domain-specific* modeling language dedicated to a domain expert. After many years, work on models of concurrency has consolidated, from an analytical point of view, into two different approaches, namely, event structures [207] and tagged structures [125]. In these approaches the non-relevant parts of a model are abstracted away into *events* (also named signal) and the focus is on how such events are related to each other through causality, timed or synchronization relationships. Both event structures and tagged structures have been used to formally specify or compare concurrency models underlying system models expressed in modeling languages. These concurrency models and can be viewed as the concurrent specification of a specific system model. However, such approaches are not related to the computational part of a model and have not been used to specify the concurrency semantics of a language.

2.4.1.2 Language Units Identification

Taking a step back from these seminal approaches, we explicitly identify the common language units that constitute the design and implementation of an executable concurrency-aware modeling language (see middle level of Fig. 2.6). Each language unit is independent of the way it is implemented, and directly benefits from language and concurrency theories described above.

Language Unit #1 The first language unit is the description of the language *abstract syntax* (see Fig. 2.6). Older approaches build the semantics of the language on top of the concrete syntax but the benefits of using the abstract syntax as a foundation for language reasoning (first introduced in [132]) have been well understood since the 1960s. In the MDE community, the abstract syntax is a first class part of a language definition. The abstract syntax specifies the syntactic domain and is used to anchor the semantics. It is however important to avoid blurring

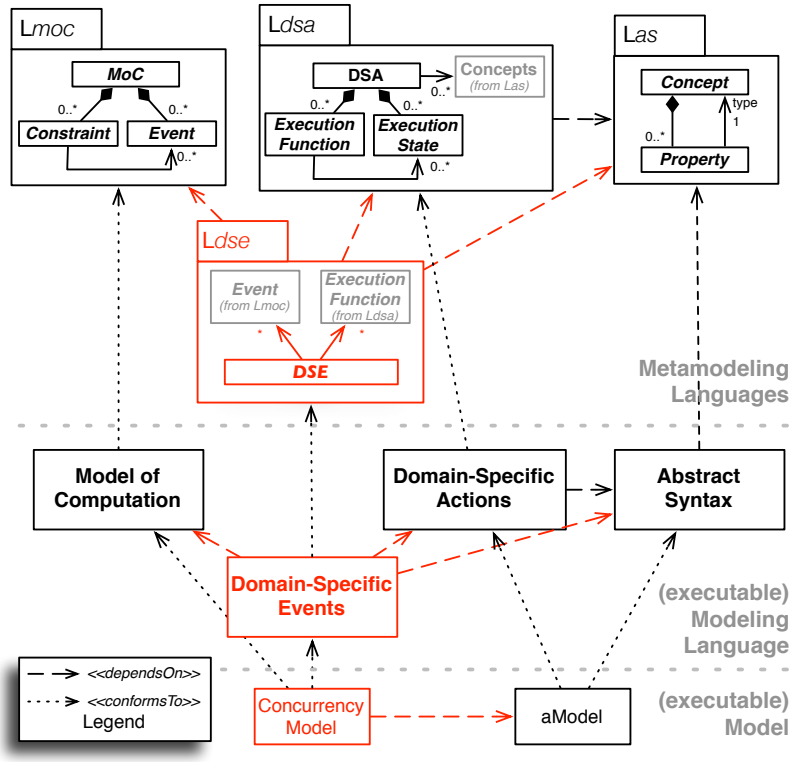


Figure 2.6: Modular design of a concurrency-aware xDSML

the syntactic domain with language elements that represent the execution state of the model (cf. DDMM in the xDSML pattern introduced in Section 2.3).

Definition 1. The Abstract Syntax (AS) specifies the concepts of the language and their relationships. An instance of the AS is a model.

Consequently, a meta-language for modeling AS (*Las* in Fig. 2.6) must provide facilities to define the language concepts (*Concept*) and the relationships between them (*Property*).

Language Unit #2 The second language unit, called *Domain Specific Actions* (DSA, see Fig. 2.6), adds new properties that represent the *execution state* of a model (cf. SDMM in the xDSML pattern introduced in Section 2.3) and a set of *execution functions* that operate on these properties during the execution of a model (cf. package *Semantics* in the xDSML pattern introduced in Section 2.3).

The execution state can be represented, for example, by the *current state* in a Finite State Machine (FSM). It can also be specified independently of the abstract syntax, as in, for example, the incidence matrix that encodes the state of a Petri net. Such information is needed to specify the state of a model during its execution but is not needed to specify the model's static structure. It is consequently part of the semantic domain.

The DSA is also composed of *execution functions* that specify how the execution state sequentially evolves during the model execution. For instance, when a transition is fired in a

FSM, the current state is updated. This is one of the roles of the execution functions. They also specify how the concepts of a language behave. For instance if the language contains a *Plus* concept, then an execution function must specify how the *Plus* instances actually behave during the model execution.

Definition 2. The Domain Specific Actions (DSA) represent both the execution state and the execution functions of a DSML. An instance of the DSA represent the state of a specific model during the execution and the functions to manipulate such a state.

No hypothesis is made on how to specify the DSA (*Ldsa* in Fig. 2.6). However, the specification of the DSA depends on the AS since it describes a part of its semantic domain. The execution state would be defined with structural properties representing the semantic domain, in the same way *Las* supports the definition of the syntactic domain. The execution functions can be specified in very concrete terms (*e.g.*, operational semantics that uses an action language to specify rewriting rules), or in more abstract terms (*e.g.*, denotational semantics that provides functions specifying the execution functions). The latter approach only denotes mathematical properties about the result, and does not specify any details on how to implement the resulting functions. This is even more abstract in an axiomatic semantics, where pre/post conditions on the execution state of the system are specified and all the functions that respect such conditions are considered as correct execution functions.

Note that the global ordering of the execution functions is not specified in the DSA since it can be concurrent (and timed). This is the role of the third language unit.

Language Unit #3 Concurrency theory has proposed many approaches, but roughly speaking a concurrency model is a way to specify how different events are causally and temporally related during an execution (in our case, the execution of a model conforming to a DSML). These ideas have been used in the notion of Model of Computation (MOC) [61, 18, 99]. All definitions of MOCs share the fact that a MOC acts as a director for some pieces of code. The MOC is then acting as an explicit concurrency pattern, which provides MOC-dependent analysis properties. The third language unit is then called *Model of Computation* (see Fig. 2.6) and explicitly specifies the concurrency.

Definition 3. The Model of Computation (MOC) represents the concurrency aspects in a language, including the synchronizations and the, possibly timed, causality relationships between the execution functions. An instance of a MOC is defined for a specific model, conforming to the DSML. It is the part of the *concurrency model* that specifies the possible partial orderings between the events instantiated with regards to the model.

A meta-language for modeling MOC (*Lmoc* in Fig. 2.6) would allow the definition of events and the specification of causal relationships (and synchronizations) such as scheduling, temporal constraints, and communications. The events can be discrete (*i.e.,* a discrete event is a possibly infinite sequence of occurrences), or dense (*i.e.,* a dense event is an infinite set of occurrences and there are an infinity of occurrences between any two event occurrences in the set). *Lmoc* must be independent of a specific AS or DSA.

2.4.1.3 Reifying Language Units Coordination

In our approach, all language units previously presented are specified separately (see middle level of Fig. 2.6). This separation benefits modularity, reuse and the identification of the concurrency related analyses supported by the language. The modeling units must then be consistently coordinate to provide an executable modeling language with reified concurrency.

This coordination has to keep the language units separated while providing a natural articulation between them.

The AS and the DSA are kept separated to support several implementations of the DSA for a single AS (to deal with semantic variation points, or with semantics for different purposes, *e.g.*, interpreter or compiler). There exists a mapping between the DSA and the AS, however the DSA is dedicated to a specific AS (see dependency between AS and DSA in Fig. 2.6), and both AS and DSA are dedicated to the DSML under design. Consequently, we did not reify this mapping. The mapping is more conveniently described directly in the DSA.

The definition of the DSML behavioral semantics then consists in specifying the coordination of a given MOC with the DSA. This coordination must keep the MOC and DSA independent to enable the (re)use of a MOC on different AS/DSA or changing the MOCs on a single AS/DSA. Hence, the coordination specification can be put neither directly in the MOC nor in the DSA. For this reason, we reify the binding as a proper language unit that bridges the gap between the MOC and the DSA. This is done through the notion of *Domain Specific Event*, a novel metamodeling facility that we propose to reify.

Language Unit #4 The *Domain Specific Events* (DSE, see Fig. 2.6) specify the coordination between the events from the MOC and the execution function calls from the DSA. The DSE depend on both the MOC and the DSA. This coordination contains four parts:

DSE → DSA The DSE specify events that are associated with one or more execution functions. When such an event occurs, it results in the call of the associated execution functions. The meta language for modeling DSE (*Ldse* on Fig. 2.6) has to make some choices about how much associated functions can be associated with an event (*e.g.*, single one, any) and if several functions are associated with a single event, it must specify how these calls must be done (*e.g.*, in sequence, in parallel).

MOC → DSE The MOC events can be specified at an abstraction level different than the execution functions from the DSA. For this reason, the DSE specify how the defined events are obtained from the ones constrained by the MOC. This specification can be, for example, the filtering of occurrences from an event or the detection of an occurrence pattern from various events. It can also be the observation of some dense events from the MOC. In this case the DSE are used to specify the relevant observations on the dense event from the MOC and, in such a way, they specify the events that can be observed by looking at the execution of the conforming models. Such an adaptation between the low level events from the MOC and the ones in the DSE can be arbitrarily complex (ranging from a simple mapping to a complex event processing). However, when *Ldse* allows adaptations more complex than a simple mapping, one must ensure that the adaptation is not breaking any concurrency-related assumptions from the MOC.

DSA → DSE The MOC and the DSE represent the specification, at the language level of the concurrency model (dedicated to a specific model conforming to the DSML). This concurrency model specifies the acceptable partial orderings of both the events constrained by the MOC and the ones from the DSE. During a specific execution, the call to some execution functions can restrict such partial orderings. For instance, if the DSML specifies a conditional concept (*e.g.*, *if-then-else*), a MOC usually specifies that going through the *then* branch or through the *else* branch depends on the evaluation of the condition (*i.e.,* the condition evaluation causes either the *then* or the *else* branch, exclusively). Both paths are specified in the concurrency model as acceptable but the actual path taken during an execution depends on the result of the call to an execution function. The specification of

the feedback from the execution function calls to the execution engine of the concurrency model must be specified in the DSE.

MOC ← DSE → AS Finally , the DSE must specify how the MOC is applied on a specific model that conforms to the DSML (*i.e.,* how to create the concurrency model according to the MOC constraints and the AS concepts). Depending on the language used for the MOC modeling, this specification can be of a different nature, however it requires the capacity to query the AS to retrieve the parameters needed for the creation of the concurrency model. For instance, in a FSM the DSE can specify that a specific constraint must be instantiated for all the *Transition* instances in the model. Also, it can retrieve the actual parameter of the constraint by querying the AS. Once again, depending on the possibility offered by *Ldse*, one must ensure the preservation of the MOC assumptions (*e.g.*, by using proven compilers or a language supporting clear and simple composition of constraints from the MOC).

Definition 4. The Domain Specific Events (DSE) represent a coordination between the MOC and the DSA to establish the concurrency-aware semantic domain. It is composed of a set of domain specific events, a mapping between these events and the execution functions from the DSA, a possibly complex mapping between the events constrained by the MOC and the domain specific events; the specification of the impact of the execution function results in the execution of the concurrency model and finally the specification of the MOC application on a specific model that conforms to the DSML.

As highlighted by the previous description, the coordination between the MOC and the DSA (*i.e.,* the DSE) is a key point to enable concurrency-aware semantic domain. However, this coordination is often implicit or hard coded. We believe that its reification enables effective use of a language that includes concurrency and computational aspects. In this section, we have identified the key ingredients for designing a concurrency-aware executable DSML that leads to the architectural pattern proposed in Figure 2.6. Consequently, we define a concurrency-aware executable DSML as follows:

Definition 5. A *concurrency-aware executable DSML* is a domain-specific modeling language whose conforming models are executable according to an explicit concurrency model. Its definition includes at least the abstract syntax and the behavioral semantics (including the DSA, the MOC and the DSE to coordinate them). In our approach, a concurrency-aware executable DSML ($xDSML$) is defined as a tuple $\langle \text{AS}, \text{DSA}, \text{MOC}, \text{DSE} \rangle$.

2.4.2 MoCCML: a Meta-Language for the Concurrency Concern in xDSMLs

MoCCML is a meta-language supporting the aforementioned pattern. It supports the definition of models of computation (incl. the concurrency and the communication, MoCC), and the protocol with the DSA through the definition of the DSE.

The meta-language MoCCML tends to crystallize the best practices from the concurrency theory and the model-driven engineering. It leverages experiences on the explicit definition of the valid scheduling of an application through a clock constraint language [129] and an automata-based language [57]. It also reifies the appropriate concepts to enable automated reasoning.

2.4.2.1 MoCCML Overview

MoCCML is a declarative meta-language specifying constraints between the events of a MoCC. At any moment during a run, an event that does not violate the constraints can occur. The

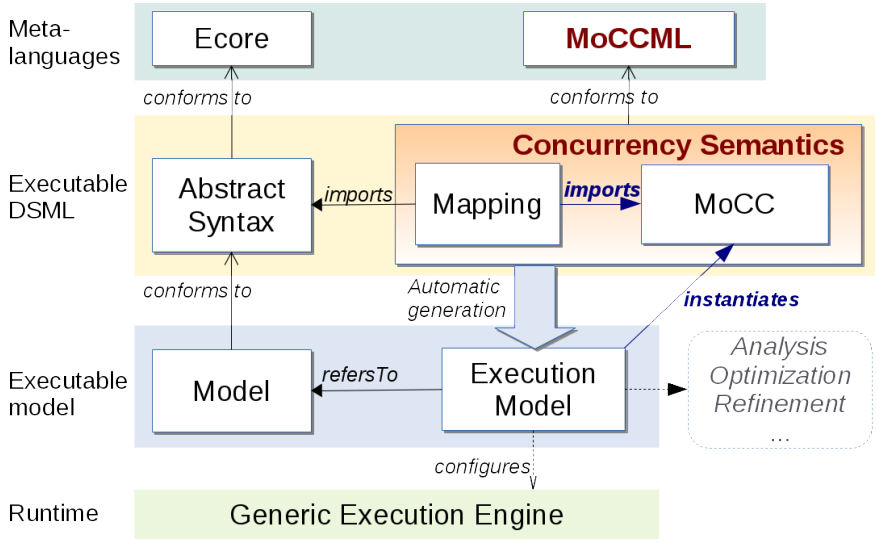


Figure 2.7: Big picture of MoCCML

constraints are grouped in libraries that specify MoCC specific constraints (named MoCC on Figure 2.7 and conforming to MoCCML). These constraints can also be of a different kinds, for instance to express a deadline, a minimal throughput or a hardware deployment. They are eventually instantiated to define the execution model of a specific model (see Figure 2.7). The execution model is a symbolic representation of all the acceptable schedules for a particular model.

To enable the automatic generation of the execution model, the MoCC is weaved into the context of specific concepts from the abstract syntax of a DSML. This contextualization is defined by a mapping between the elements of the abstract syntax and the constraints of the MoCC (achieved by the box named *Mapping* in Figure 2.7). The mapping defined in MoCCML is based on the notion of event (DSE), inspired by ECL [54], an extension of the Object Constraint Language [153]. The separation of the mapping from the MoCC makes the MoCC independent of the DSML so that it can be reused, and semantic variation points can be managed for a single abstract syntax. From such description, for any instance of the abstract syntax it is possible to automatically generate a dedicated execution model (see "executable model" in Figure 2.7).

In our approach, this execution model is acting as the configuration of a generic execution engine (see "generic execution engine" in Figure 2.7), which can be used for simulation or analysis of any model conforming to the abstract syntax of the DSML.

MoCCML is defined by a metamodel (*i.e.,*the abstract syntax) associated to a formal Structural Operational Semantics [159]. MoCCML comes with a model editor combining textual and graphical notations, as well as analysis tools based on the formal semantics for simulation and exhaustive exploration.

In the remainder of this section, we present the concepts of MoCCML (*i.e.,*the MoCCML metamodel), its concrete syntax and the semantics behind these concepts.

2.4.2.2 MoCCML Syntax

Abstract Syntax MoCCML is based on the principle of defining constraints on events. In the abstract syntax, there are two categories of constraint definitions: the *Declarative Definitions* and the *Constraint Automata Definitions* (see Figure 2.8). Each constraint definition has an

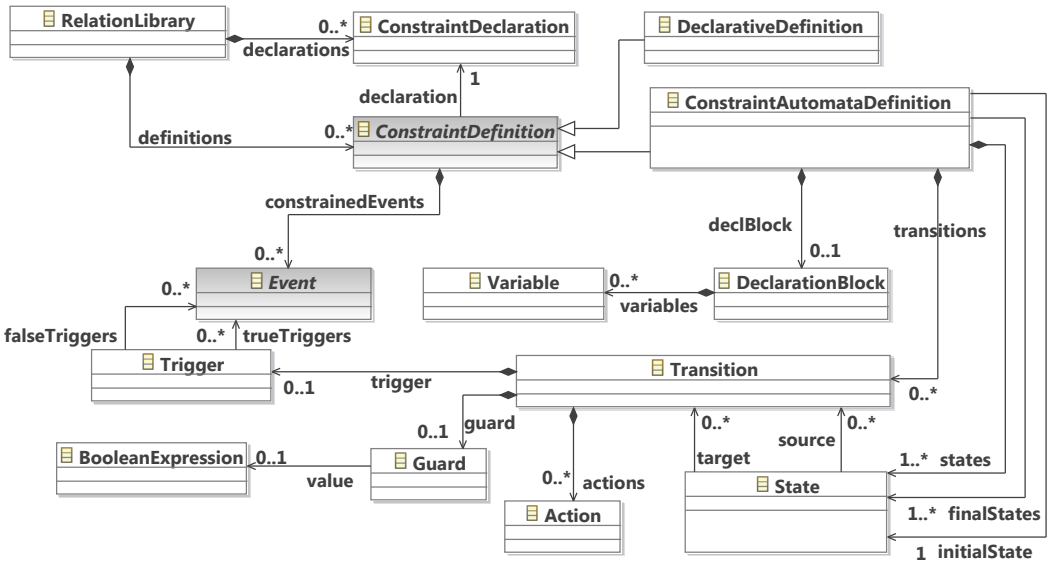


Figure 2.8: Excerpt of the MoCCML metamodel

associated *ConstraintDeclaration* that defines the prototype of the constraint. These definitions constrain some *Event*s.

A declarative definition is defined as a set of constraint instances. For more details, we refer the reader to [52] that described the declarative part inspired from the CCSL language.

As illustrated in Figure 2.8, a *Constraint Automata Definition* contains a set of *States* with a single initial state and one or more final states. It also contains *DeclarationBlocks* where local *Variable*s can be declared. To ease exhaustive simulations we restricted the types of the variables (and parameters to be *Event* or *Integer*).

The constraint automata definition introduces the concept of *Transition* which links a *source* state and a *target* state. It contains a *Trigger* that defines two sets of events (namely *trueTriggers* and *falseTriggers*). The transition is fired if the events in the *trueTriggers* set are present and the ones in the *falseTriggers* set are absent. A transition can define a *Guard*. A guard is a boolean expression over the local variables or the parameters of the definition. Finally, during the firing of a transition, actions such as integer assignments (possibly with a value resulting from an expression such as the increment of a counter) can operate on the local variables.

Concrete Syntax The concrete syntax of MoCCML is implemented as a combination of graphical and textual syntaxes to provide the most appropriate representation for each part of a MoCC conforming to the aforementioned abstract syntax.

The graphical model shown in Figure 2.9 defines a MoCC Constraint Library (*SimpleSDFRelationLibrary*), which contains a constraint declaration named *PlaceConstraint*. The constraint declaration is associated to a constraint automata definition (*PlaceConstraintDef*). In this library, we define a constraint between the *read* and *write* events. The automaton operates on 5 integer parameters (one variable: *size* ; and 4 constants: *itsCapacity*, *itsDelay*, *pushRate*, *popRate*), which are set during the instantiation process.

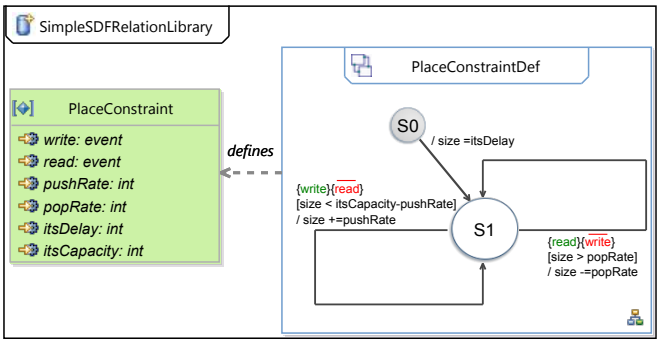


Figure 2.9: Screenshot of the MoCCML graphical editor

2.4.2.3 MoCCML Semantics

This section presents an overview of the operational semantics of MoCCML, which enables the effective construction of the acceptable schedules. The interested reader can refer to [52] for a full definition of the operational semantics. An execution model consists in a finite set of discrete events, constrained by a set of constraints. A *schedule* σ over a set of events E is a possibly infinite sequence of *Steps*, where a step is a set of occurring events. $\sigma : \mathbb{N} \rightarrow 2^E$. For each step, one or several event(s) can occur. The goal of the semantic rules is to specify how to construct the acceptable schedules.

The semantics of a specification expressed in MoCCML is given as a Boolean expression on $\mathscr{E}$, where $\mathscr{E}$ is a set of Boolean variables in bijection with E. For any $e \in \mathscr{E}$, if e is valued to *true* then the corresponding event occurs; if it evaluates to *false* then it does not occur. If no constraints are defined, each boolean variable can be either true or false and there are 2^n possible futures for all steps, where n is the number of events. Consequently, in this case the number of acceptable schedules is infinite.

Each time a constraint is added to the specification, it adds boolean constraints on $\mathscr{E}$. The boolean constraints depend on the definition of the MoCCML constraint and its internal state. When several MoCCML constraints are defined, their boolean expressions are put in conjunction so that each added constraint reduces the set of acceptable schedules. For instance, if the *sub-event* declarative constraint is defined between two events e1 and e2 (*i.e.*,e1 *sub-event* of e2), then the corresponding boolean expression is $\mathtt{e1} \Rightarrow \mathtt{e2}$.

The same principle applies to the constraint automata definitions. The boolean expression associated to a specific constraint automata is obtained according to: 1) the value of the automaton's local variables; 2) the current state; 3) the evaluation of boolean guards on the output transition of the current state and 4) the triggers ($trueTriggers$ and $falseTriggers$) on the output transitions of the current state.

The semantics of a constraint automaton is defined as a *logical disjunction* of the boolean expressions associated to the output transitions of the current state. For a transition t, if its guard is valued to true, the resulting boolean expression is the conjunction of all the events in the $trueTrigger$ set in conjunction with the conjunction of the negation of all the events in the $falseTrigger$ set. For instance, in the constraint automaton depicted in Figure 2.9 which represents a simple data-flow protocol, the boolean expression when $size$ is less than $itsCapacity$ minus $pushRate$ is: $write \wedge \neg read$. In the case where $size$ is also greater than $popRate$ the automata semantics is $(write \wedge \neg read) \vee (read \wedge \neg write)$. If the new computed step is such that the boolean equation of one transition is valued to true, then the transition is fired, meaning that the current state evolves to the target of the fired transition and the actions of this transition

are executed.

2.4.3 Model Execution and Analysis

As shown in Fig. 2.7, the MoCC included in the execution semantics of an xDSML is later used to generate automatically the execution model of a particular model conforming to the xDSML. This execution model is used by a generic execution engine either to simulate the system or to explore exhaustively all acceptable schedules [53]. The exploration of all schedules can be captured explicitly in a state space graph.

Any change or variation in the MoCC creates a new execution model that can be used in the generic execution engine without any other modification. This is quite different from existing approaches that usually hide (a part of) the semantics in a dedicated framework. Additionally, in our approach the execution model is an explicit entity that can be manipulated for reasoning. Such manipulations may include:

- the verification of temporal logic properties (safety and liveness) [160], on the state space graph structure;
- the mining of the graph to extract a schedule that optimizes specific objectives (like the extraction of minimal buffer requirements done in [84], but in our case, directly applied at the DSL level, without requiring the transformation towards another formalism);
- the extraction of the system properties by static analysis of an event-graph representation of the execution model, such as in [156].

In addition to capturing the concurrency constraints within complex software-intensive systems, the MoCC of a particular xDSML can be refined later on to handle platform constraints. This allows system engineers to simulate and analyze particular design models with regard to the deployment on such a particular platforms. MoCCML offering a declarative formalism, the initial MoCC of the language can be modularly completed with additional constraints related to specific platforms.

2.5 Execution Trace Management and Omniscient Debugging

The content of this section is an adapted excerpt from the following publications:

Erwan Bousse, Jonathan Corley, Benoit Combemale, Jeff Gray, and Benoit Baudry. Supporting Efficient and Advanced Omniscient Debugging for xDSMLs. In *8th International Conference on Software Language Engineering (SLE 2015)*, Pittsburg, USA, October 2015. ACM. [22]

Erwan Bousse, Tanja Mayerhofer, Benoit Combemale, and Benoit Baudry. A Generative Approach to Define Rich Domain-Specific Trace Metamodels. In *11th European Conference on Modelling Foundations and Applications (ECMFA 2015)*, LNCS, L'Aquila, Italy, July 2015. Springer Verlag. [23]

Erwan Bousse, Benoit Combemale, and Benoit Baudry. Scalable Armies of Model Clones through Data Sharing. In *17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014)*, Valencia, Spain, 2014. Springer. [20]

While an executable model by itself inherently expresses an *intended* behavior, dynamic V&V techniques need an *explicit* representation of behavior over time. A most common representation

of a model's behavior is the *execution trace*, which is a sequence containing all the relevant information about an execution over time. Such information may include the *execution states* reached during the execution, the *execution steps* that were responsible for these state changes, and the *stimuli* originating from the execution environment and the system.

All previously mentioned V&V approaches rely on execution traces: omniscient debugging relies on an execution trace to revisit a previous execution state; semantic differencing consists in comparing execution traces of two models in order to understand the semantic variations between them; runtime verification consists in checking whether or not an execution trace satisfies a property. In addition, execution traces are at the core of *behavioral equivalence checking* of xDSMLs, such as bisimulation [145], and can be used as evidence [59] shared among different combined V&V approaches [19],

With that in mind, it transpires that providing *execution trace management facilities* is an essential requirement to support dynamic V&V for xDSMLs. Such facilities include *acquiring*, *processing* and *visualizing* execution traces that result both from testing and deploying executable models. However, these facilities have an important prerequisite to satisfy: the definition of a *data structure* to define the content and the layout of the execution traces of an xDSML. Yet, this undertaking is not trivial for at least two reasons. First, the execution semantics of an xDSML can be arbitrarily complex, both regarding the definition of the execution state and the definition of the model transformation that changes it. As a result, structuring and adapting this information into an execution trace data structure is difficult. Second, execution traces tend to be very large artifacts: a short execution of a simple Java program of 20 classes and 3,000 lines of code can lead to 150 000 method calls to store in an execution trace [44]. Consequently, a data structure must be adapted for an efficient representation and processing of large traces.

All in all, providing execution trace management facilities can be summarized as three main inter-related challenges [21]:

- The *usability* of an execution trace data structure must be ensured to cope with the complexity of data. More precisely, both *generic manipulations* (*e.g.*, comparing the number of different states or the amount of steps) and *domain-specific manipulations* (*e.g.*, determining how many tokens traversed a Petri net place) must be taken into account.

- Since executing even a simple model or program can lead to very large execution traces, *scalability in memory* of executions traces must be taken into account. Indeed, while database solutions for storing models[6] (*e.g.*, execution traces) are more and more efficient, loading models directly in memory remains more efficient for large models and heavyweight manipulations [13].

- Finally, also because of their large size, *scalability in manipulation time* of execution traces are of primary importance, and imply the need for efficient ways to browse a trace.

To tackle these challenges, we investigate two different complementary directions. First, we focus on the representation of the *execution state* of an executed model in the context of *clone-based* execution traces. An execution trace containing all the states reached by an executed model can be obtained in a generic way by *cloning* the model after each execution step. Such an approach brings advantages regarding *usability*, since the execution trace data structure is simple and appropriate for generic execution trace manipulations. Moreover, existing model transformations and queries specific to the xDSML can directly be applied on execution states stored in a clone-based execution trace. Yet, at runtime, a model is represented by a set of elements stored in memory called the *runtime representation* of the model. Cloning is

[6] Cf. https://www.eclipse.org/cdo

usually done by duplicating the complete runtime representation of a model, hence requiring an important amount of memory, compromising the need for *scalability in memory*. To cope with this problem, we propose a scalable model cloning approach [20] to create large amounts of model clones while sparing memory usage. Our approach is based on the observation that manipulations rarely modify a whole model. In the case of model execution, the only modified part is the execution state, which may be scattered in the different parts of the model. Hence, knowing which parts might get modified, our cloning approach determines what can be shared between the runtime representations of a model and its clones. Our generic cloning algorithm is parameterized with three strategies that establish a trade-off between memory savings and the usability of clone manipulations. We propose an implementation of the approach within the Eclipse Modeling Framework (EMF), along with our evaluation of memory footprints and computation overheads with 100 randomly generated models. Results show a positive correlation between the proportion of shareable elements and memory savings, while the worst median overhead is 9,5% when manipulating the clones.

Then, we focus on the *structure* of execution traces that contain information about both states and steps. While clone-based execution traces show some benefits, they are necessarily relying on a *generic* data structure based on a *unique sequence* of execution states. This has two consequences. First, there a gap between the domain concepts of the xDSML and the generic data structure, which compromises *usability* regarding domain-specific trace manipulations. Second, execution trace manipulations that focus on a specific part of the execution state has still to browse the complete trace even if this part changed a small number of times, hence compromising *scalability in time*. To cope with these problems, we propose a generative approach to define multidimensional and domain-specific execution trace metamodels [23]. A *metamodel* is a data structure defined by an object-oriented model defining a particular domain. To enhance usability, our first idea is to go from generic trace metamodels to a *generic meta-approach* to define *domain-specific execution trace metamodels*. This is accomplished by knowing which parts of an xDSML is required by the manipulations, hence using the same principle as the previous contribution. Then, to enhance scalability in manipulation time, our second idea is to create *multidimensional* trace metamodels, *i.e.,*metamodels that provide many navigation paths to explore a trace.

We applied our trace management facilities in the field of interactive debugging, whose concern is to control (*i.e.,*pause or unpause) and observe an execution in order to find the cause of some unintended behavior. While regular interactive debugging only allows to go *forward* in an execution, *omniscient debugging* is a promising technique that relies on execution traces to enable free traversal of the reached states, which includes going *backward* in the execution. While some General-Purpose Languages (GPLs) already have support for omniscient debugging, developing such a complex tool for any executable Domain-Specific Modeling Language (xDSML) remains a challenging and error prone task A solution to this problem is to define a generic omniscient debugger for all xDSMLs. However, generically supporting any xDSML both compromises the efficiency and the usability of such an approach. To address these problems, we propose an advanced and efficient omniscient debugging approach for xDSMLs [22]. Our contribution consists in a partly generic omniscient debugger supported by generated domain-specific trace management facilities. These facilities include a multidimensional domain-specific execution trace metamodel, obtained using our second contribution. Being domain-specific, these facilities are tuned to the considered xDSML for better efficiency. Usability is strengthened by providing multidimensional omniscient debugging, which is achieved using our generated execution trace metamodel. Results show that our approach is on average 3.0 times more efficient in memory and 5.03 more efficient in time when compared to a generic solution that clones the model at each step.

2.6 Wrap-up: Design and Implementation of the UML Activity Diagram Language

The content of this section is an adapted excerpt from the following publication:

Benoit Combemale, Julien Deantoni, Olivier Barais, Arnaud Blouin, Erwan Bousse, Cédric Brun, Thomas Degueule, and Didier Vojtisek. A Solution to the TTC'15 Model Execution Case Using the GEMOC Studio. In *8th Transformation Tool Contest (TTC 2015)*, L'Aquila, Italy, 2015. [36]

As a wrap-up of the approach introduced in this chapter, we present in this section the implementation of an activity diagram language inspired from fUML [155]. This case study has been proposed as one of the case (the *Model Execution* case) at the Transformation Tool Contest (TTC) 2015[7] [131]. The implementation proposed in this section corresponds to the most complete variant of this case (*i.e.,* variant 3), whose metamodel includes various kinds of nodes (initial node, final node, fork node, join node, decision node, merge node), opaque actions, boolean and integer variables (either local or as input), and various boolean and integer expression types. This solution has been awarded as the overall winner of the Model Execution case at TTC 2015 [36].

The solution has been implemented using the various tools and methods presented in this chapter and integrated in the GEMOC studio[8]. The solution provides not only an EMF-based interpreter for UML activity diagrams, but also comes with a well integrated model debugging environment based on Eclipse, including advanced features for graphical model animation and execution trace management. We also propose an enhanced version of our solution which integrates into the operational semantics a formal and explicit model of concurrency supported by analysis tools. We evaluate our solution regarding both the benchmark provided into the case, and the criteria proposed into the case description. In particular, we give evidence for the correctness, the understandability and conciseness, and the performance of our solution.

In the rest of this section we first present an overview of the solution using the GEMOC language workbench to design and implement the Activity Diagram language, as well as the resulting environment in the GEMOC modeling workbench (Section 2.6.1). Then, we present step-by-step the implementation of the solution, including the operational semantics (Section 2.6.2), the language assembling (Section 2.6.3) to be used for trace management (Section 2.6.4) and model debugging (Section 2.6.5), and a variant which include a formal and explicit concurrency model (Section 2.6.6). Finally, Section 2.6.7 gives the evaluation results of our implementation with regard to the evaluation criteria provided into the case description. The evaluation provides evidence of the correctness, understandability, conciseness and performance of the solution.

2.6.1 Overview of the solution

Our solution uses the GEMOC Studio, an eclipse package atop the *Eclipse Modeling Framework* (EMF)[9], which includes both a language workbench to design and implement tool-supported xDSMLs, and a modeling workbench where the xDSMLs are automatically deployed to allow system designers to edit, execute, simulate, and animate their models. As a result, our solution not only provides a model interpreter conforming to the operational semantics of the UML Activity Diagram language, but also provides a graphical model animator, an advanced trace

[7] Cf. http://www.transformation-tool-contest.eu
[8] Cf. http://gemoc.org/studio
[9] Cf. https://www.eclipse.org/modeling/emf

manager, as well as an alternative version that offers an explicit and formal model of computation supporting concurrency. All materials are available from http://gemoc.org/ttc15.

To design and implement the various concerns of an xDSML, the language workbench put together the following tools, all seamlessly integrated to EMF:

- *Kermeta* (Section 2.1), which offers specific annotations for Xtend[10] to support the modular implementation of an operational semantics (both runtime concepts and steps of computation) and its weaving into an EMF-based metamodel (*i.e.,* an Ecore model).
- *Melange*[11], to build the overall language runtime seamlessly integrated to EMF and to ensure interoperability between the legacy metamodel without the operational semantics, and the metamodel extended with the operational semantics.
- *Sirius Animator*, an extension of the model editor designer Sirius[12] to create graphical animators for xDSMLs.
- *MoCCML* (Section 2.4), a tool-supported meta-language to specify a Model of Concurrency and Communication (MoCC) and its mapping to a specific metamodel and associated operational semantics of a xDSML.

The language workbench also includes a generative approach that provides a rich and efficient domain-specific trace metamodel for any xDSMLs, and facilities for omniscient debugging of conforming models (Section 2.5).

Once an xDSML is implemented with the aforementioned tools of the language workbench, the xDSML is automatically deployed into the modeling workbench, which provides to system engineers an advanced environment integrated to the Eclipse debugger for model execution[13]. In particular, the modeling workbench provides the following tools:

- A Java-based *execution engine* (parameterized with the specification of the operational semantics), possibly coupled with *TimeSquare*[14] (parameterized with the MoCC), to support the concurrent execution and analysis of any conforming models.
- A *model animator* parameterized by the graphical representation defined with Sirius Animator to animate executable models.
- A generic *trace manager*, which allows a system designer to visualize, save, replay, and explore different execution traces of their models, as well as navigating step-by-step in a given execution trace (incl., breakpoint, step forward and step backward).
- A generic *event manager*, which provides a user interface for injecting external stimuli in the form of events during the simulation (*e.g.,* to simulate the environment).

The implementation of the Activity Diagram language is automatically deployed in the GEMOC modeling workbench (see Figure 2.10). The modeling workbench offers a powerful environment to system engineers for controlling the execution of their models with a debugger-like control panel[15] (①), visualizing the execution of their models thanks to the graphical animator

[10] Cf. https://eclipse.org/xtend
[11] Cf. http://melange-lang.org
[12] Cf. https://eclipse.org/sirius
[13] See talks at EclipseCon'15, France and Europe: http://siriuslab.github.io/talks/BreatheLifeInYourDesigner/slides/
[14] Cf. http://timesquare.inria.fr
[15] Note that when using MoCCML, the concurrent computational steps are indicated in the control panel (the computational steps that will be executed concurrently during a given execution step). If the MoCC is non deterministic, the control panel proposes the different permitted execution steps, one of which can be selected either manually by the system engineer, or automatically thanks to one of the proposed heuristic.

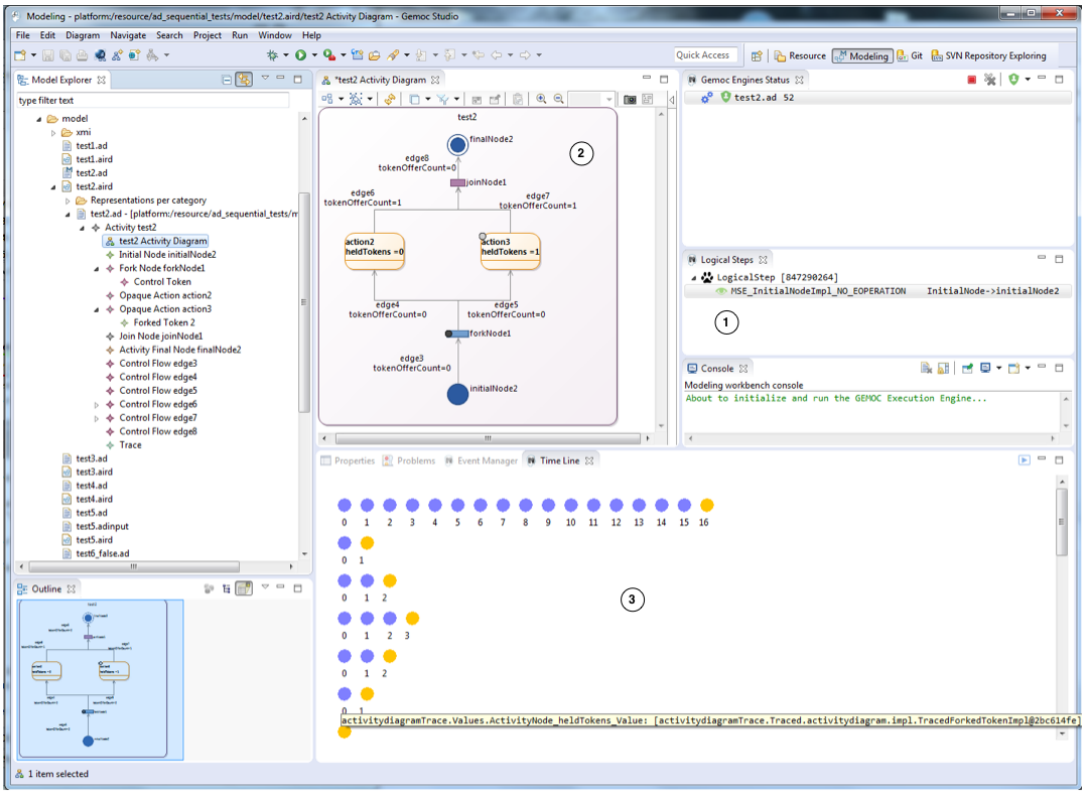


Figure 2.10: Resulting modeling workbench for UML Activity Diagram

(②), and analyzing and exploring several execution traces with a graphical timeline that supports step forward and step backward (③). Finally, the modeling workbench offers several extension points that can be used to plug in additional front-ends or back-ends, such as a timing diagram included in the modeling workbench.

In the rest of this section we provide a step-by-step description of the solution.

2.6.2 Operational Semantics

Kermeta is used to implement the operational semantics. Kermeta complements Xtend to support the definition of both the runtime concepts and the steps of computation in a separate file than the initial metamodel (reused as is), and to statically weave them in the initial metamodel. Kermeta provides static typing to safely define the operational semantics, and a compilation scheme of the operational semantics which results in a Java-based runtime seamlessly integrated to the Java code generated by EMF from the initial metamodel (i.e., the DDMM captured in the abstract syntax).

The runtime concepts of the SDMM can be additional classes that will be merged with the initial metamodel (DDMM), or new structural features (attributes or references) either in the existing classes of the initial metamodel or in the new added classes. When new structural features have to be added to a class existing in the initial metamodel, the annotation @ASPECT is used to re-open the class.

Listing 2.3 shows an excerpt of the modular definition of the runtime concepts. *Token*

and *ForkToken* are new concepts, while the content of *ActivityNodeAspect*, a collection of *Token*, will be merged into the concept *ActivityNode* from the abstract syntax (cf. annotation @*Aspect*). All the runtime concepts of the activity diagram language (see description of the case [131]) have been defined similarly.

```
1  @Aspect(className=ActivityNode)
2  class ActivityNodeAspect {
3    List<Token> heldTokens = new ArrayList<Token>
4  }
5
6  abstract class Token {
7    public ActivityNode holder
8  }
9
10 class ForkedToken extends Token {
11   public Token baseToken ;
12   public Integer remainingOffersCount;
13 }
14
15 [...]
```

Listing 2.3: Modular definition with Kermeta of the runtime concepts

The steps of computation (package *Semantics* in the xDSML pattern) are defined in terms of operations weaved into the suitable classes, either from the initial metamodel or from the new added classes of the runtime concepts. Similarly to the structural features of the runtime concepts, when an operation has to be added to a class existing in the initial metamodel, the annotation @ASPECT is used to re-open the class.

Listing 2.4 shows an excerpt of the definition of the steps of computation, which manipulate the runtime concepts previously defined. The implementation follows the design pattern *Interpreter*[16], defining one operation *execute* per concept of the abstract syntax to be interpreted. Each method is modularly defined into an aspect, and then weaved into the suitable class of the abstract syntax. Listing 2.4 shows the overall execution of an *Activity*. All the steps of computation of the most complete variant of the case have been defined similarly.

```
1  @Aspect(className=Activity)
2  class ActivityAspect {
3
4      def void execute(Context c) {
5          _self.locals.forEach[v|v.init(c)]
6          _self.nodes.filter[node|node instanceof InitialNode].get(0).execute(
               c)
7
8          var list =  _self.nodes.filter[node|node.hasOffers]
9          while (list!=null && list.size>0 ){
10             list.get(0).execute(c)
11             list =  _self.nodes.filter[node|node.hasOffers]
12         }
13     }
14 }
15
16 [...]
```

Listing 2.4: Modular definition with Kermeta of the steps of computation

Keeping the definition of the runtime concepts and the steps of computation in a separate file offers a modular mechanism to implement the operational semantics. In addition to support the separation of concerns (abstract syntax and operational semantics), this is also a way to support different implementations of the operational semantics for the same abstract syntax

[16] Cf. http://en.wikipedia.org/wiki/Interpreter_pattern

(*e.g.,* in case of semantic variation points).

2.6.3 Language Assembling

Once the operational semantics is defined with Kermeta, *Melange* can be used from the language workbench for assembling the initial metamodel and the chosen operational semantics into an xDSML.

```
1 language UMLActivityDiagram {
2   syntax "platform:/resource/.../activitydiagram.ecore"
3   with org.gemoc.ad.sequential.dynamic.*
4   exactType UMLActivityDiagramMT
5 }
```

Listing 2.5: Assembling an xDSML with Melange

As a result, Melange provides the xDSML as well as a structural interface (*aka.* model type [88]) in the form of a new metamodel that can be used to define additional tooling such as model transformations (*e.g.,* trace manager and execution engine) or animators, which use the operation semantics (runtime concepts or steps of computation). In addition to provide the assembling of the expected xDSML as well as the interoperability between the initial metamodel and the metamodel with the operational semantics, Melange also provides other features not required in this solution such as language inheritance and model transformation reuse (cf. Section 3.1.2).

2.6.4 Trace Management

Based on the resulting xDSML, the language workbench includes a generative approach that automatically provides a rich and efficient domain-specific trace metamodel. Instead of relying on complete snapshots of the executed model to construct a trace, this metamodel precisely captures what is the execution state of a model conforming to the xDSML through an efficient object-oriented structure based on the runtime concepts of the xDSML. In addition, the structure provides rich navigation facilities to browse a trace according to various dimensions (*e.g.* the value of a field or the occurrences of an event). For more details we refer the reader to [23].

2.6.5 Animation Facilities

Optionally, *Sirius Animator* can be used to complement the xDSML with a graphical model animator. *Sirius Animator* allows to either extend the graphical representation of an existing model editor defined with Sirius, or to define a separate graphical representation, based on the runtime concepts. This graphical representation is then used to visualize the state of a model during its execution.

In our solution, we defined a new graphical representation (called *viewpoint specification* in Sirius) on top of the provided metamodel for UML activity diagrams, augmented with the runtime concepts to be visualized at runtime.

2.6.6 Explicit and Formal Concurrency Model

Because concurrency is a more and more important concept, one can use *MoCCML* to specify the MoCC. A MoCC specifies in a formal way the, possibly timed, causalities and synchronizations among the steps of computation. Based on *MoCCML*, non determinism and parallelism is clearly and formally identified in the operational semantics and can be varied or refined. Analysis tools are also provided in the GEMOC studio to make analysis of the MoCC.

Listing 2.6 shows an excerpt of the MoCC specification. Lines 1 and 2 define an event in the context of an *ActivityNode* (*i.e.*, for all its instances). For each occurrence of this event the

execute function is called. All these events are constraints by some relations. For instance, in the classical case, the execution of a node is done after its predecessor has been executed (see the *Precedes* relation line 6). In the context of *Activity* appears a kind of loop since the activity can not start if not stop (line 11) and its *start* actually executes the initial node of the activity (line 15), *i.e.*, the starting point of the causality chain written line 6. From such specification, and for a specific model, a symbolic event structure is automatically derived.

```
1  context ActivityNode
2    def : executeIt : Event = self.execute()
3    inv waitControlToExecute:
4      (not self.oclIsKindOf(MergeNode)) implies
5      Relation Precedes(self.incoming.source.executeIt, self.executeIt))
6
7  context Activity
8    def : start : Event = self.initialize()
9    def : finish : Event = self.finish()
10   inv NonReentrant:
11       Relation Alternates(self.start, self.finish)
12
13 context InitialNode
14   inv startedWhenActivityStart:
15       Relation Precedes(self.activity.start, self.executeIt  )
```

Listing 2.6: Excerpt of the explicit and formal model of concurrency for activity diagrams

2.6.7 Evaluation of the solution

In this section, we evaluate our solution by using the evaluation criteria proposed in the TTC'15 *Model Execution* case description [131]. Each criterion is evaluated on three different versions of our solution, all implemented within the GEMOC studio:

- *executionOnly*: interpreter defined with Kermeta only (incl., Section 2.6.2);
- *withAnimationAndTrace*: execution within the GEMOC modeling workbench, with support of the animation and of the trace management (incl., Sections 2.6.2, 2.6.3, 2.6.4 and 2.6.5);
- *withConcurrency*: execution within the GEMOC modeling workbench, with support of the animation, the trace management and the concurrency (incl., Sections 2.6.2, 2.6.3, 2.6.4, 2.6.5 and 2.6.6).

2.6.7.1 Correctness

The correctness of our solution is based on the test suites provided by the case. All the three versions of our solutions provide correct results.

2.6.7.2 Understandability and Conciseness

Kermeta is used to design and implement the operational semantics. Based on Xtend, Kermeta provide a powerful Java-like imperative and statically typed meta-language. The meta-language follows an object-oriented paradigm which make it directly aligned with the object-oriented Ecore metamodel provided by the case.

The implementation of the operational semantics follows the well-known and time-honored *Interpreter* design pattern which supports a modular design of the operational semantics with regard to the initial metamodel which is reused as is and not affected. There is no translation into a third formalism, and this approach easily supports the definition of different variants of the semantics (*e.g.,* interpreter and compiler, different semantic variation points, etc.). Finally,

the use of the open-class and static introduction mechanisms make the design of the operational semantics even simpler than the interpreter pattern, avoiding to duplicate the initial structure into the interpreter. The operations of the operational semantics are directly weaved into the suitable classes of the initial metamodel.

The entire implementation of the operational semantics of the variant 3 of the Model Execution case comprise 441 LOC (version *executionOnly*). This comprises the entire implementation of the interpreter, sufficient for the execution of any conforming models. The other technologies Melange, Sirius Animator and MoCCML are optional, and can be used only to provide the additional features such as model interoperability, trace management, model animation and formal concurrency specification and analysis.

Note also that the analysis tools that provide the modeling workbench are not only useful for the system engineer to analyse the models, but also for the language designer to analyse the language semantics implementation.

2.6.7.3 Performance

We report in Table 2.1 the execution time (without loading and saving times) of the models provided for the performance evaluation (*Test perf 1*, *Test perf 2*, *Test perf 3_ 1* and *Test perf 3_ 2*). The execution time is provided for all the models, according to the three versions of our solution[17]. Performance evaluation has been performed using an Ubuntu VirtualBox image with Java 8 and the GEMOC Studio v1.0 (June, 2015). The virtual machine is running on top of a HP EliteBook 820 with an Intel Core i7 processor and 16Gb of memory.

	t(executionOnly)	t(withAnimationAndTrace)	t(withConcurrency)
Test perf 1	0.29	0.87	226183
Test perf 2	0.33	0.78	$\perp$
Test perf 3_1	0.37	1,01	$\perp$
Test perf 3_2	0.13	0,19	5219

Table 2.1: Execution time (in ms) of the performance tests

[17] In *Test perf 2* and *Test perf 3_ 1*, there are more than 2^{100}) possible inter-leavings. The goal of the concurrent version is to show and made explicit such interleaving but in these cases, it is obviously impossible.

Chapter 3

Metamodeling in the Large: Facing the Multiplication of DSMLs

One of the major consequences of the adoption of MDE and SLE in industrial processes is the multiplication of DSMLs to address the increasing number of application domains of interest, and the growing number of stakeholders involving in each application domain. Based on the mashup of the various heterogeneous DSML concerns addressed in the previous chapter, we extract relevant language interfaces to define structural and behavioral relationships between DSMLs. In particular, we introduce a structural and axiomatic language interface, aka. Model Type, that we use to support language reuse across different domains (Section 3.1), and a behavioral language interface, aka. event structure, that we use for coordinating behavioral heterogeneous models and check global system properties (Secion 3.2).

"A clear challenge, then, is how to integrate multiple DSLs." J. Whittle, J. Hutchinson, and M. Rouncefield, “The State of Practice in Model-Driven Engineering,” IEEE Software, vol. 31, no. 3, 2014, pp. 79–85.

Contents

3.1 Reuse of Modeling Languages: Facing the Multiplication of Application Domains

3.1.1 Model Typing as Structural Language Interface

The content of this subsection is an adapted excerpt from the following publications:

Sun Wuliang, Benoit Combemale, Steven Derrien, and Robert France. Contract-Aware Substitutability of Modeling Languages. In *9th European Conference on Modelling Foundations and Applications (ECMFA 2013)*, LNCS, Montpellier, France, 2013. Springer. [209]

Clément Guy, Benoit Combemale, Steven Derrien, and Jean-Marc Jézéquel. On Model Subtyping. In Antonio Valecillo, editor, *Proceedings of the 8th European Conference on Modelling Foundations and Applications (ECMFA 2012)*, number 7349 in Lecture Notes in Computer Science, pages 400–415. Springer, 2012. [88]

The growing use of Model Driven Engineering (MDE) and the increasing number of modeling languages has led software engineers to define more and more operators to manipulate models. These operators are defined in terms of model transformations expressed at the language level, on the corresponding metamodel. However, new modeling languages are still generally designed and tooled from scratch with few possibilities to reuse structure or model manipulations from existing modeling languages.

To address the need for a more systematic *engineering* of model transformations, various approaches have recently been proposed. These approaches include model transformation reuse [195, 46, 180, 164, 171, 49] and automatic model adaptation [185, 206, 109, 8]. Although these approaches do meet their goals, they remain somewhat disconnected from each other, and lack a unified theory enabling both their combination and comparison. Such a formalization would also help defining the scope of what can be expected (from an engineering point of view) to put model manipulation into action.

In our approach, we tackle the problem from the model substitutability point of view, through model typing. Model typing provides a well-defined theory that considers models as first-class entities, and typed by their respective model types [180]. In addition to the previous work on model typing focusing on the typing relation (*i.e.,*between a model and its model types), we introduce four model subtyping relations. These relations provide model substitutability, that is they enable a model typed by A to be safely used where a model typed by B is expected, A and B being model types.

This work provides a formal reference specification establishing a family of model type systems. These type systems enable many facilities that are well known at the programming language level, ranging from abstraction, reuse and safety to auto-completion.

3.1.1.1 Model Typing

Model Types were introduced by Steel *et al.* [180], as an extension of object typing to provide abstraction from object types and enable model manipulation reuse.

Definition 6. A type of a model is a set of types of objects which may belong to the model, and their relations.

MOF classes are closer to types (interfaces) than to object classes, thus a model type is closely related to a metamodel. The difference between model types and metamodels lies in their respective relations with models. A model has one and only one metamodel to which it conforms. This metamodel contains all the types needed to instantiate objects of the model. Conversely, a model can have several model types which are subsets of the model's metamodel.

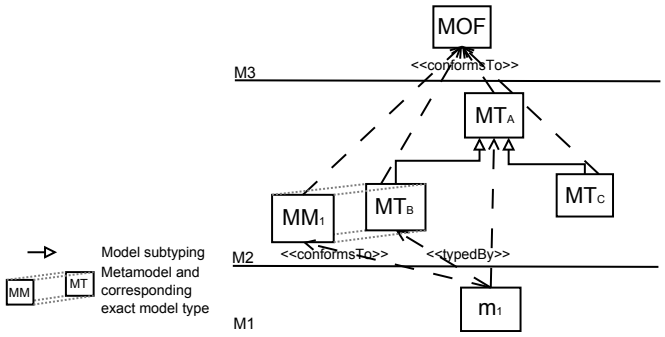


Figure 3.1: Conformance, model typing and model subtyping relations

Because model types and metamodels share the same structure, it is possible to extract the type of a model from its metamodel (we call the model type containing all the types from a model's metamodel the *exact type* of the model). Figure 3.1 represents a model m_1 which conforms to a metamodel MM_1 and is typed by model types MT_A and MT_B, MT_B being the *exact type* of m_1 extracted from MM_1. Both metamodels and model types conforms themselves to MOF.

MOF delegates the definitions of contracts (*e.g.*, pre and post-conditions or invariants) to other languages (*e.g.*, OCL, the Object Constraint Language [153]). Hence model typing does not take into account the contracts by default, but can be additionally taken into account in the subtyping relations (see Section 3.1.1.4).

Substitutability is the ability to safely use an object of type A where an object of type B is expected. Substitutability is supported through subtyping in object-oriented languages. However, object subtyping does not handle type group specialization (*i.e.,* the possibility to specialize relations between several objects and thus groups of types).[1] Such type group specialization have been explored by Kühne in the context of MDE [119]. Kühne defines three model specialization relations (specification import, conceptual containment and subtyping) implying different level of compatibility. We are only interested here in the third one, subtyping, which requires an *uncompromised mutator forward-compatibility*, *e.g.*, substitutability, between instances of model types.

Model Type Matching is a model subtyping relation proposed by Steel *et al.* to enable safe model manipulation reuse in spite of limits of object subtyping. To this end, they use the *object type matching* relation defined by Bruce *et al.* [24], which is more flexible than subtyping. For more details, we refer the reader to Steel's PhD thesis [179].

Definition 7. Model Type MT_B matches model type MT_A if for each object type C in MT_A there is a corresponding object type with the same name in MT_B such that every property and operation in MT_A.C also occurs in MT_B.C with exactly the same signature as in MT_A.C.

Limits of Model Type Matching However, *model type matching* as presented by Steel *et al.* is subject to some shortcomings. First, the type rules they present, and their implementation in Kermeta[2], violate their definition of *type matching* by permitting the relaxation of lower

[1] We refer the reader interested in the type group specialization problem to the Ernst's paper [67].
[2] Cf. http://www.kermeta.org

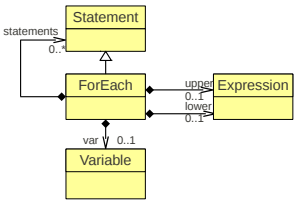


Figure 3.2: `foreach` loop in ORCC IR

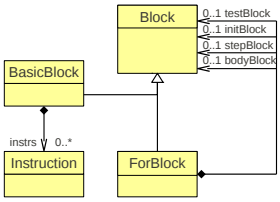


Figure 3.3: `for` loop in GeCoS IR

Figure 3.4: Extracts of ORCC IR and GeCoS IR metamodels

multiplicities, i.e. by allowing a non-mandatory attribute to be matched by a mandatory one, which could potentially lead to an invalid model.

In addition, and more significantly, finding a model type common to several model types is not always possible, even if they share numerous concepts. This impossibility is due to structural heterogeneities between the metamodels [205]. For example, Figure 3.4 presents the metamodels of two intermediate representations which propose similar concepts but with structural heterogeneities: metamodel in Figure 3.2) is a simpler loop than the one in the metamodel in Figure 3.3, iterating only by steps of one on a given variable between bounds, where a `for` can have complete code blocks as initialization, step and test.

Thus to reuse a model manipulation, a subtyping mechanism should provide facilities for the definition of an adaptation, needed to *bind* different structures to a single one on which the manipulation is defined. In our example, such an adaptation could be the transformation of `foreach` loops into more generic `for` loops, using the variable and the lower bound to produce an initialization block, the variable and the upper bound to produce a test block, and automatically producing a step block with a step of one.

This adaptation should be able to adapt back the result of the manipulation, because this manipulation could modify the model it processes or return a result containing elements of the model. For example, a program transformation on an intermediate representation for dead code elimination (DCE) modifies the representation of the program by removing code. Once the optimization has been processed on a common representation, it should be possible to adapt back the structure to apply the result in the original structure.

Although defining common model manipulations on a minimal dedicated metamodel seems to best fit the need for modularity and reuse, we need to consider the presence of legacy model transformations. For example, DCE is already implemented for the GeCoS IR (Figure 3.3)[3]. Reusing this implementation on ORCC IR[4] would avoid the creation of a generic model type and the reimplementation of the optimization. However, the GeCoS IR does not contain only the concepts required for DCE. More particularly, it contains concepts which do not exist in ORCC IR (*e.g.*, pointers). Therefore, a model subtyping mechanism should be able to accept a subtype which only possesses the concepts of the supertype required for the reuse of a specific model manipulation.

[3]GeCoS is a retargetable C compiler infrastructure targeted at embedded processors. Cf. http://gecos.gforge.inria.fr

[4]ORCC is a compiler for CAL, a dataflow actor language in which actions are described with a standard imperative semantics, Cf. http://orcc.sourceforge.net/

3.1.1.2 Model Subtyping Relations

Object-oriented type systems provide important systematic engineering facilities, including abstraction, reuse and safety. We strongly believe that these facilities can also be provided for model manipulation through a model-oriented type system. However, the existing model subtyping relation has shown some limitations.

For this reason, in this section we review four subtyping relations between model types, based on two criteria: the presence of heterogeneities between the two model types and the considered subset of the model types. Such a model subtyping relation is pictured in Figure 3.1 by the generalization arrow between model types MT_A and MT_B. Through this subtyping relation, models typed by MT_A are substitutable to models typed by MT_B, *i.e.,*model manipulations defined on MT_B can be *safely reused* on model typed by MT_A.

Isomorphic Model Subtyping An obvious way to safely reuse on a model typed by MT_B a model manipulation from a model type MT_A is to ensure that MT_B contains substitutable concepts (*e.g.*, classes, properties, operations) for those contained by MT_A. However, it is not possible to achieve type group (or model type) substitutability through object subtyping.

Thus, we use an extended definition of *object type matching* introduced by Bruce *et al.* [24] and used by Steel *et al.* to define the *model type matching* relation. Our *object type matching* relation is similar to, but stricter than the latter, because class names must be the same, as well as lower and upper bounds of multiplicity elements. Moreover, every mandatory property in the matching type requires a corresponding property in the matched type, in order to prevent model manipulation from instantiating a type without its mandatory properties.

Definition 8. MOF class T' matches T (written $T' <\# T$) iff:

1 $T.name = T'.name$

2 $T'.isAbstract \Rightarrow T.isAbstract$

3 $\forall op \in T.ownedOperation, \exists S' \in SuperClasses(T')$ such that $\exists op' \in S'.ownedOperation$ and:

- 3.1 $op.name = op'.name$
- 3.2 $op'.type <\# op.type \vee op.type <: op'.type$
- 3.3 $\forall p \in op.ownedParameter, \exists p' \in op'.ownedParameter$ such that:
  - (a) $\exists U' \in SubClasses(p'.type)$ such that $U' <\# p.type \vee p.type <: p'.type$
  - (b) $p.rank = p'.rank$
  - (c) $p.lower = p'.lower$
  - (d) $p.upper = p'.upper$
  - (e) $p.isUnique = p'.isUnique$
- 3.4 $\forall e' \in op'.raisedException, \exists e \in op.raisedException$ such that $e' <\# e \vee e' <: e$

4 $\forall a \in T.ownedAttribute, \exists S' \in SuperClasses(T')$ such that $\exists a' \in S'.ownedAttribute$ such that:

- 4.1 $a.name = a'.name$
- 4.2 $a'.isReadOnly \Rightarrow a.isReadOnly$
- 4.3 $a.isComposite = a'.isComposite$
- 4.4 $a'.type <\# a.type \vee (a'.type <: a.type \wedge a.isReadOnly)$

4.5 $a.lower = a'.lower$

4.6 $a.upper = a'.upper$

4.7 $a.opposite \neq void \Rightarrow a'.opposite \neq void \wedge a.opposite.name = a'.opposite.name$

4.8 $a.isUnique = a'.isUnique$

5 $\forall a' \in T'.ownedAttribute,\ a'.lower > 0 \Rightarrow \exists S \in SuperClasses(T)$ such that $\exists a \in S.ownedAttribute \wedge a.name = a'.name$

Where $SuperClasses(T)$, is the set of all superclasses of T, $SubClasses(T)$, the set of all its subclasses, both including T and $<:$ is the object subtyping relation.

Given the conditions under which objects may be substitutable in the context of a model type, we can define a *model type matching* relation which ensures the safe type group substitutability. Based on the definition of MOF class matching, we redefine the *model type matching* relation as follows:

Definition 9. The *model type matching* relation is a binary relation $\sqsubseteq$ on $ModelType$, the set of all model types, such that $(MT_B, MT_A) \in \sqsubseteq$ (also written $MT_B \sqsubseteq MT_A$) iff $\forall T_A \in MT_A, \exists T_B \in MT_B$ such that $T_B <: \# T_A$.

The *model type matching* relation can be seen as a kind of subgraph isomorphism which takes into account the MOF specificities (*e.g.*, inherited properties and operations). For this reason we call *isomorphic* model subtyping relation a relation which satisfies the *matching* relation.

Non-isomorphic Model Subtyping The fact that MT_B does not match MT_A does not mean that it is not appropriate for substitution. Indeed, the condition for safely substituting a model m for another is that m contains all the necessary information expected to be handled safely by the called model manipulation or to access the desired features. But this information can be under another form than expected (*e.g.*, with different class names) in which case m may be substitutable if the expected form of the information is retrieved.

Model Adaptation is the process of retrieving the information from a model in the expected form. It consists in adapting a model m_B into a model m_A which can be handled by the operation or through which it is possible to access to the desired feature. Thus a *model adaptation* is a way to *create* a *model type matching* relation between two model types. A model adaptation is a function defined at the model type level and applied on models. It takes a model m_B typed by MT_B and returns a model m_A with the same information, but in the form defined by MT_A, *i.e.,* a model whose type matches MT_A.

Definition 10. A model adaptation is a function $adapt_{MT_A}$ from MT_B to MT_C, where MT_A, MT_B and MT_C are model types and such that $MT_C \sqsubseteq MT_A$.

One way to achieve such an adaptation is to implement a model transformation from MT_B to MT_A, in which case $MT_C = MT_A$. Another way is by adding missing types and derived properties from MT_A to MT_B, creating a new model type MT_C with $MT_C \sqsubseteq MT_B$ and $MT_C \sqsubseteq MT_A$. This is the approach followed by Sen *et al.* [171].

Bidirectional Model Adaptation, that is coupled forward adaptation from MT_B to MT_A and backward adaptation from MT_A to MT_B may be needed, depending whether the adaptation is done to reuse an *endogenous* or an *exogenous* model manipulation [134].

If the adaptation to MT_A is done in order to reuse an *endogenous* manipulation, a backward adaptation is necessary in order to reflect changes made to the adapted model on the original

model. Conversely, a backward adaptation is not necessary if the reused feature is an *exogenous* manipulation.

The forward and backward adaptation together form a bidirectional adaptation, which enables the adaptation of a model typed by MT_B into a form which fits the expected model type MT_A but also to reflect the result in the original model. Moreover, a roundtrip adaptation, *i.e.,*applying the forward adaptation then the backward adaptation to the result should lead to an unchanged model. To this end, we use here rules defined by Foster *et al.* for well-behaved lenses (*i.e.,* bidirectional transformation operators) [76].

Definition 11. A bidirectional model adaptation $adapt_{MT_A}$ between model types MT_B and MT_A comprises a function $adapt_{MT_A} \nearrow$ from MT_B to MT_C and a function $adapt_{MT_A} \searrow$ from $MT_B \times MT_C$ to MT_B, where MT_C is a model type such that $MT_C \sqsubseteq MT_A$ and:

- $adapt_{MT_A} \nearrow (adapt_{MT_A} \searrow (m_B, m_C)) = m_C, \forall (m_B, m_C) \in MT_B \times MT_C$
- $adapt_{MT_A} \searrow (m_B, adapt_{MT_A} \nearrow (m_B)) = m_B, \forall m_B \in MT_B$

Bidirectional adaptation can be provided through bidirectional transformations. Bidirectional transformations are studied in different disciplines of computer science (*e.g.*, MDE, graph transformations and databases) to synchronize two data structures (a *source* and a *view*) [47, 95]. In our case, the *source* is the model typed by MT_B found in a context where a model typed by MT_A (our *view*) is expected.

Total Model Subtyping When a model of type MT_B can be used in every context in which a model of type MT_A is expected, we talk about *total* substitutability. Therefore, a subtyping relation which guarantees *total* substitutability is a *total* subtyping relation.

Definition 12. MT_B is a total subtype of MT_A if any model typed by MT_B can be safely used everywhere a model typed by MT_A is expected.

Partial Model Subtyping Conversely, a *partial* subtyping relation enable a model typed by MT_B to be used in a given context (*e.g.*, a given model transformation) where a model typed by MT_A is expected. This notion of *usage context* have been introduced by Kühne in order to define in which cases a specialization relation holds, while it does not hold in the general case [119]. Typically, a *partial* subtyping relation enables a model typed by MT_B to be substituted for a model a of type MT_A in the context of the call `m(a)` if MT_B contains the required features for m, even if MT_B is not a *total* subtype of MT_A.

Definition 13. MT_B is a partial subtype to MT_A wrt. f if models typed by MT_B can be safely used where a model typed by MT_A is expected to use the feature f.

Here, f can be an attribute or an operation from the model or a model manipulation that takes the model as argument. MT_B is a partial subtype to MT_A wrt. f if MT_B is a total subtype of MT_f, where MT_f is a model type which contains only the necessary information to apply or access f safely and such that MT_A is a total subtype of MT_f. We call MT_f the *effective model type* of f.

Definition 14. The effective model type MT_f of a feature f extracted from a model type MT_A is the model type which contains all the required features to access or call f and such that $MT_A \sqsubseteq MT_f$.

This effective model type can be processed using a function which analyzes the model type and extracts its required subset to access a given feature.

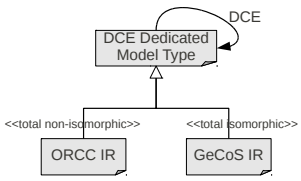


Figure 3.5: Total subtyping

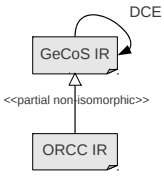


Figure 3.6: Partial non-isomorphic subtyping

Figure 3.7: Two different scenarios of the reuse of DCE between ORCC IR and GeCoS IR

Definition 15. The effective model type extraction function is a function $extractEffectiveMT(MT_A, f)$, with MT_A a model type and f a required feature belonging to MT_A, and such that $MT_f = extractEffectiveMT(MT_A, f)$ is the effective model type of f extracted from MT_A.

One possible way to extract this required subset is to use an approach like the one proposed by Sen *et al.* [171]. They compute a metamodel (called the *effective metamodel*) from a larger metamodel using the *footprint* of a model manipulation, *i.e.,* the set of types and features touched by the manipulation. This footprint can be processed statically, by analyzing the code of the model manipulation or dynamically using a trace of the execution of the operation [100]. The dynamic footprint is more accurate because it contains only the types and features of the objects which have been touched by the operation, whereas the static footprint contains all the types and features which may be touched by the operation. However, the dynamic footprint is also costlier and cannot be used for static type checking (cf. Section 3.1.1.5).

3.1.1.3 Definition of Subtyping Relations for Model Types

From these two criteria (isomorphism of the structures and totality of the subtyping), we define four model subtyping relations to provide model substitutability. In the following MT_A and MT_B are model types and $ModelType$ is the set of all model types.

The first model subtyping relation is the *total isomorphic* subtyping relation, to which the three others refer. MT_B is a *total isomorphic* subtype of MT_A if it contains one matching object type for every object type of MT_A, *i.e.,* if $MT_B \sqsubseteq MT_A$. For example, such a subtyping relation could hold between GeCoS IR and a model type extracted from GeCoS IR by selecting only the relevant concepts for Dead Code Elimination (DCE). In Figure 3.5, where the DCE arrow represent the DCE model manipulation defined on a dedicated model type, this case is represented by the generalization arrow between GeCoS IR and the DCE dedicated model type.

Definition 16. The *total isomorphic* subtyping relation is the matching relation, denoted $MT_B \sqsubseteq MT_A$.

A *partial isomorphic* subtyping relation wrt. feature f holds between MT_B and MT_A if MT_B contains matching object types for every object type belonging to the *effective model type* of f extracted from MT_A, *i.e., MT_B* is *partial isomorphic* subtype of MT_A wrt. feature f if MT_B is a *total isomorphic* subtype of the *effective model* type of f extracted from MT_A.

Definition 17. The *partial isomorphic* subtyping relation wrt. the feature f is a binary relation $\sqsubseteq_f$ on $ModelType$ such that $(MT_B, MT_A) \in <:_f$ (also written $MT_B \sqsubseteq_f MT_A$) iff $MT' \sqsubseteq extractEffectiveMT(MT_A, f)$.

MT_B is a *total non-isomorphic* subtype of MT_A if there is a adaptation able to adapt every model typed by MT_B in a model typed by a *total isomorphic* subtype of MT_A. This adaptation

must be bidirectional, or it would be impossible to reuse *endogenous* model manipulations from MT_A and the subtyping relation would not be *total*. Figure 3.5 represents such a subtyping relation between ORCC IR and the model type dedicated to DCE mentioned above. Loops from the latter are isomorphic to GeCoS IR ones, thus they cannot be isomorphic to loops from the former. Therefore an adaptation is needed, as the one described earlier (see 3.1.1.1).

Definition 18. The *total non-isomorphic* subtyping relation is a binary relation $\sqsubseteq_{\sim}$ on $ModelType$ such that $(MT_B, MT_A) \in \sqsubseteq_{\sim}$ (also written $MT_B \sqsubseteq_{\sim} MT_A$) iff $\exists adapt_{MT_A}$ a bidirectional adaptation from MT_B to MT_C such that $MT_C \sqsubseteq MT_A$.

Finally, model type MT_B is a *partial non-isomorphic* subtype of MT_A wrt. the feature f if there is an adaptation able to adapt a model typed by MT_B in a model typed by a *total isomorphic* subtype of the effective model type of f extracted from MT_A. This adaptation must be bidirectional if f is an *endogenous* feature. Such a *partial non-isomorphic* subtyping relation is pictured in Figure 3.6, where ORCC IR is subtype of GeCoS IR through an adaptation to the *effective model type* of DCE extracted from GeCoS IR.

Definition 19. The *partial non-isomorphic* subtyping relation wrt. the feature f is a binary relation $\lesssim :_f$ on $ModelType$ such that $(MT_B, MT_A) \in \lesssim :_f$ (also written $MT_B \lesssim :_f MT_A$) iff $\exists adapt_{MT_A}$ an adaptation from MT_A to MT_C such that $MT_C \sqsubseteq extractEffectiveMT(MT_A, f)$ and $adapt_{MT_A}$ is a bidirectional adaptation if f is an *endogenous* model manipulation.

3.1.1.4 Contract-aware MOF Class Matching

The subtyping relations aforementioned have been also extended by taking into account OCL contracts for a safe substitutability of models conforming to metamodels including contracts [209]. This provides a safe reuse of model transformations expressed on metamodels that include contracts. Specifically, we extend the MOF class matching by considering contracts matching and provide a technique for analyzing the matching of OCL contracts associated with two classes with different model types.

In this section we describe the contract matching technique we developed to support contract-aware model substitutability. We also describe an Alloy-based prototype tool that supports contract matching.

We consider the use of OCL invariants added to MOF classes to specify additional structural properties, and OCL pre-/post-conditions defined in the context of MOF class operations to specify the model manipulation rules (e.g., transformation) associated with model types. The MOF class matching relation is thus determined by two aspects: the structural features specified using MOF (e.g., classes, properties, operation signatures, etc.) and the contracts expressed using OCL (e.g., invariants and pre-/post-conditions).

The substitutability through model subtyping is a specialization of the Liskov Substitution Principle [127] on the model type system. Specifically the contract matching relation that enables contract-aware model substitutability must abide by the following rules: (1) invariants of the supermodel type cannot be weakened in a sub model type, (2) pre-conditions cannot be strengthened in a sub model type, and (3) post-conditions cannot be weakened in a sub model type. The extended MOF class matching relation is formalized as follows:

Definition 20. Class T' matches T (written $T' <\#$ T) iff their structures match (cf. Def. 3 of [88]), their invariants match and their operation pre-/post-conditions match, where

1 **Invariants Match** is defined as follows:
let T.ownedInvariant = $\{inv_{T1}, inv_{T2}, ..., inv_{Tk}\}$ be the invariants defined for T;
let $result_T = inv_{T1} \wedge inv_{T2} \wedge ... \wedge inv_{Tk}$;
let SuperClass(T) = $\{cls_1, cls_2, ..., cls_n\}$ where cls_i is a superclass of T;

let cls_i.ownedInvariant = { inv_{i1}, inv_{i2}, ..., inv_{ik} } be the invariants defined for cls_i, for i = 1, .., n;
let $result_i = inv_{i1} \wedge inv_{i2} \wedge ... \wedge inv_{ik}$, for i = 1, .., n;
let invs = $result_1 \wedge result_2 \wedge ... \wedge result_n \wedge result_T$;

let T′.ownedInvariant = {inv'_{T1}, inv'_{T2}, ..., inv'_{Tk} } be the invariants defined for T';
let $result'_T = inv'_{T1} \wedge inv'_{T2} \wedge ... \wedge inv'_{Tk}$;
let SuperClass(T′) = {cls'_1, cls'_2, ..., cls'_n} where cls'_i is a superclass of T′;
let cls'_i.ownedInvariant = { inv'_{i1}, inv'_{i2}, ... , inv'_{ik} } be the invariants defined for cls'_i, for i = 1, .., n;
let $result'_i = inv'_{i1} \wedge inv'_{i2} \wedge ... \wedge inv'_{ik}$, for i = 1, .., n;
let invs′ = $result'_1 \wedge result'_2 \wedge ... \wedge result'_n \wedge result'_T$;

The invariants of T and T′ match if Models(invs) ⊇ Models(invs′), where $Models(invs)$ returns all models that satisfy $invs$ and $Models(invs')$ returns all models that satisfy $invs'$.

2 **Pre-/post-conditions Match** is defined as follows:
∀ op ∈ T.ownedOperation, ∃ S' ∈ SuperClasses(T') such that ∃ op' ∈ S'.ownedOperation and:

2.1 let op.ownedPrecondition = {pre_1, pre_2, ..., pre_k } be the pre-conditions defined for op;
let pres = $pre_1 \wedge pre_2 \wedge ... \wedge pre_k$;
let op′.ownedPrecondition = {pre'_1, pre'_2, ..., pre'_k } be the pre-conditions defined for op';
let pres′ = $pre'_1 \wedge pre'_2 \wedge ... \wedge pre'_k$;

2.2 let op.ownedPostcondition = {$post_1$, $post_2$, ..., $post_k$ } be the post-conditions defined for op;
let posts = $post_1 \wedge post_2 \wedge ... \wedge post_k$;
let op′.ownedPostcondition = {$post'_1$, $post'_2$, ..., $post'_k$ } be the post-conditions defined for op';
let posts′ = $post'_1 \wedge post'_2 \wedge ... \wedge post'_k$;

The operation specifications of T and T′ match if Models(pres′) ⊇ Models(pres) and Models(posts) ⊇ Models(posts′)

Analyzing the Matching of Contracts Definition 20 can be used to formally reason about the matching relation between two MOF classes with contracts. The MOF class matching relation in Definition 20 includes the matching of the contracts from classes of two model types. Consequently, analyzing such relations requires one to formally analyze the relation between contracts (e.g., to check if the models satisfying one contract includes the models satisfying the other). To do this, a query function $Models(MT, C)$ is used to compute all models that both conform to a model type MT and satisfy an OCL contract C defined in MT. Thus given contract C_1 in a candidate supermodel type MT_1 and contract C_2 in a candidate sub model type MT_2, C_1 matches C_2 iff (1) C_1, C_2 are invariants, and $Models(MT_1, C_1) \supseteq Models(MT_2,$

C_2), (2) C_1, C_2 are pre-conditions, and $Models(MT_2, C_2) \supseteq Models(MT_1, C_1)$, and (3) C_1, C_2 are post-conditions, and $Models(MT_1, C_1) \supseteq Models(MT_2, C_2)$.

Checking the contract matching requires a tool to implement the functionality of the query function $Models(MT, C)$. We use the Alloy Analyzer [98] for this purpose. The Alloy Analyzer is used to analyze Alloy specifications. It is supported by a SAT-based model finder. The Alloy Analyzer can generate models that conform to a model type expressed in Alloy in terms of signatures and fields that specify the model type structure and a predicate that expresses the contracts. In this paper we use the Alloy Analyzer at the back-end to check whether two contracts match.

For example, given a candidate supermodel type MT_1 and a candidate sub model type MT_2, with two OCL invariants respectively, C_1 and C_2, the procedure below can be used to check if C_1 matches C_2.

1. (preprocess) Since model subtyping requires each element in the supermodel type to be matched by an element in the sub model type (see Definition 20), the contract defined in the supermodel type refers to elements that also exist in the sub model type. Thus we can move C_1 to MT_2, and use only the sub model type (i.e., MT_2) to check whether C_1 and C_2 match.

2. Transform MT_2 to an Alloy model using the technique described by Sun *et al.* [183]. Convert C_1 and C_2 into two Alloy predicates, P_1 and P_2, respectively.

3. Run an empty predicate in the Alloy Analyzer to search for a model conforming to the model type MT_2. If the Analyzer returns no model satisfying the empty predicate (i.e., $Models(MT_2, \emptyset) = \emptyset$), $Models(MT_2, C_1) = \emptyset$ and $Models(MT_2, C_2) = \emptyset$. In this case C_1 matches C_2 since $\emptyset$ is a subset of $\emptyset$; otherwise, continue to the next step.

4. Run P_1 and P_2 respectively. If the Alloy Analyzer returns no model for each predicate (i.e., $Models(MT_2, C_1) = \emptyset$ and $Models(MT_2, C_2) = \emptyset$), then C_1 matches C_2; if the Alloy Analyzer returns a model (or models) for only P_1, then C_1 matches C_2; if the Alloy Analyzer returns a model (or models) for only P_2, then C_1 does not match C_2; otherwise, continue to the next step.

5. Run a predicate to search for a model satisfying both P_1 and P_2. If the Alloy Analyzer returns a model satisfying the predicate, continue to the next step; otherwise, C_1 does not match C_2.

6. Run a predicate P_3 to search for a model satisfying both P_1 and $\neg P_2$ (i.e., the negation of P_2), and another predicate P_4 to search for a model satisfying both P_2 and $\neg P_1$. If the Alloy Analyzer returns no model for both P_3 and P_4 (i.e., $Models(MT_2, C_1) = Models(MT_2, C_2)$), C_1 matches C_2; if the Alloy Analyzer returns a model (or models) satisfying only P_3, $Models(MT_2, C_1) \supset Models(MT_2, C_2)$ and C_1 matches C_2; otherwise, C_1 does not match C_2.

The approach uses the Alloy Analyzer at the back-end to analyze the relation between two contracts, and it thus requires a translation from OCL expressions to Alloy specifications. The OCL to Alloy translation used in the prototype tool we developed is based on translation rules described in work by Bordbar et al. [3].

Contract Matching Checking Tool The contract matching approach described in the previous subsection has been implemented in a prototype tool. Figure 3.8 shows an overview of the prototype tool. It consists of an OCL parser, an Ecore[5]/OCL transformer and the

[5]Ecore is an implementation aligned with MOF included in the Eclipse Modeling Framework [182].

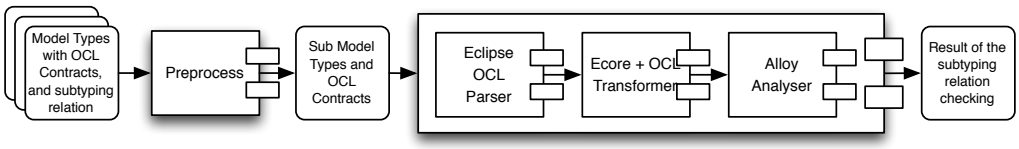


Figure 3.8: Contract Matching Checking Tool Overview

use of the Alloy Analyzer. The Ecore/OCL transformer is developed using Kermeta [147], an aspect-oriented metamodeling tool. The inputs of the prototype are (1) an Ecore file that specifies two model types, and (2) a textual OCL file that specifies the contracts from each model type. The model types and contracts are automatically transformed to an Alloy model consisting of signatures and predicates.

The prototype provides several interfaces to check contract matching. For example, *matchInv(inv1: Constraint, inv2: Constraint)* is used to check whether $inv1$ matches $inv2$. In addition, *matchInvs(cls1: Class, cls2: Class)* can be used to check whether the invariants defined in $cls1$ and the invariants defined in $cls2$ match.

3.1.1.5 Putting Subtyping Relations to Work

Defining model subtyping relations is not sufficient to build a type system. Indeed, a type system implements one or more subtyping relations and provides ways to declare and check them. Thus, we discuss here the ways to declare and check subtyping relations and the respective drawbacks and advantages of these approaches for an implementation of a model-oriented type system.

Declaration of Subtyping Relations Subtyping relations can be declared in two ways: *explicitly* and *implicitly*. We call a subtyping relation declaration *explicit* when a syntactic construct is used to state the subtyping relation. Conversely, if the type system infers the subtyping relation from the information it can gather about the types or the use which is done from their instances, the declaration of the subtyping relation is *implicit*. In addition, the declaration of the subtyping relation can take place either *at the definition* of a type or *after the definition* of the subtype and the supertype involved in the subtyping relation.

The way to declare model subtyping relations may affect the possibilities that these relations offer through the type system. For example, a *non-isomorphic* model subtyping relation can be declared *implicitly*. To this end, a tool able to infer adaptations is necessary. Such inference can be done through patterns which are known to be safe or using ontologies to find corresponding class or feature names. However, an *implicit* adaptation mechanism will be more limited in terms of possible adaptations than an *explicit* one, which let the user define its adaptation based on its knowledge of the two model types involved. On the other hand, an *explicit* adaptation mechanism needs appropriate syntactic constructs and analyses to ensure that an adaptation is safe.

Declaration of a subtyping relation *at the definition* of a type is a kind of documentation, letting know what are the subtypes or supertypes of the defined type. Conversely, it is not always possible or desirable to add this information in a type, particularly if the subtyping relation is required for a very specific use (*e.g.*, a *partial* subtyping relation for a single model manipulation) or legacy code where existing model types should be modified. In such cases, declaring the subtyping relation *after the definition* of the involved types may be a solution.

Finally, declaration of a subtyping relation *explicitly at the definition* of a model type could allow inheritance. That is, reuse of the structure of the supertype, with the possibility to

redefine or modify it in the subtype without breaking model subtyping. Moreover, if *explicit* declaration *at the definition* is the only way to declare model subtyping relations, it prevents the type system to use subtyping relations which are unknown from the user, and thus prevents accidental substitutability.

Checking of Subtyping Relations Checking of the subtyping relations is the verification that a subtyping relation holds. Regardless of the way the subtyping relation is declared, this check can be processed either at design time, *i.e.,* during the compilation or interpretation process, or at runtime.

Here again, the way to check model subtyping relations can impact the facilities provided for model manipulations. On the one hand, design time (or static) checking enables earlier detection (*i.e.,* than runtime check) of type errors and programming mistakes and thus earlier user feedback. It also enables tools to provide more facilities, such as type-based compiler optimizations, auto-completion or impact analyses. Moreover, compared to runtime checking, design time checking needs significantly fewer tests to achieve the same level of runtime safety.

On the other hand, runtime checking can be processed with more precise type information. When the program is running, the actual type of a variable is known rather than its declared type. Although possibly slower because of the process of the check during the execution of the program, dynamic checking enables valid programs which would be forbidden by a static type checker because of a lack of information. In the context of model types, knowing the actual model would enable the extraction of its model type and would possibly enable subtyping relations forbidden by a static type checker.

3.1.1.6 A Foundational Framework to Classify Existing Approaches

Several approaches have been proposed in the last decade to provide engineering facilities for model manipulation reuse. We show in this section how the different model subtyping mechanisms (*i.e.,* total/partial and isomorphic/non-isomorphic model subtyping, declaration and checking) defined in this paper can be used to classify these approaches through a unified theory. Figure 3.9 summarizes this classification. The question marks indicate the lack of information about the given mechanism.

Isomorphic vs. Non-isomorphic Subtyping Relations To the best of our knowledge, the only approach using an isomorphic subtyping relation is the bidirectional subset of the *adaptation* DSL proposed by Babau *et al.* [109, 8]. All other approaches either let class names vary or go further, enabling adaptations such as $n - to - 1$ concepts *binding* or navigation and filtering of features. The latter use different mechanisms to *bind* the subtype to its supertype and express the adaptation (*e.g., adaptation* and *binding* DSLs or static introduction). The rarity of *isomorphic* subtyping relations can be explained by the restrictions such relations impose, restrictions which can be safely relaxed in some cases, *e.g.*, class names modification.

Total vs. Partial Subtyping Relations Excepting one approach which allows the extraction of the *effective metamodel* from a model manipulation [171], all existing approaches are *total*. To be *total* a *non-isomorphic* subtyping relation must handle bidirectional adaptation. Bidirectionality is tackled in existing approaches by almost *isomorphic* relations [195, 46, 180, 164, 109] or by generating an adapted model manipulation rather than adapting the model [49, 185, 206].

Declaration of Subtyping Relations All the existing approaches declare the subtyping relation or *binding after the definition* of the two model types (or their equivalent). However,

Figure 3.9: Classification of different model manipulation reuse approaches

	Total / partial	Iso / non-iso	At / after definition	Explicit / implicit	Checking	Legacy tool reuse
Varró *et al.* [195]	Total	Non-iso (Class renaming)	After	Implicit	?	No
Cuccurru *et al.* [46]	Total	Non-iso (Abstract class renaming)	After	Explicit (Genericity and explicit object subtyping)	?	Yes
Steel *et al.* [180]	Total	Non-iso (Class renaming, multiplicities contraction)	After	Implicit	At compile-time, with possible run-time type errors	Yes
Sanchez Cuadrado *et al.* [164]	Total	Non-iso (Class renaming, multiplicities contraction)	After	Explicit (Binding DSL)	?	Yes
Sen *et al.* [171]	Partial (Effective metamodel)	Non-iso (Potentially any adaptation)	After	Explicit (Static introduction)	At compile-time, with possible run-time type errors	Yes
De Lara *et al.* [49, 185, 206]	Total	Non-iso (Class renaming, navigation and filtering of properties, $n-to-1$ bindings)	After (Binding), At definition (Specialization)	Explicit (Binding DSL)	?	No
Babau *et al.* [109, 8]	Total (Bidirectional subset)	Iso (Bidirectional subset)	After	Explicit (Adaptation DSL)	?	Yes

de Lara *et al.* authorize specialization of model types (called *concepts* in their terminology) using a mechanism close to inheritance (*i.e.,at definition*) [49]. Only two approaches declare subtyping relations *implicitly* [195, 180] whereas the others use *explicit* mechanisms mainly through DSLs [164, 49, 185, 206, 8, 109], with the exception of the approaches from Cuccuru *et al.* [46] and Sen *et al.* [171] which use respectively genericity and static introduction.

Checking of Subtyping Relations Little is said about the checking of the subtyping relations, apart from the work of Steel *et al.* [180], in which subtyping relations are checked *at compile time*. De Lara *et al.* [49] mention a notion of *valid* binding, but do not formalize it.

Legacy Tools Reuse One group of our examples, abstract interpretation analyses, are implemented in an existing tool (P-Interproc). Among the existing approaches, some need to specifically define the model manipulation to be reused [195], or to process it in order to generate

an adapted model manipulation [49, 185, 206]. By doing so, they prevent reusing existing model manipulations which have not been defined using their own mechanisms or for which the sources are not available. The other approaches, which enable a subtyping relation with a legacy tool, are the ones with the fewest possible adaptations [46, 180, 164, 8, 109], or without any guarantee on the bidirectionality of such adaptations [171].

3.1.2 Melange: A Meta Language for Language Reuse

The content of this subsection is an adapted excerpt from the following publication:

Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. Melange: A Meta-language for Modular and Reusable Development of DSLs. In *8th International Conference on Software Language Engineering (SLE)* , Pittsburg, USA, October 2015. ACM. [56]

Despite the wide range of domains in which DSMLs are used and their constant evolution, many of them are very close and share commonalities such as a particular action language, a common paradigm, etc. (*i.e.,* the family of DSMLs for statechart [45]). Recent work in the SLE community focused on language workbenches that support the modular design of DSMLs, and the possible reuse of such *modules* (usually using a scattered clause `import` linking separate artefacts) [106, 189]. Besides, particular composition operators have been proposed for unifying or extending existing languages [136]. However, while most of the approaches propose either a diffuse way to reuse language modules, or to reuse as is complete languages, there is still little support for easily assembling language modules with customization facilities (*e.g.*, restriction) in order to finely tune the resulting DSML according to the language designer's requirements.

In this subsection, we present Melange, a tool-supported dedicated meta-language to safely assemble language modules, customize them and produce new DSMLs. Melange provides specific constructs to assemble together various abstract syntax and operational semantics artifacts into a DSML. DSMLs can then be used as first class entities to be reused, extended, restricted or adapted into other DSMLs. Melange relies on a particular model type system that statically ensures the structural correctness of the produced DSMLs, and specific subtyping relationships between DSMLs to reason about their substitutability. Newly produced DSMLs are correct by construction, ready for production (*i.e.,* the result can be deployed and used as-is), and reusable in a new assembly.

3.1.2.1 Approach Overview

Figure 3.10 gives a high-level overview of our approach. On the right side are legacy language artifacts that must be reused and assembled to build new DSMLs. Imported artifacts include abstract syntax and semantics, possibly with their corresponding tooling and services (*e.g.*, checkers, or transformations). These tools consist in manipulating models conforming to a particular metamodel. Similarly, semantic definitions directly access and manipulate model elements in an execution or compilation purpose. Hence, abstract syntax and semantics are related one to another through *binding* relations: semantic artifacts *require* a particular shape of abstract syntax, which is *provided* by a given metamodel. On the left side of Figure 3.10 are the newly built languages. *Assembly operators* (*merge*, *weave*) realize the transition from legacy artifacts to new DSMLs. They import and connect disparate language artifacts, *e.g.*, by merging and slicing different abstract syntaxes or by binding a given *Sem* to a new *AS*. Naturally, the same artifacts can be reused in different assemblies. The output of assembly operators is encapsulated in a language definition. Once new assemblies are created, *customization operators* (*slice*, *merge*, *inherits*) offer the possibility to refine the newly built DSMLs so as to meet additional

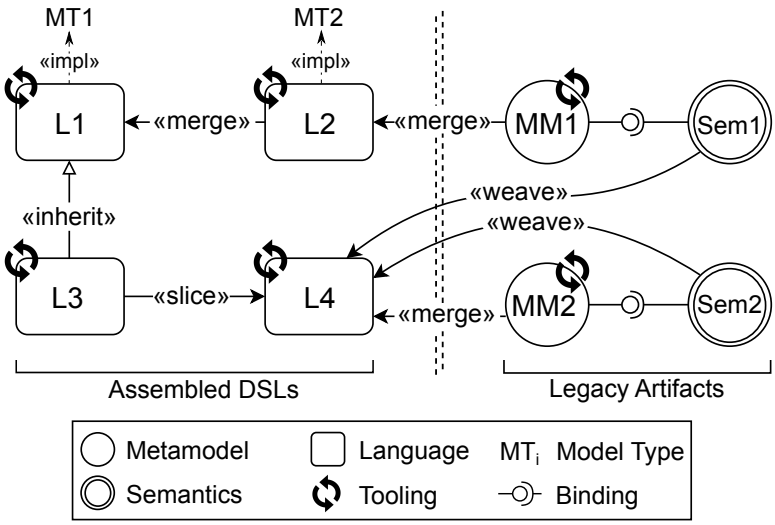


Figure 3.10: DSMLs Assembly and Customization with Melange

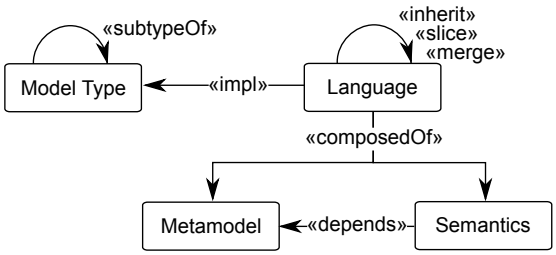


Figure 3.11: Model Typing Relations

requirements or to fit a specialized context. Both assembly and customization operators are captured in an algebra.

Assembling and customizing DSMLs is a complex task that requires checking the composability of heterogeneous parts and the validity of the result. For example, based on Figure 3.10, it is clear that not all *Sem* may be bound to any *AS*: the elements manipulated by *Sem* must be provided by a corresponding *AS*. Similarly, the intuitive meaning of inheritance, as found in most OO languages, implies compatibility between the super- and sub- elements. It follows that the compatibility between $L1$ and $L3$ in Figure 3.10 must be ensured to guarantee that $L1$'s tooling can be reused for $L3$. What is missing here to guarantee these properties is an abstraction layer that would support reasoning about the compatibility between different languages artifacts. In our approach, we rely on the notion of model typing as introduced by Steel *et al.* [180] and further refined by Guy *et al.* in the previous section [88]. Model types are structural interfaces over the abstract syntax of a language, defined by a metamodel. As such, they also take the form of a metamodel. They are linked one another by subtyping relations that specify if a model conforming to a given metamodel can be manipulated through another metamodel. Several metamodels may *implement* the same model type, meaning that transformations and tools defined over a model type can be reused for all matching metamodels. Moreover, model typing allows to reason about the compatibility between different metamodels.

We further extend the concept of model typing by explicitly separating implementations of

languages (*i.e.,*abstract syntax, semantics, and tools) from their structural interfaces (*i.e.,*model types exposing part of their features) as canonical representation of languages. As depicted in Figure 3.11, each language has at least one model type that captures its structural interface. Then, the associated type system enables reasoning about compatibility between different artifacts, *e.g.*, to check whether a given semantics can be applied on a given abstract syntax, or to ensure that within an inheritance relation the sub-language remains compatible with the super-language.

In the following we introduces the proposed algebra for assembling and customizing DSMLs. We first provide the definitions and concepts required to define the algebra (Section Section 3.1.2.2). We then introduce the operators for language assembly (Section Section 3.1.2.3) and customization (Section Section 3.1.2.4).

3.1.2.2 Language Definition

Based on the informal conceptual model aforementioned, we define a language $\mathcal{L}$ as a 3-tuple of its abstract syntax, semantics, and exact model type:

$$\mathcal{L} \triangleq \langle AS, Sem, MT \rangle$$

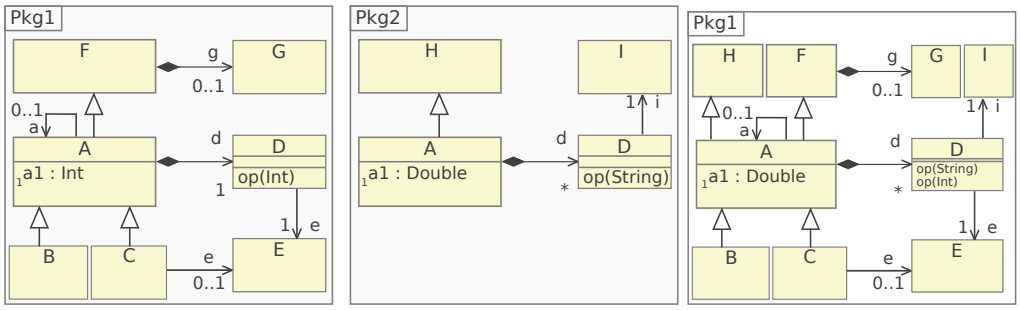
Including the exact model type of a language into the tuple allows to directly specify the impact of each of the operators of the algebra on the typing layer. As explained in this section, model types also indirectly support the reuse of languages tooling. In the following, for any language $\mathcal{L}$, we denote $AS(\mathcal{L})$ its abstract syntax, $Sem(\mathcal{L})$ its semantics, and $MT(\mathcal{L})$ its exact model type. On non-ambiguous cases, we simply refer to them as AS, Sem, and MT. The next sub-sections detail each of them.

Syntax and Syntax Merging In our algebra, the abstract syntax AS of a language $\mathcal{L}$ is specified using a metamodel, *i.e.,*a multigraph of classes and their relations. When assembling several abstract syntaxes, their concepts must be merged together so that the resulting abstract syntax is no less capable than its ancestors. Informally, this means that the abstract syntax resulting from the merge must incorporate concepts from all languages and merge the definitions of shared elements. In our specific case, merging several abstract syntaxes boils down to the problem of metamodel merging. Figure 3.12 illustrates the *PackageMerge* operator on a simple example. Following the UML terminology, we use the terms *merged package*, *receiving package*, and *resulting package* to refer to the three packages involved in the merging operation. Similarly, the terms *receiving language* and *resulting language* will be used throughout this section.

Depending on the meta-language used for defining metamodels, different merging operators may be employed with different politics for matching and merging rules, conflicts management, etc.. The choice of the concrete semantics of the merge operator is left to the implementer of the algebra. In the remainder of this section, we denote $\circ$ the abstract syntax merging operator.

Model Typing Each language $\mathcal{L}$ has one exact model type MT. Like abstract syntaxes, model types are described with a metamodel. The exact model type of a language is its most precise structural interface, *i.e.,*the model type that exposes all its features. Thus, the exact model type of a language exposes both its concepts and their relations (*i.e.,*its metamodel) and the signature of its semantics. Hence, the exact type MT of a language $\mathcal{L}$ is defined as the structural merge (*i.e.,*through $\circ$) of its abstract syntax and the signature of its semantics. The signature sig of a semantics Sem is introduced in the next paragraph.

$$MT(\mathcal{L}) \triangleq AS(\mathcal{L}) \circ sig(Sem(\mathcal{L}))$$



(a) Receiving Package (b) Merged Package (c) Resulting Package

Figure 3.12: Merge Operator

Any change in either the abstract syntax or the signature of the semantics of a language will result in a different type. The issue of tooling is indirectly addressed through the reasoning capabilities provided by the model typing layer: if the result of the application of operators leads to a language $\mathcal{L}$ whose model type MT is a subtype of the model type MT' of another language $\mathcal{L}'$, then tools defined for $\mathcal{L}'$ can be reused for $\mathcal{L}$. In the following, we denote $<:$ the subtyping relation between model types and $<\bullet$ the implementation relation between a metamodel and a model type.

Semantics and Semantics Merging The semantics Sem of a language $\mathcal{L}$ is defined by a sequence of aspect definitions A_i^t, where A is a class, t is a pointcut and i is the index of A in the sequence. In this case, the pointcut t specifies the concept of the language's abstract syntax (a meta-class) on which the aspect must be woven. The advice is the class itself, consisting of attributes and methods. When a joinpoint is found, *i.e.,* when a matching concept is found in the language, elements of the advice are inserted in the target meta-class. Since aspects are defined using classes in an OO manner, they may inherit from each other. To cope with possible specialization and redefinition of methods, aspects are ordered by hierarchy in a sequence:

$$\begin{aligned} Sem(\mathcal{L}) \triangleq (A_i^t \in Aspects) \text{ where} \\ \forall A_i^t \in Sem(\mathcal{L}), \exists c \in AS(\mathcal{L}) : c \; match \; t \\ \forall A_i^t, A_j^t \in Sem(\mathcal{L}) : A_i^t \lhd A_j^t \implies i > j \end{aligned}$$

where *match* denotes the joinpoint matching relation and $\lhd$ denotes the class inheritance operator. For a language to be well-formed, each of its aspects must have a matching meta-class in its abstract syntax; this is what the first property ensures. Ordering the aspects that compose a semantics in a sequence let the choice of linearization and/or disambiguation opens to the implementer when several aspects are in conflict (*e.g.*, insert the same method on the same target t). The merging of two semantics, denoted $Sem \bullet Sem'$, consists in producing a new semantics structure. As the definition shows, merging two semantics is equivalent to concatenating their sequences of aspects. So, this operator is not commutative and any redefinition of an aspect or method in Sem' overrides the previous definition in Sem:

$$Sem \bullet Sem' \equiv Sem \frown Sem'$$

where $\frown$ denotes the sequence concatenation operator. We also denote sig the *signature* of an aspect A. The signature of an aspect is a metamodel that exposes all the features (*i.e.,*class

names, attributes and methods) defined in an aspect and its dependencies, omitting the concrete method bodies. The signature of a semantic specification Sem is thus the structural merge of the signature of the aspects that compose it:

$$sig(Sem) \triangleq \bigcup_{A_i^t \in Sem} sig(A_i^t)$$

3.1.2.3 Operators for Language Assembly

Syntax Merging When building new languages, it is likely that previously defined language abstract syntax fragments may be reused as is. For instance, the syntactic constructs of a simple action language (*e.g.*, with expressions, attributes access, basic I/O) may be shared by all languages that encompass the expression of queries or actions. This first scenario of language assembly thus consists in importing a fragment of abstract syntax from another language in order to reuse (part of) its definition. In such a case, the language resulting from the merge of the receiving language and the merged abstract syntax must incorporate all the concepts of both, while preserving the semantics of the receiving language. Also, its model type must be updated accordingly to incorporate the new syntactic constructs. Hence, we define the abstract syntax merging operator, denoted $\xleftarrow{m}$, as follows:

$$\mathcal{L} \xleftarrow{m} AS' = \langle AS \circ AS', Sem, MT \circ AS' \rangle$$

In most cases, the resulting model type $MT' = MT \circ AS'$ is a subtype of both AS and MT since it incorporates the features of both. It is however worth noting that new elements introduced in a model type with the $\circ$ operator may break the compatibility with the super model type (*e.g.*, the introduction of a new mandatory feature [88]). In the former case, when the compatibility can be ensured through subtyping, tooling defined over AS' and/or $\mathcal{L}$ (*e.g.*, transformations, checkers) can be reused as is on the resulting language.

Semantic Weaving Another scenario of language assembly consists in importing predefined semantic elements in a language. When different languages share some close abstract syntax, such as different flavors of an action language, their semantics are likely to be similar, at least for the common subparts (*e.g.*, the semantics of integer addition is likely to remain unchanged). When the case arises, one would like to import the semantics definition of addition from one action language to another. We denote $\xleftarrow{w}$ the semantics weaving operator, which consists in weaving a semantics Sem' on a language $\mathcal{L}$. In such a case, the two semantics are merged and the exact type of $\mathcal{L}$ is updated to incorporate the syntactic signature of the new semantics.

$$\mathcal{L} \xleftarrow{w} Sem' = \langle AS, Sem \bullet Sem', MT \circ sig(Sem') \rangle$$

Following the previous definitions, this operator can be successfully applied only if there is a matching meta-class in AS for each aspect in Sem and Sem'. Since the two semantics are concatenated, Sem' may override any previous definition of Sem, meaning that the semantics merge operator may be employed either to augment or to override part of the semantics of the receiving language $\mathcal{L}$. The semantics weaving operator is thus particularly relevant for incrementally implementing semantic variation points [27].

3.1.2.4 Operators for Language Customization

In the previous subsection, we specified how the abstract syntax merging and semantics weaving operators help to build new languages by assembling predefined fragments of abstract syntax

and semantics. However, although the reuse of language artifacts significantly decreases the development costs, the resulting languages may not fit exactly the language designer's expectations. Thus, we introduce in this section an algebra for language customization. Customization may include specialization of the abstract syntax or semantics of a language for a given context, restriction to a subset of its scope or composition with (possibly part of) other language definitions. In a recent paper, Erdweg *et al.* propose a taxonomy of different composition operators between languages, including language extension, restriction, and unification [64]. The operators of our algebra closely match their taxonomy: the *inheritance* operator is similar to language extension, the *slicing* operator is similar to language restriction, and the *merging* operator is similar to language unification.

Language Merging Situations arise where two independent languages may be composed to form a more powerful one. For instance, a finite-state machine language may be defined as a basic language of states and labeled transitions combined to an action language for expressing complex guards and actions. The resulting language may in turn be merged with a language for expressing classifiers where the state machines would describe their behavior. To support this kind of scenario, we introduce the language merging operator, denoted $\uplus$. The output of this operator is a new language that incorporates both the syntactic and semantic definitions of its two operands. In this case, the receiving language is augmented with the merged language to produce the resulting language. Since the merged language can override part of the semantics of the receiving language, order matters and commutativity can not be ensured.

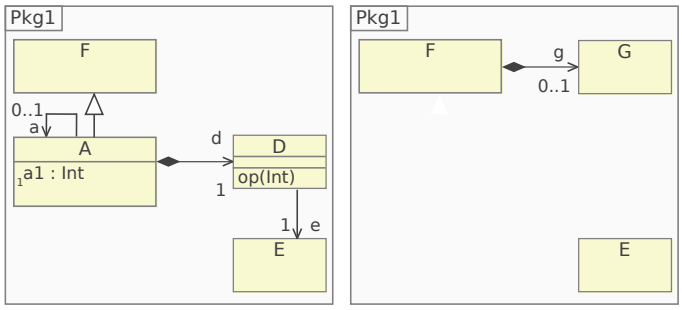
$$\mathcal{L} \uplus \mathcal{L}' = \langle AS \circ AS', Sem \bullet Sem', MT \circ MT' \rangle$$

Language Inheritance In essence, the language inheritance operator is similar to the language merging operator, as both aims to combine the definitions of two languages. The language inheritance operator, denoted $\oplus$, differs from the language merging operator in that it does not consider the two languages on equal terms: a *sub-language* inherits from a *super-language*. Moreover, the language inheritance operator ensures that the sub-language remains compatible with its super-language. Whatever the subsequent operators applied to the sub-language, it must remain compatible with the super-language, otherwise an error is reported. Concretely, it means that the exact model type of the sub-language must remain a subtype of the exact model type of the super-language: the $MT{<:}MT'$ property is conservative, meaning that any operators apply on $\mathcal{L}$ must not violate it. In a sense, the language inheritance operator supports some sort of language design-by-contract, as the language designer is assured that tools defined over $\mathcal{L}'$ will be reused untouched on $\mathcal{L}$.

$$\begin{aligned} \mathcal{L} \oplus \mathcal{L}' = \langle AS \circ AS', Sem' \bullet Sem, MT'' \rangle \text{ where} \\ MT'' = MT \circ MT' \text{ and} \\ MT''{<:}MT' \end{aligned}$$

Note that in this case, the semantics Sem and Sem' are concatenated in reverse order $Sem' \bullet Sem$. The sub-language first inherits the abstract syntax and semantics of its super-language, and may then override part of the inherited artifacts to further refine its definition.

Language Slicing Model slicing [16, 170] is a model comprehension technique inspired by program slicing [203]. The process of model slicing involves *extracting* from an input model a subset of model elements that represent a *model slice*. Slicing criteria are model elements from the input model that provide entry points for producing a model slice. The slicing process starts



(a) Positive slicing on MM_1 with $A.a$ as input

(b) Negative slicing on MM_1 with D as input

Figure 3.13: Examples of metamodel slices

by slicing the input model from model elements given as input (the slicing criteria). Then, each model element linked (*e.g.*, by inheritance or reference) to a slicing criterion is sliced, and so on until no more model elements can be sliced. For instance, model slicing can be used to extract the static metamodel footprint MM' of a model operation defined over a metamodel MM, *i.e.,*extracting the elements of MM used by the operation [100]. Model slicing can be positive or negative. Positive model slicing consists of slicing models according to structural criteria. These criteria are the required model elements from which the slice is built. For instance, one may want to slice the MM_1 metamodel using as slicing criterion the reference a of the class A. This slicing consists of statically extracting all the elements of MM_1 in relation with a (a included). The result of this slicing is depicted by Figure 3.13a: the class A that contains a is sliced; the super class of A (F) is sliced; the A's references with a lower cardinality greater than 0 are sliced (only the mandatory references and attributes are sliced); the target classes of these references (*e.g.*, D) are sliced. This slicing process continues recursively until no more elements can be sliced. We extended the model slicing principles proposed by Blouin *et al.* [16] to support negative slicing. Negative slicing consists of considering the slicing criteria as model elements not to have in the slice. For instance, a negative slicing of MM_1 with the class D as slicing criterion produces the slice depicted by Figure 3.13b: a clone, that will be the output slice, of MM_1 is created; The class D is removed from this clone; all the classes that have a mandatory reference to D are removed (class A); all lower classes of the removed classes are also removed (classes B and C). This slicing process continues recursively until no more elements can be removed.

Model slicing can be used to perform language restriction. For instance, a language designer may want to shrink a legacy metamodel to its sub-set used by a set of model operations of interest [100]. This consists of a positive slicing from a set of operations. A language designer may also want to restrict the features of a language (*e.g.*, removing specific features of a programming language) to reduce its expressiveness [64]. This consists of a negative slicing from unwanted elements.

In the context of language engineering, we leverage the slicing operation to permit a language designer to slice a language according to some slicing criteria, formalized as follows. Given a language $\mathcal{L}_1 \triangleq (AS_1, Sem_1, MT_1)$, slicing $\mathcal{L}_1$ using slicing criteria c consists of slicing positively or negatively (resp. denoted Λ^+ and Λ^-, or Λ^+_- when considering both operators) its abstract syntax AS_1 using c to produce a new abstract syntax AS_2, such that $AS_2 \subseteq AS_1$. Then, the aspects A^t_i, that compose Sem_1, that only refer to elements defined in AS_2 are extracted to

form Sem_2, as formalized as follows:

$$\begin{aligned}\Lambda^{+}_{-}(\mathcal{L}_1, c) = \langle AS_2, Sem_2, MT_2\rangle, \text{ where:}\\ AS_2 \triangleq \lambda^{+}_{-}(AS_1, c),\\ Sem_2 \triangleq \left\{A^t_i \in Sem_1, fp(A^t_i, AS_1) \subseteq AS_2\right\},\\ MT_1 <: MT_2,\\ AS_2 \subseteq AS_1\end{aligned}$$

The footprint operation (denoted fp) extracts the metamodel elements of AS_1 used in the aspects a. The choice of applying a positive ($\Lambda+$) or negative ($\Lambda-$) slicing is made by the language designer during the language design according to his/her requirements. The abstract syntax slicing operation [16] (denoted λ^+, λ^-, or λ^+_-) slices a given abstract syntax AS_1 according to slicing criteria c to produce an output abstract syntax AS_2. Because of the strict slicing that extracts metamodel elements by assuring the conformance, the model type MT_1 is a sub-type of the output MT_2.

3.1.2.5 Specification of Melange

Melange is a meta-language and framework for DSML engineering. Instead of providing its own dedicated meta-languages for the specification of each part of a DSML (abstract syntax, type system, semantics, etc.), Melange relies on other independently-developed components to provide such functionalities. The abstract syntax of DSML is specified using the Ecore implementation of the EMOF standard provided by the Eclipse Modeling Framework (EMF)[6]. The choice of Ecore is motivated by the success of EMF both in the industry and academic areas. This allows Melange to possibly integrate a wide range of existing DSMLs: over 300 metamodels can be found on the "metamodel zoo" [139], over 9000 on Github. For semantics specification, Melange relies on the Xtend programming language to express operational semantics with the definition of aspects. The algebra previously introduced has been implemented within the Melange language, providing features for assembly and customization of legacy DSMLs artifacts. Overall, Melange is tightly integrated with the EMF ecosystem. Newly built DSMLs can thus benefit from other EMF-compatible components such as Xtext [71] for defining their textual concrete syntax or Sirius[7] for their graphical representation. Melange is bundled as a set of Eclipse plug-ins[8].

In the rest of this section, we present the meta-language Melange through its abstract syntax, concrete syntax, implementation choices and integration with the EMF ecosystem.

Abstract Syntax The abstract syntax of Melange (the metamodel depicted in Figure 3.14) includes the concepts and relations discussed in the approach overview (cf. Figure 3.11). This abstract syntax is supplemented with static semantics rules expressed as OCL constraints not presented here for the sake of conciseness. *LanguagesSpec*, the root of Melange's abstract syntax, defines a meta-program that: specifies an assembly of legacy DSMLs; delimits the scope for the inference and checking of model typing relations.

A *Language* is defined by its *Metamodel* and its associated *Semantics*. A *Metamodel* is composed of a set of *Class*es. A *Semantics* composed of a set of *Aspect*s used to weave behavior into its classes [102]. This mechanism relies on static introduction and is inspired by the concept of open classes. As specified in the algebra, operators can be applied on languages. A language

[6] Cf. https://www.eclipse.org/modeling/emf
[7] Cf. https://eclipse.org/sirius
[8] Melange's source code is freely available http://melange-lang.org

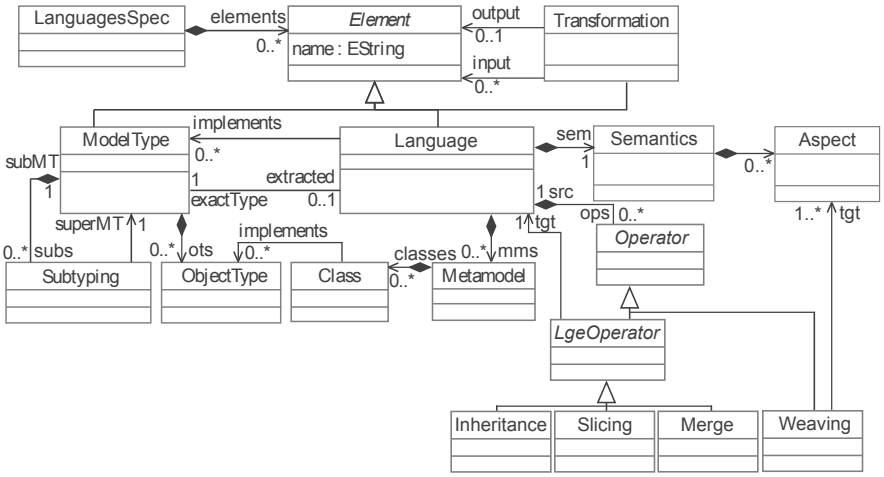


Figure 3.14: Excerpt of the abstract syntax of Melange

can inherit (*Inheritance* operator) from a "super" language. The *Slicing* operator permits to slice a given language using slicing criteria from another one. The *Merge* operator allows language designers to merge one language into another one. The *Weaving* operator weaves aspects into a given metamodel.

A *ModelType* defines an interface to manipulate models. It consists of a set of *ObjectType*s, thereby defining a group of interrelated types. Model types can be created from scratch, or automatically inferred from a concrete metamodel. In the latter case, the metamodel explicitly references this new model type as its *exactType*. Model types are linked one another by subtyping relations: if MT' is a subtype of MT, then there is one and only one *Subtyping* instance that references MT' as its *subType* and MT as its *superType*.

A *Transformation* defines a model transformation that takes *Element*s as input and may produce an *Element* as output. This means that transformations can operate on *Metamodel*s, *ModelType*s, or *Transformation*s themselves.

Concrete Syntax Melange provides a textual editor that allows DSMLs designers to import, manipulate, and reason about DSMLs. For the sake of conciseness, we do not detail the whole grammar of the concrete syntax of Melange. Instead, we illustrate its typical use through the mere examples aforementioned to illustrate the proposed language operators. In a single Melange file, multiple languages can be designed. For instance, the *L1* and *L2* languages are designed by importing respectively the *pkg1.ecore* and *pkg2.ecore* abstract syntaxes (Lines 1 to 6). The *L3* language (Lines 7 to 9) corresponds to the merging result depicted by Figure 3.12c. *L3* inherits from *L1*, *i.e., L3* reuses the abstract syntax *pkg1.ecore*. This abstract syntax is then merged with the *L2* language to form *L3* (Line 8). The *L4* language (Lines 10 to 12) is designed by performing a positive slicing on the *L1* using as input the reference *a* of the class *A*. Similarly, *L5* is designed (Lines 13 to 15) by performing a negative slicing on *L1* using as input the class *D*. Finally, *L6* (Lines 16 to 18) inherits from *L5*. The aspect *ExecuteAspect* is then woven into *L6*.

```
1 language L1 {
2   syntax "pkg1.ecore"
3 }
4 language L2 {
5   syntax "pkg2.ecore"
6 }
7 language L3 inherits L1{
8   merges L2
9 }
10 language L4 {
11   slices+ L1 using [Pkg1.A.a]
12 }
13 language L5 {
14   slices- L1 using [Pkg1.D]
15 }
16 language L6 inherits L5{
17   with @ExecuteAspect
18 }
```

Listing 3.1: Concrete syntax of Melange

Implementation Choices The operators aforementioned for language assembly and customization can be implemented in various ways. We report here on additional choices we made in its concrete implementation within Melange. The algebra does not require a particular formalism for expressing metamodels. In our implementation, we rely on the Ecore implementation of the EMOF standard provided by EMF to specify the abstract syntax of DSMLs. Different operators for metamodel merging have been proposed in the literature (*e.g.*, [112, 62]). Interestingly, the UML2.0 specification introduces the notion of *PackageMerge* that specifies "*how the contents of one package are extended by the contents of another package*" [151]. Informally, the UML specification states that "*a resulting element will not be any less capable than it was prior to the merge*". Matching of elements of both sides mostly occurs based on name equality. When a match is found between two elements, the resulting package incorporates both sides of its definition. We choose to use a slightly improved version of the *PackageMerge* operator as defined in the UML specification and refined by Dingel *et al.* [58]. Dingel *et al.* refine the UML specification with additional constraints that enforce the unique, associative, and commutative algebraic properties of the *PackageMerge* operator. To meet our requirements, we adapt the *PackageMerge* operator by trading its UML specificities with EMOF specificities, while preserving its general spirit. For example, we do not consider the concept of *Profile* and adapt the concept of UML *Association* to the concept of EMOF *Reference*. The *PackageMerge* operator specifies a set of constraints that must be ensured for the merge to succeed. If one of the constraints is violated, the merge is ill-formed and an error is reported. It follows that operators of the algebra that rely on the abstract syntax merging operator share the same property: if the constraints are violated the operation is invalid and an error is reported to the user, otherwise the result is guaranteed to be well-formed. On the semantics part, we choose to use the Xtend programming language supplemented with annotations we developed to specify the operational semantics of DSMLs through the definition of aspects. Xtend compiles directly to Java code for a seamless integration with other artifacts generated using the EMF framework. A simple example of aspect used to weave executability in the *State* meta-class of a FSM language is given in Listing 3.2. The *_self* variable refers to the element on which the aspect is ultimately woven (a *State* object in this case) and allows the aspect to access all its features (*outgoingTransitions* in this case). Here, the *ExecutableState* aspect inserts a *step* method in the *State* meta-class to fire the appropriate transition given an input character *c*. Note that aspects may also declare new attributes that are introduced in the target meta-classes.

```
1 @Aspect(className = fsm.State)
2 class ExecutableState {
3   def void step(char c) {
4     val t = _self.outgoingTransitions
5             .findFirst[input == c]
6     if (t == null) throw new Exception
7     else t.fire
8   }
9 }
```

Listing 3.2: Weaving executability with aspects

The *@Aspect* annotation specifies the pointcut of the aspect, while the rest of the class definition defines its advice (new methods and attributes to be inserted). Since pointcuts and advices are not clearly separated, the process of re-binding a set of aspects to a new abstract syntax consists in copying the aspects while updating their pointcuts to target the appropriate concepts of the abstract syntax.

We also made the following choices in the priorities given to each operator. The inheritance operator has the highest priority, followed by the merge and slice operator (in order of appearance), ending with the aspect weaving operator. First, languages may inherit part of their definition from a super-language. As a consequence, the type system ensures that the sub-typing relation between the two languages is kept, otherwise an error is reported. Then, other artifacts may be assembled, merged or sliced on top of the inherited definition. Finally, aspect weaving comes last to support both the redefinition of imported parts and the addition of "glue code" to make the different parts fit together. As an example, when two merged languages exhibit no common subparts, a new aspect can be woven to connect them in a meaningful way by adding structural references between their abstract syntax, or by inserting some additional code to make their respective interpreters cooperate, *e.g.*, through context translation. Finally, for each language declaration, Melange infers its corresponding exact model type. The embedded model-oriented type system automatically infers the subtyping hierarchy through structural typing. This hierarchy is used to ensure the subtyping relation when inheritance is involved and is displayed to the user in a dedicated Eclipse view.

Compilation Scheme and Integration with EMF From a Melange program, such as the one depicted in Listing 3.1, the Melange compiler first reads and imports the external definitions and assembles them according to the rules of the algebra. Once the new DSMLs are assembled, customization operators are applied. Then, the compiler completes the resulting model by inferring the subtyping hierarchy among the model types inferred for each language. The implementation relations between metamodels and model types are also inferred in this phase, leading to a complete Melange model conforming to the metamodel of Figure Figure 3.14. Then, it generates a set of artifacts for each declared language: (i) an Ecore file describing its abstract syntax (ii) a set of aspects describing its semantics attached to the concepts of its abstract syntax (iii) an Ecore file describing its exact model type and (iv) an Eclipse plug-in that can be deployed as is in a new Eclipse instance to support the creation and manipulation of models conforming to it. To generate the runtime code for the new artifacts, Melange relies on the EMF compiler (a *genmodel* generating Java code from an Ecore file), and the Xtend compiler (generating Java code from the aspects file). For each language definition, the Java code generated by both compilers is associated to a plug-in. Since Melange reuse the formalism for language definition of EMF, along with its compilation chain, it is fully interoperable with the EMF ecosystem. Newly created DSMLs may thus benefit from other tools of the EMF ecosystem such as Xtext for the definition of a textual editor or Sirius for a graphical representation.

3.1.2.6 Case Study

In this section, we illustrate how the proposed operators implemented within Melange can be used by language designers to assemble legacy DSMLs. We then discuss about the results, the integration of the proposed operators in an existing language workbench, and the development overhead. All the material of the case study is provided on a companion web page available on the Melange website[9].

Language requirements To illustrate Melange, we design an executable modeling language for the Internet of Things (IoT) domain, *i.e.,*for embedded and distributed systems. This language is inspired by general purpose executable modeling languages (*e.g.*, Executable UML [133] or FUML [169]) and IoT modeling languages (*e.g.*, ThingML [75]). This language will permit the modeling of the behavior of communicating sensors built on top of resource-constrained embedded systems, such as low-power sensor and micro-controller devices (Arduino[10], Raspberry Pi[11], etc.). Such a language aims at providing abstractions and specific simulators, interpreters, or compilers depending on the targeted platforms. We consider the three following requirements while designing this language:

i) *The language has to provide an IDL (Interface Definition Language) to model the sensor interface in terms of provided service.* This motivates the reuse of structural diagrams (*e.g.*, the class diagram of (F)UML, the SysML block definition diagram [81], or MOF). All these languages provide an OO interface definition language.

ii) *The language must be able to model concurrent sensor activities.* This leads us to be inspired by process modeling languages. One can reuse the (F)UML/SysML activity diagram or BPEL/BPMN language.

iii) *The primitive actions that can be called within the activities must be expressed with a common language IoT developers are familiar with.* This aims at finding a common action language used to develop services. Such a language can be shared by a community and embedded on a set of devices used in the IoT domain. Even though the C language is the common base language of all embedded platforms, its lack of abstraction hinders its exploitation in a modeling environment. Instead, we choose the Lua language[12]. Lua is a dynamically typed language commonly used as an extension or scripting language. Lua is notably popular in the IoT domain since it is compact enough to fit on a variety of host platforms.

Language design using Melange With the aim of validating Melange, the experimental protocol consists in selecting three publicly-available implementations of existing EMF-based languages to support these three requirements. For the structural part, we use the Ecore language itself as an implementation of EMOF. EMOF provides a structural modeling close to the notion of the UML class diagram. For the behavioral part, we reuse materials from the *Model Execution Case* of the TTC'15 tool contest[13]. The case foresees the specification of the operational semantics of a subset of the UML activity diagram language with transformation languages. For the action language part, we reuse an existing implementation of the Lua language developed using Xtext. We provide an operational semantics of the Lua language using Xtend and a set of active annotations.

The new language has to provide three perspectives: i) Capturing the services offered by IoT devices, ii) Defining the behavior of these services through a model of an internal process

[9] http://melange-lang.org/sle15
[10] http://www.arduino.cc/
[11] Cf. https://www.raspberrypi.org
[12] Cf. http://www.lua.org
[13] http://www.transformation-tool-contest.eu/solutions_execution.html

```
1 language ActivityLang {
2  syntax "platform:/resource/activitydiag.ecore"
3  with OperationalSemanticsActivityAspect
4 }
5
6 language LuaLang {
7  syntax "platform:/resource/xtext/Lua.ecore"
8  with org.k3.lua.OperationalSemanticsAspect
9 }
10
11 language EcoreLang {
12  syntax "platform:/resource/Ecore.ecore"
13 }
14
15 language ActivitySlice{
16  slices+ ActivityLang using ['OpaqueAction',
17   'MergeNode','DecisionNode', 'InitialNode',
18   'JoinNode','ForkNode','ActivityFinalNode']
19 }
20
21 language ActivityEcoreLang{
22  merges ActivitySlice
23  merges EcoreLang
24  with fr.inria.diverse.glue.EOperationAspect
25 }
26
27 language ActivityELuaLang{
28  merges ActivityEcoreLang
29  merges LuaLang
30  with fr.inria.diverse.glue.ExpressionAspect
31 }
32
33 language  LuaExtensionLang{
34  syntax "platform:/resource/LuaExt.ecore"
35  with org.luaext.OperationalSemanticsAspect
36 }
37
38 language FinalLang inherits ActivityELuaLang{
39   merges LuaExtensionLang
40   with fr.inria.diverse.lua.ExpressionAspect
41 }
```

Listing 3.3: Assembling the IoT language with Melange

describing the workflow of activities, and iii) Modeling activity implementations. Each activity can execute an action defined using the Lua language. This action language is extended to integrate a new primitive to send messages containing data. These messages are used to invoke services on other devices. This language is built using the following Melange assembly definition. The definition is decomposed in multiple languages to ease the description of the process. In a real situation, this definition can be shortened.

1. The abstract syntax of the three languages (the activity diagram from TTC15, Lua, and Ecore) are imported into Melange to form languages respectively called *ActivityLang*, *LuaLang*, and *EcoreLang* (Lines 1 to 13).
2. To design the *ActivitySlice* language, *ActivityLang* is sliced to preserve only semantics of the activity diagram without the action language concept (Lines 15 to 19). To do so, we manually identified the classes of interest (Lines 16 to 18).
3. *ActivitySlice* and *EcoreLang* are merged (Lines 21 to 25). The *EOperationAspect* then

binds *EOperation* and *Activity* (Line 24). This step creates a language *ActivityEcoreLang* that enables the modeling of objects. Such objects can be for example a temperature sensor in a specific room instance with an operation *getTemperature*. The implementations of the operations are defined through activity diagram definitions.

4. *ActivityEcoreLang* and *LuaLang* are then merged to form a new language *ActivityELuaLang* (Lines 27 to 31). The *Expression* classes from both languages are bound by an aspect (Line 30).
5. A new language is designed to supplement Lua with message sending to other objects for the synchronization between several complex objects Lines 33 to 36.
6. A new language *FinalLang*, that inherits from *ActivityELuaLang*, is then created (Lines 38 to 41). *ActivityELuaLang* is merged with the *LuaExtensionLang* and provides a specific glue to link the *ActivityELuaLang* semantics with the *LuaExtension* semantics.

The proposed type system for languages is mandatory to check the whether the inheritance relations that stand between languages (*e.g.*, Line 38) can be defined. This type system also computes at run time whether a given transformation can be reused on *FinalLang*. For instance, transformations based on Ecore can be reused with *FinalLang*.

Through this experiment, we obtain a new executable modeling language resulting from the composition of three legacy languages not originally built to be composable. The Melange composition script uses all the operators provided by Melange (*Slice*, *Merge*, *Inherits*). This experiment shows that a language designer can create a new language with its abstract syntax and its semantics through the integration of three fragments of non-composable languages.

Discussion A critical point concerns the ability of Melange to be integrated into an existing ecosystem. The integration using Melange of three existing EMF languages allows a language designer to obtain a new EMF language. If we do not consider the imposed methodology for defining the language semantics (the use of the interpreter pattern [83]), no modification on these languages was required to support that composition. This illustrates how Melange can be integrated into an existing language workbench without any change on the handled legacy abstract syntaxes. All the Melange operator are used for this case study. This does not guarantee that these operators are sufficient but it highlights that all of them are required when a language designer needs to compose existing languages.

Another major point is the overhead in term of performance and lines of code that stem from the use of Melange. Compared to a top-down approach in which we design our executable modeling language for IoT, we observe no additional concepts integrated within the abstract syntax definition. At the semantics level, a glue is injected for the implicit conversion of the interpreter pattern context resulting from the composition of the various contexts stemming from various operational semantics. At run time, no additional cost in terms of performance were observed to the use of the language resulting from the composition. The following Table sums up this result.

	Melange	Top-down
Metaclasses (#)	104	104
LoC for the glue (#)	27	0
Efficiency (sec)	30,0	25,9

Performance comparison is obtained in loading and executing a model with 10 objects that contains one operation with a workflow with 1000 basic actions that do mainly 10 numeric operations. The comparison was done on the same laptop designed with an Intel i7 with 16Gb of memory, a Linux 64bit operating system and an Oracle Java 8 virtual machine.

Threats to validity First, all the languages must be designed in the same technical ecosystem. Melange does not provide any support for integrating heterogeneous languages in terms of technical ecosystem. Second, Melange can not compose all the language semantics. Composition can be done if and only if the semantics is operational and defined in following the interpreter pattern. This pattern can be, for instance, implemented using static introduction, in modifying the code generated from the abstract syntax, or in combining it using a visitor pattern. The interpreter pattern can be used to create a language interpreter or a code generator. However, we can only compose operational semantics that are defined using this pattern. Third, the slice operator is often used before the merge operator to solve composition conflicts. It can be seen as a manual task to align language before composition. Finally, the same person implemented the language using Melange and using a traditional top-down approach. This person is an expert in language design and modeling technologies. Besides the top-down language design has been reviewed by three experts in language design from the research team and is publicly available on Github.

3.1.2.7 Related Work

A DSML allows developing software for a particular application domain quickly and effectively, yielding programs that are easy to understand, reason about, and maintain [97]. There may be, however, a significant overhead in creating the infrastructure needed to support a DSML. Numerous works proposed to create reusable and composable language units to tackle this issue. Methodologies have been proposed for building DSMLs embedded within an existing, higher-order, and typed programming language [96]. Techniques have been then designed for building modular interpreters and tools for such embedded DSMLs. Different techniques have been studied for addressing the challenge of language extension and composition, such as projectional editing [196]. Spoofax, however, relies on meta-languages for defining syntaxes and semantics, which are inherently modular and composable [200]. Although basic import mechanisms are supported, they usually lack a powerful support for customization. More recently, an overview of the support provided by language workbenches has been provided [66]. In the grammar world, several techniques demonstrated the possibility to create language units using attribute grammars [105, 165, 138]. MontiCore applied modularity concepts for designing new DSMLs by extending an existing one, or by composing other DSMLs [116]. MontiCore reifies as a first-class object the concept of language inheritance to allow language feature reuse. Other works propose to leverage concepts from the component-based software engineering community to modularly develop DSMLs [189, 211].

In the MDE domain, several meta-tooling platforms propose mechanisms for improving language design modularity. Ledeczi *et al.* propose to compose domain-specific design environments using MDE technologies [123]. Melusine [70], Xtext [71], or MPS [1] are frameworks supplemented with IDEs for building textual DSMLs. In both the MDE and grammar domains, the increasing trend to create new DSMLs, from scratch or by adapting existing ones, causes the emergence of families of DSMLs. A family of DSMLs is a set of DSMLs sharing common aspects but specialized for a particular purpose. The emergence of a family of DSMLs raises the need to reuse common tools among a given family [117, 113] and the need to create language composable units. To ease the language unit composition, Steel *et al.* [180] and De Lara *et al.* [50] propose to define a clear contract and a typing system that can be used for composing language units. De Lara *et al.* present the *concept* mechanism, along with *model templates* and *mixin layers* leveraged from generic programming to MDE [48]. *Concepts* are close to model types [180] as they define the requirements a metamodel must fulfill for its models to be processed by a transformation, under the form of a set of classes. Sánchez, Wimmer *et al.* go further than strict structural mapping by renaming, mapping, and filtering metamodel elements [185, 206].

Erdwel *et al.* proposed a taxonomy to ease the positioning of approach related to language composition [64]. According to this classification, our algebra supports the language extension, restriction, and unification operators. Additionally, we do not consider that restriction is only a matter of additional validation rules. Instead, we prune the language from the unwanted parts so that only the necessary concepts are kept.

3.1.3 Variability Management in Language Family

The content of this subsection is an adapted excerpt from the following publications:

Edoardo Vacchi, Walter Cazzola, Benoit Combemale, and Mathieu Acher. Automating Variability Model Inference for Component-Based Language Implementations. In Patrick Heymans and Julia Rubin, editors, *18th International Software Product Line Conference (SPLC'14)*, Florence, Italie, September 2014. ACM. [190]

Edoardo Vacchi, Walter Cazzola, Suresh Pillay, and Benoit Combemale. Variability Support in Domain-Specific Language Development. In Richard F. Paige Martin Erwig and Eric van Wyk, editors, *6th International Conference on Software Language Engineering (SLE 2013)*, Lecture Notes in Computer Science, Indianapolis, USA, 2013. Springer. [191]

DSMLs are commonly developed from scratch, even when they share some concepts and even though they could share bits of tool support. This cost can be reduced by employing modern modular programming techniques that foster code reuse. However, selecting and composing these modules is often only within the reach of a skilled language engineer.

We propose to combine modular language development and variability management, with the objective of capitalizing on existing assets. This approach explicitly models the dependencies between language components, thereby allowing a domain expert to configure a desired DSML, and automatically derive its implementation. The approach is tool supported, using Kermeta to implement language components, and the Common Variability Language (CVL) for managing the variability and automating the configuration. More specifically, the approach follows the stepwise methodology illustrated in Figure 3.15:

1. Breaking down the family of DSLs into language components
2. Variability Modeling
3. Language unit composition

In this context, in [191, 190] we describe

1. a method to extract structured information from the set of existing assets in the form of a graph of dependencies,
2. a strategy to construct a variability model using the extracted information,
3. an implementation of a derivation operator to generate the language implementation from the VM automatically,

thereby facilitating the collaboration between the language developer and the domain expert, to the extent that the domain expert becomes autonomous in extracting a desired language. The implementation of this approach will be demonstrated using a real working toolset applied to a family of state machine languages.

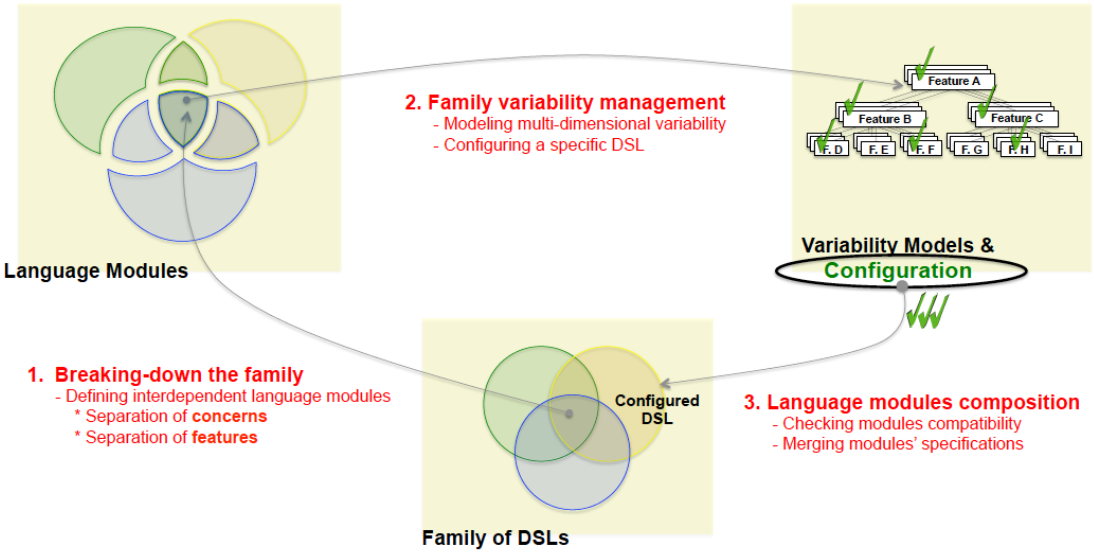


Figure 3.15: Approach overview for variability management in DSMLs

3.1.3.1 Approach Overview

Each component that we add to a language usually has some dependencies, such as a semantic concept, a syntactic requirement, or both of them. For instance, if we want some looping construct to terminate, be it `for`, `while`, or whichever we may pick, we might as well include some concept of *truth value* and the idea of a *condition* to test. Likewise, we would need some syntax to express this concept. Similarly, there might be concepts that, together, in the same language may *conflict*. For instance, we cannot have a three-valued logic and the simple boolean logic to just *coexist* in the same places: what if the condition of a loop evaluates to `null`? Should the loop exit or not?

Component-based language development is close to providing people with an easy way to implement a language by just selecting components, but implicit dependencies and conflicts between them create a barrier to opening such development to a wider audience. The challenge lies in the fact that an in-depth knowledge of how the components are designed is required prior to using such an approach. Applying variability modeling to a modular language framework allows the explicit modeling of the relations between components in a manner understandable to the domain expert or end-user.

In our approach, component-based development is necessary for users to be able to selectively pick components; the feature model is necessary to represent how components may interact and to relieve users from the burden of satisfying complicated dependencies by hand. The variability model explicitly represents the constraints and the resolution model complies with these constraints, so that the result of the derivation is guaranteed to behave as expected. Typically a variability model is used to represent a family of products; in our case we will use it to represent a *language family*, that is a set of languages that share a common set of features. In a perfect world, language components would be developed from scratch, with the target variability model in mind, and therefore they would be guaranteed to compose well together. However, implementing a language from the ground requires a substantial investment. To minimize cost during component-based language development, one approach would be to maximize reuse of a set of already available language components.

We will focus on the case of Kermeta and CVL, but the approach that we present can be applied to any kind of feature modeling approach and any componentized language development tool that will fit our framework. In particular, the main requirement for the language framework is to support a way to define the language constructs in separate components. The global approach is a two-level process: first, the reusable language components are capitalized and their possible combinations are captured in a variability model. Second, the variability model is used to select an expected set of features (or *configuration*) from which a woven model is produced by composition of the suitable reusable language components.

From a methodological perspective, we also distinguish two roles for users of our approach:

- **Language Engineer**. A person experienced in the field of DSML implementation, and who knows how to break down a language into components.
- **Domain Expert**. A person that knows the concepts and the lexicon of the target domain. People in this category would also be users of the language.

In practice, the overall approach presented in Figure 3.15 is divided into the following six steps [191]:

1. the *language engineer* collects all of the available language components: these could be pre-existing components or newly created components.
2. the relations between components are extracted automatically and represented as a *dependency graph*
3. the *language engineer* and the *domain expert* collaborate to define a variability model using the dependency graph as a guide, in such a way as to define a language family most relevant for the given domain.
4. the *domain expert* becomes autonomous: it is now possible to extract a desired language by resolving the variability (selecting a set of features).
5. using a derivation operator, a list of composition directives is derived from the resolution of the variability model
6. the language development tool generates a complete interpreter/compiler for the desired language.

In our case, if Kermeta is the language framework, and CVL is the variability language, then we will implement the reusable language components (`aspects`) using Kermeta and extract the dependency graph from Kermeta; the specification of the variability (called *variability abstract model*) will use the choice diagram proposed by CVL; variability resolution will be CVL's *resolution model*; the composition directives (the `language` descriptor) will be derived using a dedicated derivation operator, implemented using the CVL opaque variation point (and included in the *variability realization model*); finally Melange will compose the aspects contained in the language descriptor.

Note that designing variability models in this context would be challenging since it requires not only a good understanding of these frameworks and the way components interact, but also an adequate familiarity with the problem domain. We propose an approach to automatically infer a relevant variability model from a collection of already implemented language components, given a structured, but general representation of the domain [190]. We describe techniques to assist users in achieving a better understanding of the relationships between language components, and find out which languages can be derived from them with respect to the given domain.

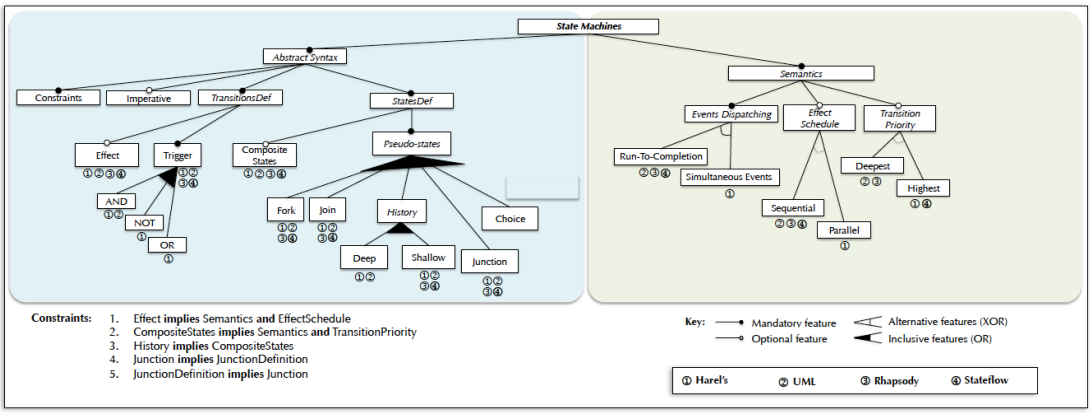


Figure 3.16: Feature model of the family of DSMLs for Finite State Machines

3.1.3.2 Case study: A family of DSMLs for Finite State Machines

In this section we illustrate the applicability of our approach for a family of DSMLs for state machines. The family considered in our experiments is inspired from the work of Crane et al. [45]. Hence, the family includes UML state diagrams [151], Rhapsody [89], and Harel's state charts [90]. In addition, we include Mathworks' Stateflow[14].

After the implementation of the various languages constructs using the approach presented in Chapter 2, we use the aforementioned approach to manage the variability among them. Figure 3.16 shown the resulting feature model for the proposed family of DSMLs for state machines. Once defined, we developed a tool which allows the domain expert to configure its own language, and generate the Melange specification that will eventually produce the expected DSML implementation (cf. Figure 3.17).

3.2 Globalization of Modeling Languages: Facing the Multiplication of Stakeholders

3.2.1 The Grand Challenge

The content of this subsection is an adapted excerpt from the following publication:

Benoit Combemale, Julien Deantoni, Benoit Baudry, Robert France, Jean-Marc Jézéquel, and Jeff Gray. Globalizing Modeling Languages. *Computer*, pages 68–71, June 2014. [37]

The development of modern complex software-intensive systems often involves the use of multiple DSMLs that capture different system aspects. In addition, models of the system aspects are seldom manipulated independently of each other. System engineers are thus faced with the difficult task of relating information presented in different models. For example, a system engineer may need to analyze a system property that requires information scattered in models expressed in different DSMLs. Current DSML development workbenches provide good support for developing independent DSMLs, but provide little or no support for integrated use of multiple DSMLs. The lack of support for explicitly relating concepts expressed in different DSMLs makes it very difficult for developers to reason about information spread across different models.

[14] Cf. http://www.mathworks.com/products/stateflow

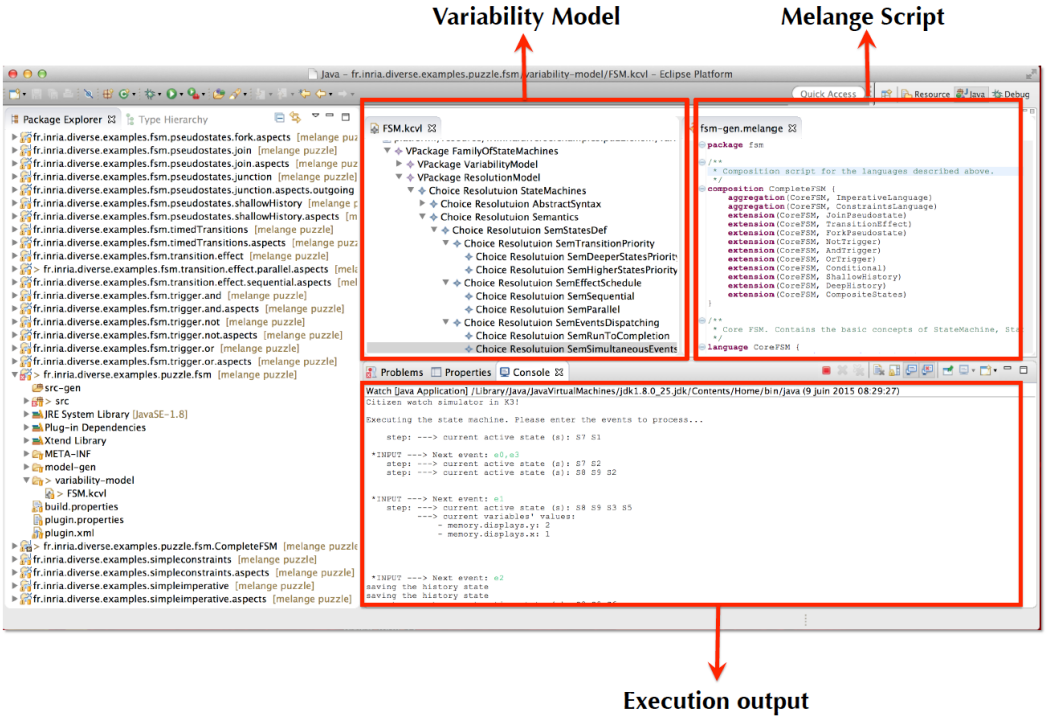


Figure 3.17: Screenshot of the tool for DSML Variability Management

3.2.1.1 Globalized DSML Challenge: Looking Ahead

Past research on modeling languages focused on their use to bridge wide problem-implementation gaps. A new generation of complex software-intensive systems, for example, smart health, smart grid, building energy management, and intelligent transportation systems, presents new opportunities for leveraging modeling languages. The development of these systems requires expertise in a variety of domains. Consequently, different types of stakeholders (e.g., scientists, engineers and end-users) must work in a coordinated manner on various aspects of the system across multiple development phases. DSMLs can be used to support the work of domain experts who focus on a specific system aspect, but they can also provide the means for coordinating work across teams specializing in different aspects and across development phases.

Supporting coordinated use of DSMLs leads to what we call the globalization of modeling languages, that is, the use of multiple modeling languages to support coordinated development of diverse aspects of a system. One can make an analogy with world globalization in which relationships are established between sovereign countries to regulate interactions (e.g., travel and commerce related interactions) while preserving each country's independent existence. The term "DSML globalization" is used to highlight the desire that DSMLs developed in an independent manner to meet the specific needs of domain experts, should also have an associated framework that regulates interactions needed to support collaboration and work coordination across different system domains.

Globalized DSMLs aim to support the following critical aspects of developing complex systems: communication across teams working on different aspects, coordination of work across the teams, and control of the teams to ensure product quality.

In the globalized DSML vision, integrated DSMLs support teams working on systems that span many domains and concerns to determine how their work on a particular aspect influences work on other concerns. The objective is to offer support for communicating relevant information, and for coordinating development activities and associated technologies within and across teams, in addition to providing support for imposing control over development artifacts produced by multiple teams.

Coordination and related separation of concerns issues have been the focus of software engineering since early work on modularizing software. For example, Parnas' use of the term "work product" to denote a module that can be the source of independent development is also a focus of team demarcation across design and implementation tasks. Modularity in modern software-intensive systems development leads to well-known coordination problems, such as problems associated with coordinating work over temporal, geographic or socio-cultural distance [4]. This has also led to the recognition of socio-technical coordination, including coordination of the stakeholders and the technologies they use to perform their development work, as a major system development challenge [5].

In this context, DSMLs can be used to support socio-technical coordination by providing the means for stakeholders to bridge the gap between how they perceive a problem and its solution, and the programming technologies used to implement a solution. DSMLs also support coordination of work across multiple teams when they are supported by mechanisms for specifying and managing their interactions. In particular, proper support for coordinated use of DSMLs leads to language-based support for social translucence, where the relationships between DSMLs are used to extract the information needed to make a team working on a system aspect aware of the relevant work performed by teams working on other aspects. Such awareness is needed to minimize the counter-productive form of social isolation that can occur when work is distributed across different teams.

3.2.1.2 On the Globalization of Modeling Languages

To support globalization, the use of multiple heterogeneous modeling languages will need to be related to determine how different aspects of a system influence each other. We identify three forms of relationships among modeling languages that can be used to support interactions across different system aspects: interoperability, collaboration, and composition.

Interoperable modeling languages provide support for the exchange of information across models expressed in the languages. Interoperable DSMLs can be developed in a relatively independent manner, but relationships defined across the different DSMLs allow information expressed in one model to be related to information contained in models expressed in different DSMLs. These DSML relationships facilitate the development of integrated modeling tool chains in which information from a model built for a specific purpose (e.g., a SysML model used to describe system architecture) is used to decorate a model that serves a different purpose (e.g., a Generalized Stochastic Petri Net used for performance analysis). Interoperable DSMLs have the lowest coupling of the three relationships we identified; the focus is on supporting coordinated use of modeling tools, as opposed to tightly coupling the development of the different models.

Collaboration relationships among modeling languages are used to support coupled development of models. DSMLs in a collaboration relationship are referred to as collaborative modeling languages. The development of a model expressed in a collaborating modeling language can directly influence the form of models created using other collaborating modeling languages. For example, consistency relationships defined across the DSMLs can be used by developers to ensure that the different models they create are consistent with each other. Model authoring tools for collaborating DSMLs are thus coupled. Collaborating DSMLs can be used to support a priori as well as a posteriori global analysis of properties.

Interoperable and collaborating DSMLs support DSML interactions without deriving new forms of information from information spread across different models. However, there are situations that call for combining information scattered in other models to create new forms; for example, to support generation of system documentation, test cases, or to provide support for simulating global system behavior. Model composition (e.g., weaving and merging) is thus the third form of interaction that is facilitated by explicit definitions of relationships across elements in different DSMLs.

The prior discussion focused on the use of DSMLs to coordinate the work of developers. These ideas can be applied at various development life-cycle phases, ranging from early analysis to system runtime. Models can also be used to coordinate work done by different components, subsystems, or services. The use of DSMLs to coordinate work can potentially have a beneficial impact on the management of running systems. Different types of models are currently being used as runtime abstraction layers to support reasoning about the system or even adapting it [6]. Explicitly defined relationships across DSMLs for runtime models can be leveraged by these model-based runtime environments to coordinate the manipulation of models at runtime.

Challenging issues will need to be addressed to realize the above forms of language integration. Relationships among the languages will need to be explicitly defined in a form that corresponding tools can use to realize the desired interactions. Requirements for tool manipulation are thus another topic that will be a focus for future work in the area of DSML globalization.

This grand challenge is currently supported by the GEMOC initiative (see http://gemoc.org). GEMOC is an open international initiative that brings together a community to develop software language engineering breakthroughs that support interoperable, collaborative, and composable modeling languages. The GEMOC initiative provides a framework that facilitates collaborative work on the globalization of DSMLs.

3.2.2 Event Scheduling as Behavioral Language Interface

The content of this subsection is an adapted excerpt from the following publication:

Matias Ezequiel Vara Larsen, Julien Deantoni, Benoit Combemale, and Frédéric Mallet. A Behavioral Coordination Operator Language (BCOoL). In *18th International Conference on Model Driven Engineering Languages and Systems (MODELS 2015)*, August 2015. [194]

Some coordination languages deal with the complexity of model behaviors by treating models as black boxes encapsulated within the boundary of an interface. A model behavioral interface gives a partial representation of the model behavior therefore easing the coordination of behavioral models. However, it is not uniquely defined and may vary depending on approaches. For instance, in *Opus* [26], the interface is a list of methods provided by the model. Other approaches abstract away the non-relevant parts of the behavior of models as events [207] (also named signals [125]). These approaches focus on events and how they are related to each other through causal, timed or synchronization relationships. Following the same idea, *control-driven* coordination languages rely on a model behavioral interface made of explicit events [174, 69, 10]. While in Rapide [174], the interface is only a set of events acceptable by the model, some other approaches go further and also exhibit a part of the internal concurrency. For instance Barroca *et al.* [10] propose an interface that contains services and events, but also properties that express requirements on the behavior of the components. Such requirements act as a contract and can be checked during the coordination to ensure a correct behavior. In these approaches, the model behavioral interface provides information to coordinate the behavior of a model. In particular, in event-driven coordination approaches events act as “coordination points" and exhibit what can be coordinated. This gives a support for control and timed coordination while remaining independent of the internal model implementation. Moreover, event-driven coordinations are

non intrusive; *i.e.,*models can be coordinated without any change to their implementation, thus ensuring a complete separation between the coordination and the computational concerns. Several causal representations from the concurrency theory are used to capture event-based behavioral interface. A causal representation captures the concurrency, dependency and conflict relationships among actions in a particular program. For instance, an event structure [207] is a partial order of events, which specifies the, possibly timed, causality relations as well as conflict relations (*i.e.,*exclusion relations) between actions of a concurrent system. This fundamental model is powerful because it totally abstracts data and program structure to focus on the partial ordering of actions. It specifies, *in extension* and *in order*, the set of actions that can be observed during the program execution. An event structure can also be specified *in intention* to represent the set of observable event structures during an execution (see *e.g.*, [4] or [14]).

In our approach, to capture the specification of coordination patterns between languages, we require a behavioral interface, but at the language level. A language behavioral interface must abstract the behavioral semantics of a language, thus providing only the information required to coordinate it, *i.e.,*a partial representation of concurrency and time-related aspects. Furthermore, to avoid altering the coordinated language semantics, the specification of coordination patterns between languages should be non intrusive, *i.e.,*it should keep separated the coordination and the computation concerns. In Chapter 2.4, elements of event structures are reified at the language level to propose a behavioral interface based on sets of *event types* and *contraints* [38]. Event types (named DSE for Domain Specific Event) are defined in the context of a metaclass of the abstract syntax (AS), and abstract the relevant semantic actions. Jointly with the DSE, related constraints give a symbolic (intentional) representation of an event structure. With such an interface, the concurrency and time-related aspects of the language behavioral semantics are explicitly exposed and the coordination is event-driven and non intrusive.

Then, for each model conforming to the language, the model behavioral interface is a specification, in intention, of an event structure whose events (named MSE for Model Specific Event) are instances of the DSE defined in the language interface. While DSEs are attached to a metaclass, MSEs are linked to one of its instances. The causality and conflict relations of the event structure are a model-specific unfolding of the constraints specified in the language behavioral interface. Just like event structures were initially introduced to unfold the execution of Petri nets, we use them here to unfold the execution of models.

We propose to use DSE as “coordination points" to drive the execution of languages. These events are used as handles or control points in two complementary ways: to observe what happens inside the model, and to control what is allowed to happen or not. When required by the coordination, constraints are used to forbid or delay some event occurrences. Forbidding occurrences reduces what can be done by individual models. When several executions are allowed (nondeterminism), it gives some freedom to individual semantics for making their own choices. All this put together makes the DSE suitable to drive coordinated simulations without being intrusive in the models. Coordination patterns are captured as constraints at the language level on the DSE.

To illustrate the approach, we introduce a simple state-based language named Timed Finite State Machine (TFSM) and its behavioral interface; a state machine language augmented with timed transitions (see Figure 3.18). The metamodel describes the abstract syntax of the TFSM language (see Figure 3.18). A *System* is composed of *TFSMs*, global *FSMEvents* and global *FSMClocks*. Each *TFSM* is composed of *States*. Each state can be the source of outgoing guarded *Transitions*. A guard can be specified either by the reception of an *FSMEvent* (*EventGuard*) or by a duration relative to the entry time in the source state of the transition (*TemporalGuard*). When fired, transitions generate a set of simultaneous *FSMEvent* occurrences.

The TFSM language defines the following DSE: *entering* and *leaving* a state, *firing* a

transition, the occurrences (*occurs*) of a FSMEvent and the *ticks* of a FSMClock (see at the top of Figure 3.18). These DSE are part of the language behavioral interface of TFSM. DSE are defined by using a specific language named ECL (standing for Event Constraint Language [54]) which is an extension of OCL [153] with events. ECL takes benefits from the OCL query language and its possibility to augment an abstract syntax with additional attributes (without any side effects). Consequently by using ECL, it is possible to augment AS metaclasses and add DSE. A partial ECL specification of TFSM is shown in Listing 3.4 where the DSE *entering* and *leaving* are defined in the context of State (Listing 3.4: line 6) while *occurs* is defined in the context of FSMEvent (Listing 3.4: line 4). When a metaclass is instantiated, the corresponding DSE are instantiated; *e.g.*, for each instance of the metaclass *State*, DSE *entering* is instantiated. Each instance of DSE is a MSE. In the case of TFSM, since two States are instantiated (*S1* and *S2*), there are two MSE of *entering*: *S1_ entering* and *S2_ entering*. All MSE are part of the model behavioral interface.

Listing 3.4: Partial ECL specification of TFSM

```
1 package tfsm
2 context FSMClock
3   def: ticks : Event = self
4 context FSMEvent
5   def: occurs : Event = self
6 context State
7   def : entering : Event = self
8   def : leaving : Event = self
```

In the following section, the TFSM language together with its behavioral interface is used to introduce our language B-COoL

3.2.3 B-COoL: a Meta Language for Behavioral Coordination of Modeling Languages

The content of this subsection is an adapted excerpt from the following publication:

Matias Ezequiel Vara Larsen, Julien Deantoni, Benoit Combemale, and Frédéric Mallet. A Behavioral Coordination Operator Language (BCOoL). In *18th International Conference on Model Driven Engineering Languages and Systems (MODELS 2015)*, August 2015. [194]

3.2.3.1 B-COoL Overview

B-COoL is a dedicated (meta)language to explicitly capture the knowledge about system integration. With B-COoL, an integrator can explicitly capture coordination patterns at the language level. Specific *operators* are provided to build the coordination patterns and specify how the DSE of different language behavioral interfaces are combined and interact. From the B-COoL specification, we generate an executable and formal coordination model by instantiating all the constraints on each and every instance of DSE. Therefore, the generated coordination model implements the coordination patterns defined at the language level.

The design of B-COoL is inspired by current structural composition languages (*e.g.*, [112, 74]). These approaches rely on the *matching* and *merging* phases of syntactic model elements. A matching rule specifies what elements from different models are selected. A merging rule specifies how the selected model elements are composed. In these approaches the specification is at the language level, but the application is between models. Similarly, a B-COoL operator relies on a *correspondence matching* and a *coordination rule*. The correspondence matching identifies what elements from the behavioral interfaces (*i.e.*, what instances of DSE) must be selected. The merging phase is replaced by a coordination rule. While in the structural case the merging operates on the syntax, the coordination rule operates on elements of the semantics

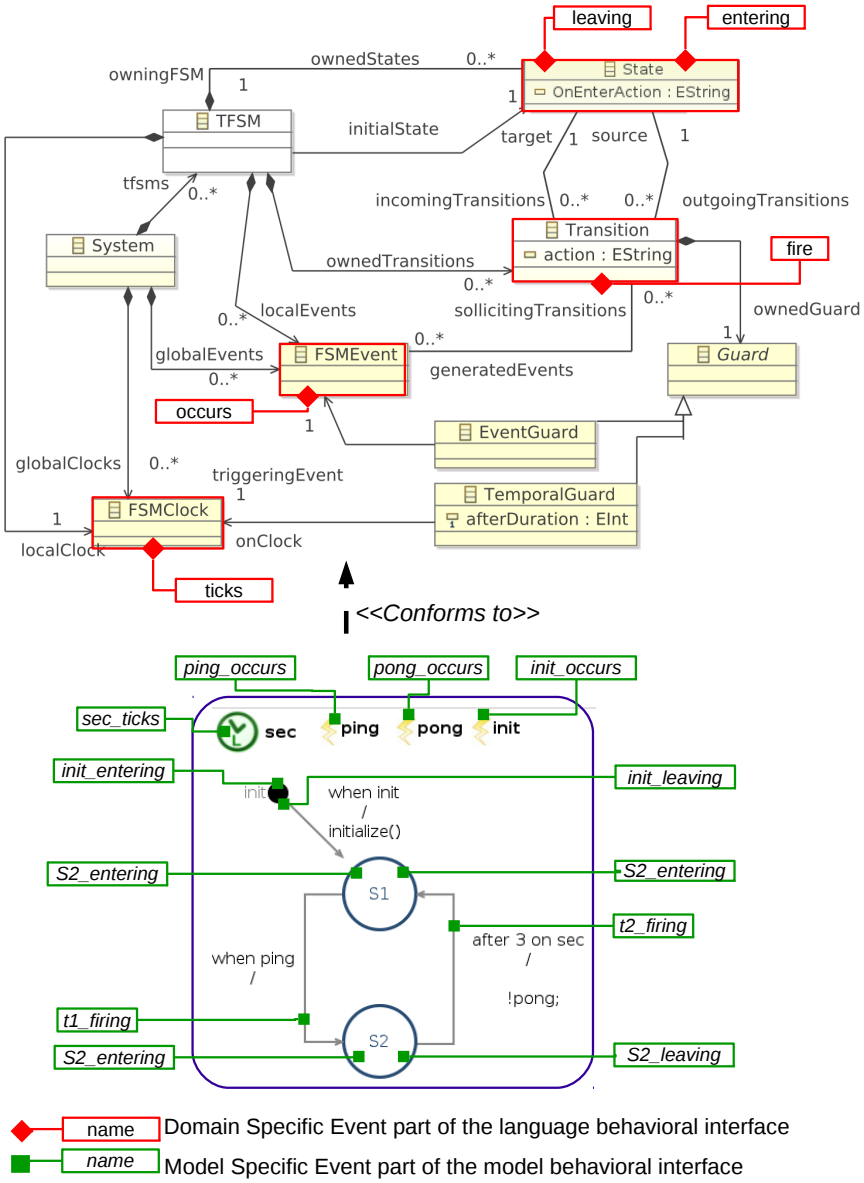


Figure 3.18: A TFSM metamodel (top) and a conforming model (bottom) with their respective behavioral interfaces

(*i.e.*, instances of DSE). Thus, coordination rules specify the, possibly timed, synchronizations and causality relationships between the instances of DSE selected during the matching.

We illustrate the use of B-COoL through a (simple) running example: *i.e.*, building the synchronized product of TFSM. This is a very classical "coordination" operation on automata with frequent references in the literature [6]. The goal here is to show that we can build this operator and use it off-the-shelf when needed. It is informally defined as follows: When coordinating two state machines, all events belonging to both state machines must be synchronized using a

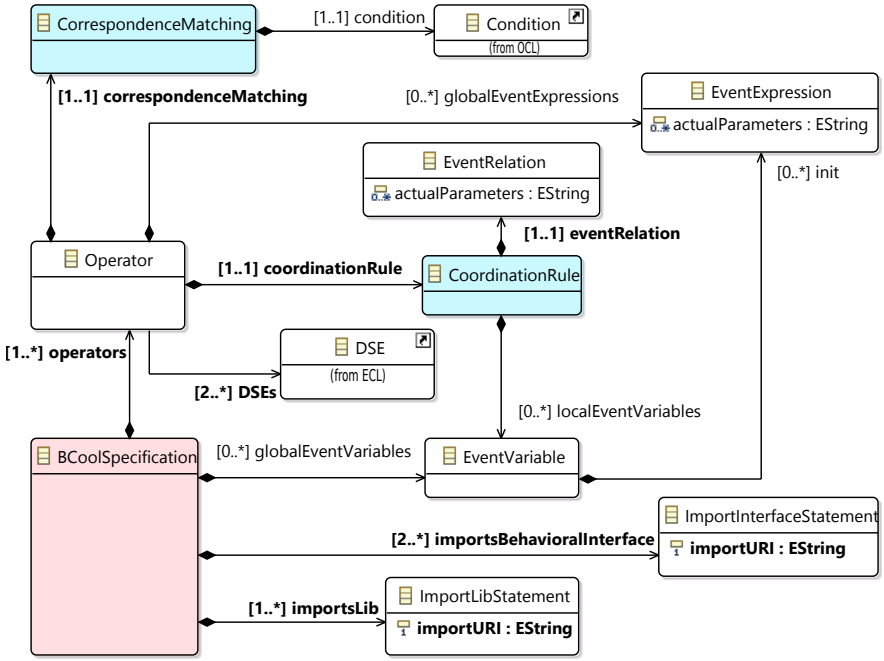


Figure 3.19: Excerpt of the B-COoL metamodel

"rendez-vous". All the other events, belonging to only one state machine, can occur freely at any time. The synchronized product is defined for any state machine, at the language (metamodel) level. When applied it concerns two specific state machines, at the instance (model) level. Note that this first behavioral coordination pattern is homogeneous (*i.e.,* it involves a single language). Examples of heterogeneous coordination patterns (*i.e.,* that involve several languages) are provided in the related publication **??**.

In the following, we first present the abstract syntax, and then, the execution semantics of B-COoL. We finish this section by showing the language workbench of B-COoL which is implemented as part of the Gemoc studio. We use the studio on the running example, and we generate the coordination model between two particular TFSMs. We then show how the generated coordination model can be executed.

3.2.3.2 Abstract Syntax of B-COoL

The main element of B-COoL (see Figure 3.19) is a *BCoolSpecification* that contains language behavioral interfaces (*importsInterfaceStatements*) and *Operators*. The specification must import at least two language behavioral interfaces. Interfaces provide the *DSE* needed for the coordination. The imported DSEs serve as parameters for the operators. Then, an operator specifies what instances of these DSE are selected and how they are coordinated (the *DSEs* reference). For instance, to build the synchronized product of TFSM, we need to synchronize FSMEvents. This is done by coordinating the instances of DSE *occurs* (see Figure 3.18). First, the language behavioral interface of TFSM is imported. Then, an operator is defined with *occurs* as a parameter.

Each operator contains both a *correspondenceMatching* and a *coordinationRule*. The former relies on a Boolean *Condition* defined as an OCL expression. It acts as a precondition for the coordination rule, *i.e.,* it is a predicate that defines when the coordination rule must be applied

to the given parameters. To specify the predicate, it is possible to navigate through the context of the DSE and query a specific element used within the Boolean expression. For instance, for the synchronized product, the condition selects an instance of DSE *occurs* by looking at its attribute *name*.

The *coordinationRule* specifies how the selected instances of DSE must be coordinated. To do so, the user must define some *EventVariables* (*localEventVariables*) and an *EventRelation*.

An event variable can be either defined locally within the operator or globally for the whole specification (*globalEventVariables*). These variables either define global events used across different operators, or create a new event from the selected instances of DSE and possibly from attributes of the input models. The definition of these events is made by using an *EventExpression*. An event expression returns a new event from a given parameter. For instance, this can be used to select only some occurrences of a DSE instance, thus allowing the implementation of filters. An event expression can also be used to join in a single event the occurrences of different events (union). When used in the coordination rule, the resulting events can be used as parameters of event relations, constraining by transitivity (some of) the occurrences of DSE instances.

How the selected events are coordinated is determined by event relations that restrict the occurrences of the events on which it is applied. The actual parameters of the event relation can be some instances of DSE and/or some *EventVariables*. For instance, the synchronized product specifies a strong synchronization. Thus, the coordination rule uses a "rendez-vous" relation between the selected instances of DSE *occurs*. As a result, all the occurrences of these events are force to happen simultaneously.

In B-COoL, the definition of event expressions and relations is made in dedicated libraries, which must be imported. This is further explained in the following subsection.

3.2.3.3 B-COoL library

Libraries gather some predefined event expressions and relations, which must be imported by the specification (*ImportedLibStatement* in Figure 3.19). Libraries can be organized by modeling domains to gather all the relevant operators.

A library is a set of declarations together with their formal parameters. A library also contains some definitions, which give the actual behavior of the declarations. Declarations are referenced in a B-COoL specification. Generally speaking, event expressions create a new event from their parameters (*e.g.*, building the *Union*, or the *Intersection* of its parameters). They can be used to filter some occurrences of existing events. Such constraints are used in B-COoL either to provide global events used in different operators or to define some filters used in the coordination rules. Relations, however, constrain the evolution of the events given as formal parameters. For instance, a relation can define a *Rendezvous* synchronization on its parameters. Lots of other relations, more or less complex can be defined (*e.g.*, *Causality*, *FIFO* or ad-hoc relations for specific protocols).

Currently, B-COoL includes a library, named *facilities.bcoollib*, that provides all the declarations used in all the following examples. The integrator however can extend the current library by defining new specific constraints depending on its problems and domain. The definition part of B-COoL is common to the one of CCSL [4], a formal language dedicated to event constraints. As a result, when building B-COoL operators a CCSL specification is produced [193]. We can then use CCSL tool (TimeSquare [55]) to analyze and execute the generated coordination specification. This is further discussed in Section 3.2.3.4. We could also use another language to build the semantics of operators and then take benefit from other analysis tools.

3.2.3.4 Concrete Syntax

B-COoL is a set of plugins for Eclipse[15] as part of the GEMOC studio[16]; which integrates technologies based on Eclipse Modeling Framework (EMF) adequate for the specification of executable domain specific modeling languages. B-COoL is itself based on the EMF and its abstract syntax has been developed using Ecore (*i.e.,* the meta-language associated with EMF). The textual concrete syntax of B-COoL has been developed by using Xtext[17] which provides advanced editing facilities. To introduce the concrete syntax, we describe the running example in B-COoL (see Listing 3.5).

The specification begins by importing twice the ECL specification of the TFSM language (Listing 3.5: lines 3 and 4). In B-COoL, the number of language behavioral interfaces must correspond to the number of accepted models. Since the operator is specified between two models both conforming to the TFSM language, the ECL specification is imported twice and named *tfsmA* and *tfsmB*. Then, we define a new operator named *SyncProduct* (Listing 3.5: line 6) to select and coordinate instances of DSE *occurs*. Pairs of instances of these events are selected by comparing the attribute name defined in the context of FSMEvent. In Listing 3.5: line 7, the instances of DSE mapped as *dse1* and *dse2* are queried to get attribute name. Then, the attributes are used as operands for a boolean condition. When the two instances of DSE have the same name, the pairs are selected and the coordination rule is applied. The selected instances are synchronized with a *Rendezvous* relation (Listing 3.5: line 8). This results in forcing a simultaneous occurrence of the two events.

We use the specification presented in Listing 3.5 to generate the coordination specification between two particular TFSM models: *TFSMA* and *TFSMB* (Figure 3.20). The workbench is then used to execute this coordination specification. Figure 3.20 illustrates the partial timing output of the execution of the whole example. The workbench also offers the possibility to obtain by exploration quantitative results on the scheduling state-space. A video presenting the whole flow (compilation, execution, diagram animation, state-space exploration) can be found on the companion the webpage[18].

To validate our approach, we present in the following section the development of some B-COoL coordination operators. We then use these operators to generate the coordination model for a video surveillance system. The workbench is finally used to execute and validate the result.

Listing 3.5: B-COoL specification of synchronized product between TFSM models

```
1 BCOoLSpec ProductTfsmAndTfsm
2 ImportLib "facilities.bcoollib"
3 ImportInterface "TFSM.ecl" as tfsmA
4 ImportInterface "TFSM.ecl" as tfsmB
5
6 Operator SyncProduct(dse1:tfsmA::occurs,dse2:tfsmB::occurs)
7     CorrespondenceMatching: when(dse1.name = dse2.name)
8     CoordinationRule: RendezVous(dse1, dse2)
9 end operator
```

3.2.3.5 Execution semantics

In this section we give a rough description of the execution semantics of B-COoL, *i.e.,* how a B-COoL specification is used to obtain a coordination model. The detailed semantics is available in [192].

[15] http://www.eclipse.org
[16] http://gemoc.org/studio/
[17] http://www.eclipse.org/Xtext/
[18] http://timesquare.inria.fr/BCOoL

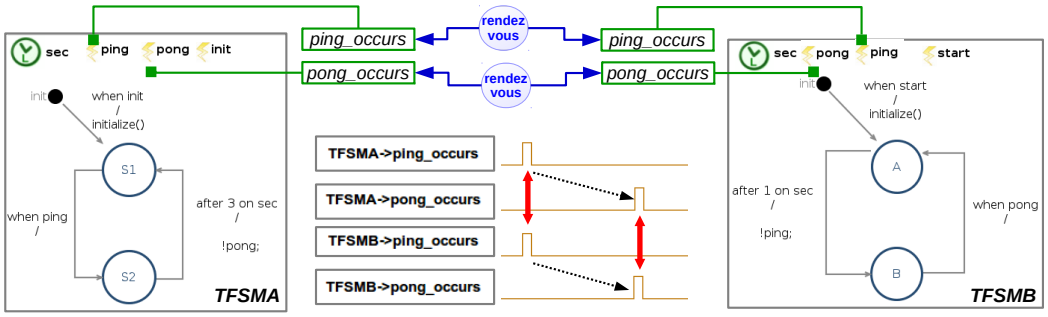


Figure 3.20: Application of the synchronized product operator on two TFSMs

Let Ev be the (finite) set of event type names (representing the DSE). Considering a language L, A behavioral interface i_L is a subset of event type names, $i_L \subset Ev$. A B-COoL specification imports N disjoint language interfaces and a set of operators $\mathcal{O}p$, with $N \geq 2$. Each operator from $\mathcal{O}p$ has a set of formal parameters $\mathcal{P}$, where each parameter is defined by a name and its type (*i.e.,* an event type). Each operator also has a correspondence matching condition (denoted CMC) and a correspondence rule (denoted CR). A B-COoL specification is applied to a set of input models denoted $\mathcal{M}_\mathcal{I}$, with $|\mathcal{M}_\mathcal{I}| = N$. From an operational point of view, the first step consists in producing the model behavioral interface of each input model (see example in Figure 3.18). It results in a set of model interfaces denoted $\mathcal{I}_{\mathcal{M}_\mathcal{I}}$, of size N. An interface is a set of events, each of which is typed by an event type. Each operator *op* in $\mathcal{O}p$ is processed individually and several times with different actual parameters, which depend on the model interfaces in $\mathcal{I}_{\mathcal{M}_\mathcal{I}}$. The set of actual parameters to be used is obtained by a *restricted* Cartesian product of all the model interfaces in $\mathcal{I}_{\mathcal{M}_\mathcal{I}}$. The restriction consists in two steps: First, a new set of model interfaces (denoted $\mathcal{I}'_{\mathcal{M}_\mathcal{I}}$) is created. For each parameter p in $\mathcal{P}$, a new model interface $\mathcal{I}^p_{\mathcal{M}_\mathcal{I}}$ is created and all the events in $\mathcal{I}_{\mathcal{M}_\mathcal{I}}$ that have the same type than p are collected in $\mathcal{I}^p_{\mathcal{M}_\mathcal{I}}$. Then, $\mathcal{I}^p_{\mathcal{M}_\mathcal{I}}$ is added to $\mathcal{I}'_{\mathcal{M}_\mathcal{I}}$. Second, a classical Cartesian product is applied on $\mathcal{I}'_{\mathcal{M}_\mathcal{I}}$. It results in a set containing the list of actual parameters to be used with the operator, *i.e.,* each set in the result of the Cartesian product represents the actual parameters of the operator. For each set $actualParams$ in the result of the Cartesian product, if $actualParams$ satisfies the correspondence matching condition (CMC), then the coordination rule (CR) is instantiated with the values in $actualParams$. The instantiation is made in two steps. First, the local events, if any, are created in the targeted coordination language according to the expression used to initialize it. The expression can use any event in $actualParams$ and possibly some constants (*e.g.*, some Integer constants). The local events are added to $actualParams$ so that they can be used in the next. The second step is the application of the relation. It results in the creation of the corresponding relation in the targeted coordination language. The actual parameters of the coordination rule are then the ones from $actualParams$ or some constants, like for the expressions.

Chapter 4

Conclusion

In this chapter, I first summarize the overall scientific vision explored during my research work (Section 4.1). Then I present the main outcomes resulting from my research activities (Section 4.2). These outcomes crystallize all the scientific and technological breakthroughs achieved within the vision presented in this document, either through software platforms (Melange and the GEMOC studio) or through the community (the GEMOC initiative).

Contents

4.1 Overall scientific vision and major achievements

The first phase of research and development in SLE has matured the technology to a level where industry adoption is wide-spread and few fundamental issues remain for efficiently designing any single (disposable) software language. However, the traditional view on SLE suffers from a number of key problems to integrate domain-specific knowledge that cannot be solved without changing our perspective on the notion of language, and especially of DSML. These problems include i) the lack of first-class representations of design decisions in DSML: since design decisions are cross-cutting and intertwined, they are easy to forget and hard to change, leading to high maintenance costs; ii) the lack of support for explicitly relating different DSMLs that makes it very difficult for systems engineers to use multiple DSMLs while enabling a coordinated development of the diverse system aspects, and to reason about information spread across artifacts built with different DSMLs.

We need to take the next step and adopt the perspective that a software language is, fundamentally, software too, that is, the result of a composition of design decisions. These design decisions should be represented as first-class entities in the software language workbench and it should, during the language lifecycle, be possible to add, remove and change language design decisions with limited effort to go from continuous design to continuous meta-design.

Figure 4.1 summarizes the overall vision I implemented to support a *Language-Oriented Modeling*. I first developed contributions to face the development of DSMLs. I focused on foundational concepts and engineering facilities which help language engineer to capture domain-specific knowledge, while being able to customize it for reuse in another context. This results in tools and methods to develop DSMLs from the mashup of modular and reusable domain-specific design choices. I focused on executable DSMLs to support Validation & Verification tools.

Then I presented new foundations and tool-supported approaches to face the multiplication of DSMLs, either to support their reuse and customization in another domain of interest, and to support their coordination with other DSMLs used across the development process. This is achieved by defining relevant DSML interfaces that are used to support composition operators for reuse and coordination.

All my contributions have been implemented in software platforms – the language workbench *Melange* and *the GEMOC studio* – and experienced in real-world case studies to assess their validity. I also founded *the GEMOC initiative*, an attempt to federate the community – both academia and industry – on the grand challenge of the globalization of modeling languages. In the rest of this chapter, I present those main outcomes of my research activities. They embody and integrate my scientific contributions as well as my industrial case studies and experiments.

4.2 Technological outcomes

4.2.1 The Melange Language Workbench

Melange[1] is a language workbench which helps language engineers to mashup their various language concerns as language design choices, to manage their variability, and support their reuse. It provides a modular and reusable approach for customizing, assembling and integrating DSMLs specifications and implementations. The language workbench embeds a model-oriented type system that provides model polymorphism and language substitutability, i.e. the possibility to manipulate a model through different interfaces and to define generic transformations that can be invoked on models written using different DSMLs (see Section 3.1.1). Melange also provides a dedicated meta-language where models are first-class citizens and languages are used to instantiate and manipulate them (see Section 3.1.2). By analogy with the class-based, object-oriented paradigm, Melange can be classified as a language-based, model-oriented programming language.

Melange is tightly integrated with the Eclipse Modeling Framework ecosystem and relies on the meta-language Ecore for the definition of the abstract syntax of DSMLs. Executable meta-modeling is supported by weaving operational semantics defined with Xtend (see Section 2.1). Melange is bundled as a set of Eclipse plug-ins.

4.2.2 The GEMOC Studio

The GEMOC Studio[2] is an eclipse package that contains components for building and composing executable Domain-Specific Modeling Languages (DSMLs). It includes the two workbenches:

- *The GEMOC Language Workbench*: intended to be used by language designers (aka domain experts), it allows them to build and compose new executable DSMLs.
- *The GEMOC Modeling Workbench*: intended to be used by domain designers, it allows them to create and execute models conforming to executable DSMLs.

[1] Cf. http://melange-lang.org
[2] Cf. http://gemoc.org/studio

The GEMOC Studio complements Melange to formally define in a modular way the concurrency model of xDSML, and provides analysis and coordination facilities based on the concurrency model. It also integrates all the contributions presented in this document related to model execution, animation, debugging and trace management.

4.3 The GEMOC Initiative

The GEMOC Initiative[3] is an open and international effort to coordinate and disseminate the research results regarding the support of the coordinated use of various modeling languages that will lead to the concept of globalization of modeling languages, that is, the use of multiple modeling languages to support coordinated development of diverse aspects of a system.

The GEMOC Initiative coordinate effort to develop techniques, frameworks, and environments to facilitate the creation, integration, and automated processing of heterogeneous modeling languages.

GEMOC focuses on three design and validation issues in complex software-intensive systems:

- *Consider various concerns.* Multiple stakeholders are involved in the design process, each with a specific domain expertise. Stakeholders express their perspective with their own language, which then must be composed for global analysis and execution.
- *Integrate heterogeneous parts.* Complex systems integrate different devices specialized for different applications to deliver a global service. Thus, communication, synchronization must be modeled to compose heterogeneous parts and characterize the emerging behavior.
- *Deal with evolution and openness.* It is not possible to establish an exhaustive, finite list of domain languages, communication and timing models. Thus, tools and environments must be open and allow the evolution or the creation of languages and models.

The members involving in the GEMOC initiative gather complementary expertise from software (programming and modeling) languages, models of computation (including time and communication issues), model driven engineering (MDE), and software validation & verification (V&V) and Testing (see members).

The governance of the GEMOC initiative is ensured by the Advisory Board. The role of the Advisory Board is i) to build the community (organization of a GEMOC workshop series at the conference MODELS, Dagstuhl seminar, maintenance of an integrated software studio, etc.); ii) to coordinate the various efforts of the initiative members; and iii) to ensure a proper dissemination of the information related to the initiative (events, scientific and technological results, etc.).

[3] Cf. http://gemoc.org

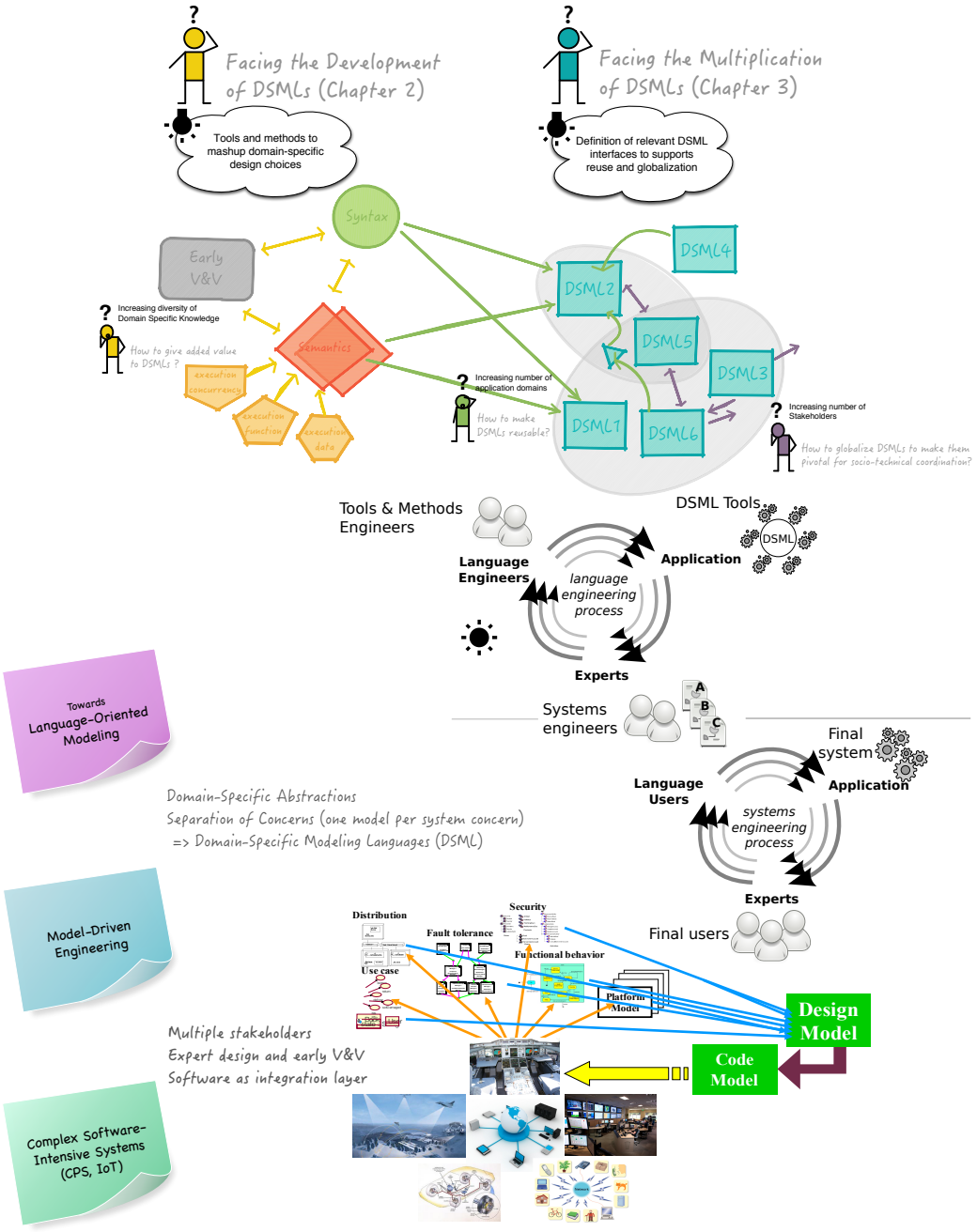


Figure 4.1: The overall vision towards a *Language-Oriented Modeling*

Chapter 5

Perspectives

In this last chapter, I explore a broader vision that pushes forward the use of DSMLs as being pivotal for the socio-technical coordination of the various activities in the development and runtime management of complex software-intensive systems. Despite the wide adoption of DSMLs in industrial development processes, they still suffer from rigidity. The vision we foresee is leading towards more agile, plastic DSMLs from design to runtime. I first explore dynamically adaptable software languages (Section 5.1), with respect to both the language user needs (*Metamorphic DSLs*) and the system environment (*Approximate DSL Runtime*). Then I explore the integration of scientific and *what-if* models in the adaptation loop of dynamically adaptable systems for sustainability (Section 5.2), and the related challenges for DSMLs.

Contents

5.1 Dynamically Adaptable Software Languages

5.1.1 Metamorphic DSLs

The content of this subsection is an adapted excerpt from the following publication:

Mathieu Acher, Benoit Combemale, and Philippe Collet. Metamorphic Domain-Specific Languages: A Journey Into the Shapes of a Language. In *Onward! Essays*, Portland, USA, September 2014. [2]

DSLs are found to be valuable because a well-designed DSL can be much easier to use than a traditional library. The case of SQL is a typical example. Before SQL was conceived, querying and updating relational databases with the available programming languages led to a huge

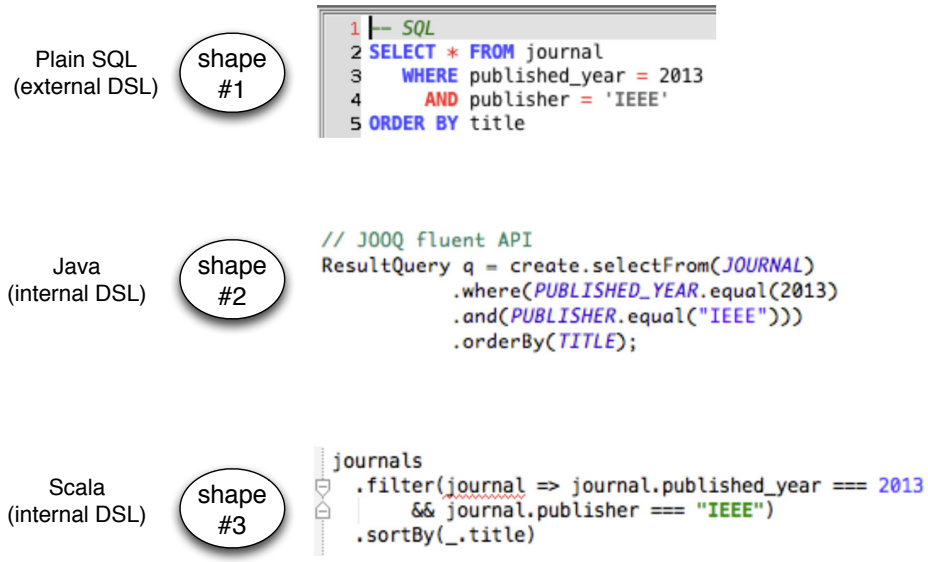


Figure 5.1: Three SQL *shapes*: plain SQL, JOOQ fluent API in Java, Slick API in Scala

semantic gap between data and control processing. With SQL, users can write a query in terms of an implicit algebra without knowing the internal layout of a database. Users can also benefit from performance optimization: a query optimizer can determine the most efficient way to execute a given query.

Another benefit of DSLs is their capacity at improving communication with domain experts [137, 78, 197], thus tackling one of the hardest problems in software development. But DSLs are also ordinary languages, in the sense that many difficult design decisions must be taken during their construction and maintenance [137]. They usually can take different shapes: plain old to more fluent Application Programming Interface (APIs) ; internal or embedded DSLs written inside an existing host language ; external DSLs with their own syntax and domain-specific tooling. To keep it simple, a useful and common distinction is to consider that a DSL can come in two main shapes [78]: external or internal. When an API is primarily designed to be readable and to "*flow*", we also consider it as a DSL.

As for SQL – invented in 1974 and one of the first DSLs – it is interesting to note that it comes itself in different shapes. Figure 5.1 shows three of these shapes on the same basic query example. The top part of the figure shows the plain SQL variant, with a classical *"select, from, where"* clause. In the middle part of the figure, we show the same query written in Java with JOOQ (`http://jooq.org`), a fluent API that emphasises its typesafe nature. The lower part of the figure shows again the same query using the Slick API in Scala (`http://slick.typesafe.com`).

All shapes of a DSL have strengths and weaknesses whoever you are – a developer or user of the DSL. These SQL shapes illustrate this situation. The plain SQL version is an external DSL, making it easier for database experts to write complex queries, but making harder the software engineering job of integrating the DSL with other programming languages. On the entire other side, the JOOQ API is a Java internal DSL. As Java does not provide enough mechanisms to host DSL, the best result is a fluent API which mimics the SQL statement in successive method calls. Some of the SQL concepts are clearly recognizable by SQL experts, but some constructs, such as the AND clause, may lead to scoping errors. The job of the DSL developers is also

reduced to an API implementation. The third example in Slick shows what can be achieved when the host language, here Scala, has some powerful constructs, such as "*filter*", that can be reused. The syntax is then less close to the original SQL, but easier for the average Scala developer. Another solution could have been to use the syntactic flexibility of Scala to closely mimic SQL, but this would not have suppressed some drawbacks, i.e., the internal DSL is a leaky abstraction: some arbitrary code may appear at different places in the domain scope [78, 137], and its concrete syntax can pose a problem for domain experts or non programmers [181].

The basic trade-offs between internal and external DSLs have been already identified and are subject to extensive discussions and research for several years. A new trend though is observed. DSLs are now so widespread that they are used by very different users with separate roles and varied objectives. In the case of SQL, users can be marketers, database experts, system administrators, data warehouse managers, software engineers, or web developers. The objectives can vary a lot: prototyping of basic queries to search some data, sophisticated integration of SQL queries into a web application, etc. *Depending on the kinds of users, roles or objectives, an external shape or an internal shape of a language tends to be a better solution.* Most of the new learners of SQL (e.g., students) have started to learn SQL with explanations and examples written in the external shape of the language as well as a dedicated interactive environment. On the other side, software engineers may have the need to use an internal shape when integrating database concerns.

This diversity poses a major challenge for the DSL engineering discipline:

> How to provide the good shape of a DSL according to the needs of a user?

The idea of having different shapes of a language is indeed appealing. In the case of SQL we can envision the use of different shapes and a transformation from one shape to another, for example, plain SQL could be converted to JOOQ code (and vice-versa). Users could rely on a familiar syntax, in their own environments; the integration of their work with other software ecosystems and languages could be facilitated as well. Yet it must be acknowledged that there is no concrete solution for realizing and supporting the idea.

One reason for the lack of solutions may be the complexity of the problem. By itself, developing one shape of a DSL with a dedicated syntax coming with a set of tools (e.g., type checkers, editing and refactoring facilities) is a difficult and time-consuming task [108]. Fortunately, the increasing maturity of language technologies (e.g., workbenches for creating external DSLs) has democratized the creation of numerous DSLs [78, 198, 188, 66, 107]. Providing the support for transitioning from one shape to another is another difficult problem and open challenge. The number of bridges between N languages ($N \geq 2$) is theoretically exponential. Language workbenches such as Jetbrain's MPS support the transformation of a shape, but with the strong limitation of projecting in a fixed host language and/or environment.

Another barrier is the possible drawbacks of combining multiple programming languages (and multiple paradigms) in application development – sometimes called *polyglot programming* [201, 141, 77]. Yet we want to emphasize the fact that a user usually focuses on an unique shape – the most optimal one according to her task, know-how, education, or simply taste [142, 143]. It is the other stakeholders that are using different shapes in another context and environment.

Despite these socio-technical difficulties, our vision for the future of DSL engineering is as follows:

> The discipline, its foundations, methods, and tools, should go beyond the *constraints* which are imposed by the shapes that a DSL can take for its respective various users. We claim that developers and users of DSLs should not have to choose sides: a DSL should be *metamorphic* and change its shape accordingly!

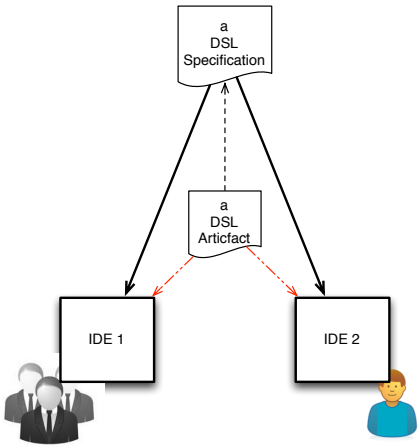


Figure 5.2: Metamorphic DSL (abstract scenario)

Based on our own experience and the analysis of the time-honored evolution of computer languages, we now show how this vision of metamorphic DSLs finds its roots in ongoing work, how it can be defined and what challenges it brings to the community.

5.1.1.1 Ongoing Work

Some recent work already started to wipe off the gap between the different shapes. For example, some approaches for implementing external DSLs try to incorporate the benefits from internal DSL by supporting the reuse of already implemented languages and tools. For instance, Xtext supports the reuse of language libraries such as Xbase, which come with their whole tooling such as editor, type checker and compiler [60]. Conversely, some approaches try to limit the scope of the host infrastructure that can be reused in an internal DSL to ensure a well-defined isolation of the domain-specific constructs. For example, LMS relies on the staging mechanism to define an external DSL on top of the Scala host infrastructure [162]. Besides, projectional language workbenches such as Jetbrain's MPS support the projection in a shape of a purely external DSL or as embedded in a host language similarly to internal DSLs [198]. More generally different strategies for embedding a DSL have been proposed [188, 65, 94, 86].

These recent efforts attempt to integrate the advantages of the different approaches, progressively bridging the gap between them. This is an essential experience to master the various possible bridges and differences between the different shapes. Nevertheless, the same concern (variability or database queries in our example) usually flows through the life cycle, being addressed by different stakeholders with their specific points of view and objectives. Each user expects to manipulate the same programming artefact through the most appropriate shape of the DSL (incl., the whole tooling). For example, a product manager would manipulate SQL queries with plain text SQL and dedicated tools while a software engineer would use an internal DSL (see Figure 5.3 for an illustrative scenario).

5.1.1.2 Vision

Beyond the unification of the different approaches, the vision that we foresee is

> the ability of software languages to be self-adaptable to the most appropriate shape

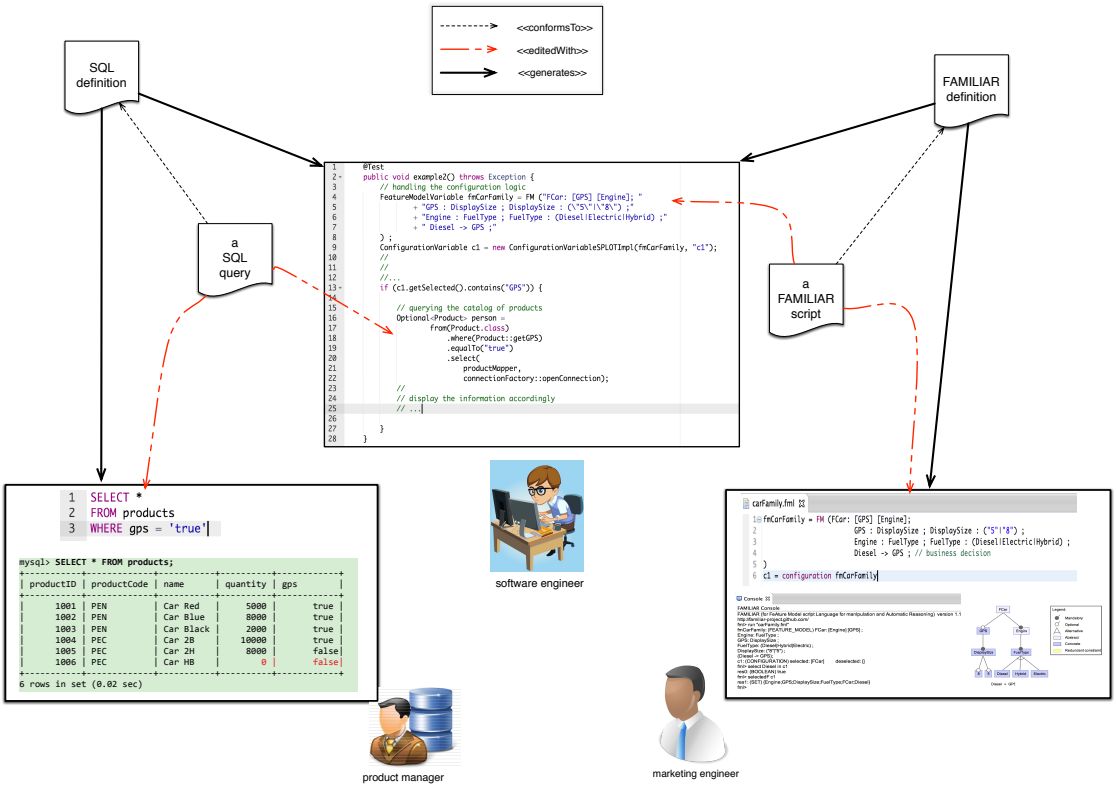


Figure 5.3: Metamorphic DSL (concrete illustration)

(including the corresponding IDE) and according to a particular usage or task. We call *metamorphic DSL* such a domain-specific language, able to change from one shape to another.

From the same language description, we envision the ability to derive various IDEs that can be used accordingly (see Figure 5.2). The challenge consists in supporting the manipulation of the same artefact from the different IDEs dedicated to the different points of view, each one with their specific representation as well as integration into a host infrastructure.

In our vision, the same programming artefact could be started in isolation so that a stakeholder could describe her concern with a highly dedicated environment. The same artefact would flow (e.g., refinement, transformation, composition, consistency checking) and be combined to the other concerns until eventually obtaining the final global system. The vision we propose does not conflict with the use of multiple languages as mentionned in Section 3.2 [37]. It is rather a way to support their *integration* for a coordinated development of diverse domain-specific concerns of a system.

The scenario of Figure 5.3 illustrates the vision on two DSLs, namely SQL and FAMILIAR. It involves different stakeholders (marketing engineer, product manager, software engineer) that aim at providing a configurator of sales product. Each DSL provides two IDEs from the same language definition (incl., one shared by the two DSLs) that can be indifferently used to edit the same conforming artefact. A shared IDE is used as a common infrastructure to integrate the two concerns (e.g., the integration of FAMILIAR and SQL in Java).

5.1.1.3 Challenges

The integration of multiple metamorphic DSLs raises many challenges. One must still find solutions for the integration of domains, especially between business and technical domains. Systematic methods for evaluating *when* a shape of a DSL meets the expected properties (e.g., learnability) and is more suitable to another would benefit to developers of DSL, but are far from being complete. The vision also gives rise to questions about the modularization of artefacts. The technical challenge is to share some information, while being able to visualize and manipulate an artefact in a particular representation and in a particular IDE. A global mechanism must ensure consistency of the artefacts between these heterogeneous IDEs.

The ability to shape up a DSL would open a new path for an effective communication between humans and the realization of global software engineering scenarios: could Metamorphic DSLs bring language engineering to the next level, enabling user-driven task-specific support in domain-specific worlds?

5.1.2 Approximate Software Languages

Language adaptability is also worthwhile at runtime. A model (resp. program) is executed according to the semantics of the language used to design it. The semantics fix the model of computation (MoC) that determines the model's execution flow. Unfortunately, there exist many situations where a fixed MoC over-constrains the possible flow of a model: this prevents the simulation of the model in different execution contexts (*e.g.*, for design-space exploration), the runtime adaptation to the most suitable execution flow according to the environment (*e.g.*, for runtime management with regard to the environment), or even the diversification of (the execution flow of) models or their environments to limit the computation predictability (*e.g.*, for cybersecurity).

In this work, we explore the automatic diversification of Virtual Machines (VMs) by varying the MoC for simulation purposes, to support the adaptation at runtime of the execution flow, and to reduce the predictability of model's computation. In particular, we explore different strategies to change the MoC, relying on the fact that some parts of the computation can be randomly reordered, replaced or even removed. Based on the concurrency model explored in previous work (see Section 2.4) and tailored source code transformation for varying a given model [11], the research program we foresee is as follows:

- Identify plastic computation zones in the source code and the language's operational semantics. Through a combination of static and dynamic analysis, it is possible to identify what we call "plastic computation zones" in the code. We identify different categories of such zones: (i) areas in the code in which the order of computation can vary (e.g. the order in which a block of sequential statements is executed); (ii) areas that can be removed, keeping the essential functionality [176] (e.g., skip some loop iterations); (iii) areas that can be replaced by alternative code [167] (e.g., replace a try-catch by a return statement). If we can identify such plastic zones in the code, we can then tag them as "randomizable" zones for the VM.

- Vary the model of computation. Once we know which zones in the code can be randomized, it is necessary to modify the VM to implement a model of computation that leverages the computation plasticity. This consists in introducing variation points in the interpreter to reflect the diversity of models of computation. Then, the choice of a given variation is performed randomly at runtime, for software V&V, diversification or adaptation.

5.2 Modeling for Sustainability

The content of this section is an adapted excerpt from the following publications:

Benoit Combemale, Betty H.C. Cheng, Ana Moreira, Jean-Michel Bruel, and Jeff Gray. Modeling for Sustainability. Research Report. 2015. [40]

Jean-Michel Bruel, Benoit Combemale, Ileana Ober, and Hélène Raynal. MDE in Practice for Computational Science. In *International Conference on Computational Science (ICCS 2015)*, Reykjavík, Iceland, June 2015. [25]

Gunter Mussbacher, Daniel Amyot, Ruth Breu, Jean-Michel Bruel, Betty Cheng, Philippe Collet, Benoit Combemale, Robert France, Rogardt Heldal, James Hill, Jörg Kienzle, Matthias Schöttle, Friedrich Steimann, Dave Stikkolorum, and Jon Whittle. The Relevance of Model-Driven Engineering Thirty Years from Now. In 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014), volume 8767 of *Model-Driven Engineering Languages and Systems*, page 18, Valencia, Spain, September 2014. Springer. [148]

Computing-based technology has contributed significant capabilities and tools needed to address sustainability challenges [41]. Examples include computational modeling, large-scale data analysis, and sensor technology. In general, sustainable development should meet the needs of the present without compromising the viability of the needs of the future generations [63]. We use the term *sustainability systems* to refer to those computing-based systems that are used to support sustainable development, such as smart grids, smart homes and buildings, and other resource production and management systems. Sustainability systems differ from other types of systems in that their functionality must balance the trade-offs between the three pillars of sustainability: social, economic, and environmental [63]. In addition, as sustainability issues gain increasing attention, so will the demand for access to sustainability systems by a broader population of users. While progress has been made by leveraging modeling technology to manage the complexity of sustainability systems [41, 25, 177], numerous challenges remain as the problem complexity and scope increase, stakeholders and their needs change, and technological advances offer new options to exploit. The vision introduced in this section proposes a new approach to combine disparate sustainability models that will enable broader engagement of society, while supporting the development and the run-time management of sustainability systems.

A long-standing problem has been the inaccessibility of models and the associated data for complex systems to non-experts. This challenge emerges regardless of the model subject. *Scientific models* are used to describe existing systems or phenomena (e.g., biological systems, chemical composition, weather patterns). *Engineering models* are often used to describe systems to be built (e.g., architecture models for buildings, wiring diagrams for electrical systems, UML diagrams for a web server). Sustainability issues have been described primarily by scientific models that enable scientists to understand the impact of changes in one or more of the three pillars of sustainability. Engineering models have been used by (software) developers for the construction of computing-based systems to support various aspects of sustainability systems, such as ecosystem monitoring, power grid management, and climate-control in smart buildings. In contrast, sustainability is a global problem that must be addressed at all levels, from individual decisions to world-wide policies. It is important that different types of stakeholders (e.g., individuals, community leaders, policy makers, industrial organizations) with varying technology proficiencies be able to make well-informed decisions based on exercising the scientific models when using sustainability systems.

This vision paper describes two key insights into how modeling can be used to support sustainability, enable broader engagement of the community, and facilitate more informed

decision-making. First, we observe that many of the foundational concepts used for Model-Driven Engineering (MDE) need to be reconsidered when developing sustainability systems. Instead of considering sustainability as yet another application domain, we need to analyze carefully the global nature of such systems to infer the dual and complementary needs of engineering and scientific models. Second, we propose two modeling feedback loops, one for the engineering model and another for the scientific model, which work together symbiotically to support the development and the run-time management of sustainability systems.

This paper offers a modeling vision for sustainability systems that requires a broader use of modeling techniques. It also provides scientists and engineers an epistemic study of the use of models in science and engineering when considered in the context of sustainability. The dual feedback loops leverage the modeling techniques from both disciplines to achieve a more holistic MDE-based approach to supporting sustainability systems. The engineering model feedback loop comprises the models used to develop and manage the software infrastructure for a cyber-physical system for sustainability (e.g., smart grid management). The scientific modeling loop comprises a multi-view scientific modeling infrastructure for capturing the three pillars of sustainability (i.e., social, economic, and environmental) that uses an *Aggregator* to integrate multiple scientific models and incorporate information from a sustainability system and its context to enable a stakeholder to select specific "views" of sustainability to explore (e.g., pose "what if" scenarios across multiple dimensions with "zoom-in" and "zoom-out" capabilities for fine-grained or global-level views, respectively). The results of the science-based inquiries can either be used to predict the social, economic, or environmental impact of a behavior change in resource usage, or sent back through the engineering modeling feedback loop to adapt the sustainability system.

5.2.1 Problem Statement

Sustainability has increasingly become an urgent issue over the past several decades [63]. At the same time, the pervasiveness of the Internet in our daily lives and the increasing use of global software-based systems that handle large, complex, networked, and heterogeneous systems that involve a wide range of users and hardware devices, opens new frontiers for innovative designs and solutions. The technological advances offer new user experiences and access to information that collectively have the potential to influence behavior change, including making an impact on sustainability. The next generation of development approaches should support multiple dimensions of sustainability, ranging from long-lasting dependable and dynamically adaptive software, to green software requiring less computing and fewer energy resources, to software that encourages sustainable human behavior (e.g., smart plugs and appliances, Tesla Eco-Batteries to support the Roof PV generation at Home [51], and market design and regulations that transition consumers towards more energy-saving practices; the so-called "power of the negawatt" [128]). Approaches supporting various dimensions of sustainability are likely to have a substantial positive impact on people, the economy, and the environment for the short and long term.

Traditional software abstractions and development techniques are inadequate to tackle this task in several important ways. These techniques are (computing) system focused, where abstractions are intended to manage the system complexity, provide different views of the system, and non-functional qualities refer to properties about the system, such as performance, memory usage, and reliability. In contrast, sustainability systems must also consider how that functionality has to be delivered in changing contexts as defined by the environment, economic circumstances, and social policies. For example, when control systems are used to support sustainability, it is insufficient to only consider the management of an individual element, such as a single transformer in a smart grid system. Instead, the context in which the transformer is used must be considered holistically, including the power sources (e.g., windmills, water power,

solar), changing environmental conditions, the volume and types of power consumers, usage patterns, and economic factors.

Modeling has been the essential mechanism to cope with the complexity of reality. In science, models are used to describe existing phenomena of the real world [178, 157]. In engineering, models are typically used to describe a system to be developed in the future. Thus, engineering models are typically *constructive* while scientific models are *descriptive*. An engineering model may incorporate scientific models to build the systems.

New modeling approaches are needed to meet the challenges posed by the change in the nature of sustainable, global software systems that are different from the traditional MDE approaches. Existing mechanisms need to be leveraged to facilitate modeling and analysis of the multiple dimensions of sustainability system requirements and architectures. In particular, we need to explore the challenges of modeling for sustainability; explore modeling one or more dimensions of sustainability; develop innovative decision support techniques, including what kind of data processing and visualizations are needed to enable different types of stakeholders with complementary interests to understand sustainability concerns and impacts of individual decisions [148]; propose approaches that cope with the increasing demand of precision, trustability, reliability, scalability, adaptation and context-awareness, and timely acquisition and/or computation of information for decision-making; investigate cognitive models, methods and tools, that can be used by a broad range of stakeholders and users, not just engineers. Therefore, a sustainability (eco)system has to be modeled in the context in which it operates, with respect to its environment, social policies, and economic conditions.

In the following, we examine how MDE is currently applied in engineering disciplines in the context of sustainability, discuss its usefulness in science and show how the differences in model use between the two fields is decreasing given current trends. Our proposed vision is to facilitate their symbiotic use to support sustainability systems.

5.2.2 Vision and Challenges

Computing and technology continue to play an increasing role to support sustainability research. More sophisticated ecosystem monitoring techniques have yielded petabytes of data for scientists to analyze. As scientists build increasingly more complex scientific models, additional challenges are posed to the computing disciplines to make the data and model-based analysis results more accessible and understandable to the scientists and other stakeholders, and automatically fed back to the engineering models to adapt the corresponding sustainability systems.

Building from the context of the previous figures, Figure 5.4 depicts an integrated approach where each model provides feedback to the other. In this approach, engineering models are dynamically adapted by interpreting the resulting impact on the sustainability system, while scientific models are also impacted by the use of the engineering models in a dynamic feedback loop that continuously adapts the system to reflect the change in priorities among the three pillars. Other stakeholders (e.g., individuals, community leaders, policy makers, industrial organizations) can select specific (personalized) views of sustainability to explore the impact of changes in social behavior, policies, and resource consumption.

We introduce the concept of *Aggregator* at the core of the approach, whose objective is threefold: i) to ensure the fusion of the various heterogeneous data sources (scientific models, personal expectations and information collected from the sensors); ii) to adapt the scientific models accordingly and to provide specific views accessible to a broader public; and iii) to (automatically) update the engineering models according to science-informed decisions taken from the scientific models. The three central roles of the Aggregator are further described in the remainder of this section.

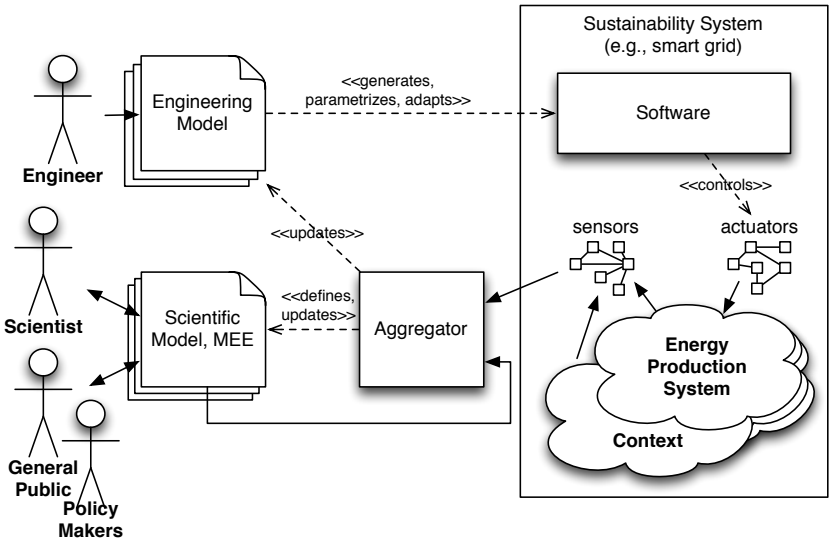


Figure 5.4: Intertwined use of engineering and scientific models

5.2.2.1 Model and Data Fusion

When considering the modeling needs of an autonomic or self-adaptive sustainability system, it must be able to monitor its environment, adapt to changing conditions, and be resilient to component failures and attacks. Many of these systems make use of wireless sensor networks and cyber-physical systems. As these systems grow in scale, complexity, and heterogeneity, they become ultra-large scale systems (ULS) [149] or Internet of Things (IoT) [7]. ULS systems comprise tens of thousands of sensor nodes, thousands of decision nodes, involve heterogeneous components (e.g., wind turbines, solar panels, small motors) that must work together collectively to produce specific results, such as energy production [149]. In the original definition of ULS systems, the concept of a *ULS Ecosystem* was introduced to capture the interplay between cyber-physical, environmental, and social.

Consider the emerging intelligent transportation system (ITS) comprising numerous intelligent vehicle systems, each containing numerous sensors and onboard controllers to provide autonomic features, such as lane keeping, smart cruise control, and collision avoidance. These cars need to communicate with each other, with satellites and roadway beacons. The objective of the entire ITS is to minimize collisions, while increasing throughput of vehicles traveling from point A to point B. MDE is used to capture the intended vehicle behavior in the context of social concerns (e.g., traffic laws) and environmental conditions (e.g., road conditions, weather). The main challenge with modeling social and environmental concerns is uncertainty. When considering sustainability systems for ULS, we need to add the economic factors to the ULS Ecosystem.

Unlike an ITS, a sustainability system requires consideration of the many trade offs between the numerous scientific models when looking for potential solutions, where solutions may involve changes to one or more of the three pillars of sustainability, as well as the sustainability system itself. For example, when considering power consumption and power production from multiple sources (e.g., wind, solar, nuclear, water), policy analysts may use the Aggregator to explore several scenarios involving the respective scientific models and the sensors from the sustainability systems to determine what to do during a drought season with record high temperatures. The Aggregator analysis may suggest how to adjust cost models and/or legislate temporary laws during the drought season (e.g., impose fines for washing cars or watering lawns). In addition, the

feedback from the Aggregator to the smart grid system may be to harness more power from solar sources, decrease demand from water-based sources, and reroute power to high-demand areas and ensure that hospitals and other care facilities have sufficient resources to run air-conditioning, among other needs.

5.2.2.2 Personalizing Sustainability

A key insight to our vision is that sustainability approaches must be customer-centric, focusing on individuals. According to OPOWER[1], "What's at stake are tomorrow's energy consumers. To thrive for another century, utilities must capture their attention and exceed their expectations. It starts with the customer experience." To effectively address sustainability, the customer has to understand and share the vision of the providers and vice versa. Previously, we proposed the concept of *Model Experiencing Environments (MEEs)* [148] as an approach to support complex model and data integration, while offering customizable interfaces for accessing model analysis results and their visualizations. In this paper, the Aggregator is intended to support the different types of model and data integration needed by a MEE, where a MEE interface and visualization support is customizable according to a given user and their interests. The objective of MEEs is to enable different types of stakeholders to "experience" models according to their level of interests and needs. A MEE can be used as an educational medium for enabling children and others to learn about the impact of their individual actions and decisions (e.g., what is the impact of taking 4-minute showers instead of 15-minute showers over a year long period on water consumption for the household? for the town?). MEEs can also be used to study the effects of collective behavior by a community (e.g., what is the impact on environmental resources by decreasing red meat consumption down to once per week?). For land use or environmental policy analysts, MEEs can be used to pose more sophisticated "what if?" scenarios to understand the impact of specific policies or legislation. MEEs empower stakeholders to better understand the cause and effects of individual and global actions and decisions. MEEs are intended to make the complexity of the scientific models, the data and model fusion process, the distributed nature of the data, and the resolution of temporal and spatial differences between the models transparent to the MEE user, thus enabling them to focus on the science questions.

5.2.2.3 Feedback to the Engineering Models

Each of the social, economic and environmental pillars must have their respective information captured in a (domain-specific) model. New integration techniques are needed to support a systematic, well-defined approach to integrate these disparate models to enable well-informed decisions and functionality to be provided by sustainability systems. When the Aggregator is integrating the scientific models, it must consider the temporal and spatial dependencies, as well as the granularity of the data used for the different models and sensor data from the sustainability system. Updates to the engineering models for the sustainability system must consider the type of user engaged with the system. In other words, an individual power consumer will not be allowed to effect change to the power distribution system. But the manager of a power generation company may use the MEE interface to explore different "What if" scenarios to determine the most reliable and cost-effective strategy for delivering power to its customers. The results of the analysis may propose updates to be autonomously made to the sustainability system via the feedback link from the Aggregator.

[1] See http://www2.opower.com/moments-that-matter-whitepaper.

5.2.2.4 Additional Socio-Technical Concerns

In addition to the modeling challenges previously mentioned, several additional areas need to be considered to facilitate the socio-technical modeling shift. Approaches should include resource usage analytics techniques to enable researchers to examine complex relationships between models and variables, using the power of predictive analytics to understand behavior patterns and the impact of one pillar with respect to the other pillars. New approaches should integrate ideas from behavioral science to produce persuasive solutions to engage individuals; therefore, going beyond the traditional one-size-fits-all solutions. In such a setting, involving thousands or millions of people, data security and anonymity to preserve customer privacy needs to be planned from the very early stages of the development. Finally, the proposed models, languages and techniques need to handle the voluminous amount of disparate data coming from a wide variety of sources. In the era of Big Data, reducing large-scale problems to a scale that humans can comprehend and act upon is fundamental. Thomas and Cook [186] discuss various scalability challenges that range from information scalability, to visual scalability, display scalability, human scalability, and software scalability.

Bibliography

[1] Language and IDE modularization and composition with MPS. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *Proc. of GTTSE'13*, pages 383–430, 2013.

[2] Mathieu Acher, Benoit Combemale, and Philippe Collet. Metamorphic Domain-Specific Languages: A Journey Into the Shapes of a Language. In *Onward! Essays*, Portland, USA, September 2014.

[3] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. On challenges of model transformation from uml to alloy. *Software and Systems Modeling*, 9(1):69–86, 2010.

[4] C. André. Syntax and Semantics of the Clock Constraint Specification Language (CCSL). Technical report, 2009.

[5] K. Androutsopoulos, D. Binkley, D. Clark, N. Gold, M. Harman, K. Lano, and Z. Li. Model projection: Simplifying models in response to restricting the environment. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 291–300, New York, NY, USA, 2011. ACM.

[6] André Arnold. Transition systems and concurrent processes. *Mathematical problems in Computation theory*, 1987.

[7] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[8] Jean-Philippe Babau and Mickael Kerboeuf. Domain Specific Language Modeling Facilities. In *MoDELS Workshop on Models and Evolution*, 2011.

[9] Gianfranco Balbo. Introduction to generalized stochastic petri nets. In Marco Bernardo and Jane Hillston, editors, *Formal Methods for Performance Evaluation*, volume 4486 of *Lecture Notes in Computer Science*, pages 83–131. Springer Berlin Heidelberg, 2007.

[10] L. Barroca, J.L. Fiadeiro, M. Jackson, R. Laney, and B. Nuseibeh. Problem frames: A case for coordination. In *Coordination*. 2004.

[11] Benoit Baudry, Simon Allier, and Martin Monperrus. Tailored Source Code Transformations to Synthesize Computationally Diverse Program Variants. In *ISSTA*, pages 149–159, United States, 2014.

[12] Reda Bendraou, Jean-Marc Jézéquel, and Franck Fleurey. Combining aspect and model-driven engineering approaches for software process modeling and execution. In *Trustworthy Software Development Processes*, Lecture Notes in Computer Science, pages 148–160. Springer, 2009.

[13] Amine Benelallam, Massimo Tisi, and David Launay. Neo4EMF, a Scalable Persistence Layer for EMF Models. 2014.

[14] Albert Benveniste, Benoît Caillaud, Luca Carloni, and Alberto Sangiovanni-Vincentelli. Tag machines. In *ACM Emsoft*, 2005.

[15] Bernard Berthomieu, Jean-Paul Bodeveix, Mamoun Filali, Patrick Farail, Pierre Gaufillet, Hubert Garavel, and Frederic Lang. FIACRE: an Intermediate Language for Model Verification in the TOPCASED Environment. In *ERTS'08*, January 2008.

[16] Arnaud Blouin, Benoît Combemale, Benoit Baudry, and Olivier Beaudoux. Kompren: modeling and generating model slicers. *Software and System Modeling*, 14(1):321–337, 2015.

[17] Patrick Borras, Dominique Clément, Thierry Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and Valérie Pascual. Centaur: the system. *ACM SIGPLAN Notices*, 24(2):14–24, 1989.

[18] Frédéric Boulanger and Cécile Hardebolle. Simulation of Multi-Formalism Models with ModHel'X. In *First International Conference on Software Testing, Verification, and Validation (ICST)*, pages 318–327. IEEE, 2008.

[19] Erwan Bousse. Combining Verification and Validation techniques. In *Doctoral Symposium of ECMFA, ECOOP and ECSA 2013*, pages 1–10, 2013.

[20] Erwan Bousse, Benoit Combemale, and Benoit Baudry. Scalable Armies of Model Clones through Data Sharing. In *17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014)*, Valencia, Spain, 2014. Springer.

[21] Erwan Bousse, Benoit Combemale, and Benoit Baudry. Towards Scalable Multidimensional Execution Traces for xDSMLs. In *11th Workshop on Model Design, Verification and Validation Integrating Verification and Validation in MDE (MoDeVVa 2014)*, Valencia, Spain, September 2014.

[22] Erwan Bousse, Jonathan Corley, Benoit Combemale, Jeff Gray, and Benoit Baudry. Supporting Efficient and Advanced Omniscient Debugging for xDSMLs. In *8th International Conference on Software Language Engineering (SLE 2015)* , Pittsburg, USA, October 2015. ACM.

[23] Erwan Bousse, Tanja Mayerhofer, Benoit Combemale, and Benoit Baudry. A Generative Approach to Define Rich Domain-Specific Trace Metamodels. In *11th European Conference on Modelling Foundations and Applications (ECMFA 2015)*, Lecture Notes in Computer Science, L'Aquila, Italy, July 2015. Springer.

[24] Kim B. Bruce, Angela Schuett, Robert van Gent, and Adrian Fiech. Polytoil: A type-safe polymorphic object-oriented language. *ACM TOPLAS*, 25(2), 2003.

[25] Jean-Michel Bruel, Benoit Combemale, Ileana Ober, and Hélène Raynal. MDE in Practice for Computational Science. In *International Conference on Computational Science (ICCS 2015)*, Reykjavík, Iceland, June 2015.

[26] Barbara Chapman, Matthew Haines, Piyush Mehrota, Hans Zima, and John Van Rosendale. Opus: A coordination language for multidisciplinary applications. *Sci. Program.*, 1997.

[27] Franck Chauvel and Jean-Marc Jézéquel. Code generation from UML models with semantic variation points. In *Model Driven Engineering Languages and Systems*, pages 54–68. 2005.

[28] Hyun Cho and Jeff Gray. Design patterns for metamodels. In *SPLASH '11 Workshops*, pages 25–32. ACM, 2011.

[29] Tony Clark, Andy Evans, and Stuart Kent. The Metamodelling Language Calculus: Foundation Semantics for UML. In *FASE'01*, volume 2029 of *Lecture Notes in Computer Science*, pages 17–31. Springer, 2001.

[30] Tony Clark, Paul Sammut, and James Willans. Applied Metamodelling – A Foundation for Language Driven Development. Second Edition, 2008.

[31] Tony Clark, Paul Sammut, and James Willans. SUPERLANGUAGES – Developing Languages and Applications with XMF. 2008.

[32] Curtis Clifton and Gary T. Leavens. Multijava: Modular open classes and symmetric multiple dispatch for java. In *Int. Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, pages 130–145, 2000.

[33] Benoit Combemale, Xavier Crégut, Pierre-Loic Garoche, and Xavier Thirioux. Essay on Semantics Definition in MDE. An Instrumented Approach for Model Verification. *Journal of Software*, 4(9):943–958, November 2009.

[34] Benoit Combemale, Xavier Crégut, Pierre-Loic Garoche, Xavier Thirioux, and Francois Vernadat. A Property-Driven Approach to Formal Verification of Process Models. In Jorge Cardoso, José Cordeiro, Joaquim Filipe, and Vitor Pedrosa, editors, *Enterprise Information System IX*, volume 12, pages 286–300. LNBIP, Springer, 2008.

[35] Benoit Combemale, Xavier Crégut, and Marc Pantel. A Design Pattern to Build Executable DSMLs and associated V&V tools. In *19th Asia-Pacific Software Engineering Conference (APSEC 2012)*, Hong Kong, December 2012. IEEE.

[36] Benoit Combemale, Julien Deantoni, Olivier Barais, Arnaud Blouin, Erwan Bousse, Cédric Brun, Thomas Degueule, and Didier Vojtisek. A Solution to the TTC'15 Model Execution Case Using the GEMOC Studio. In *8th Transformation Tool Contest (TTC 2015)*, 2015.

[37] Benoit Combemale, Julien Deantoni, Benoit Baudry, Robert France, Jean-Marc Jézéquel, and Jeff Gray. Globalizing Modeling Languages. *IEEE Computer*, pages 68–71, June 2014.

[38] Benoit Combemale, Julien Deantoni, Matias Vara Larsen, Frédéric Mallet, Olivier Barais, Benoit Baudry, and Robert France. Reifying Concurrency for Executable Metamodeling. In Richard F. Paige Martin Erwig and Eric van Wyk, editors, *6th International Conference on Software Language Engineering (SLE 2013)*, Lecture Notes in Computer Science, Indianapolis, USA, 2013. Springer-Verlag.

[39] Benoit Combemale, Cécile Hardebolle, Christophe Jacquet, Frédéric Boulanger, and Benoit Baudry. Bridging the Chasm between Executable Metamodeling and Models of Computation. In *5th International Conference on Software Language Engineering (SLE 2012)*, LNCS, Dresden, Germany, 2012. Springer.

[40] Benoit Combemale, Betty H.C. Cheng, Ana Moreira, Jean-Michel Bruel, and Jeff Gray. Modeling for Sustainability. Technical report, 2015.

[41] Committee on Computing Research for Environmental and Societal Sustainability. *Computing Research for Sustainability*. National Academies Press, 2012.

[42] Steve Cook, Gareth Jones, Stuart Kent, and Alan C. Wills. *Domain-specific development with visual studio dsl tools*. Addison-Wesley Professional, 2007.

[43] Jonathan Corley, Brian P. Eddy, and Jeff Gray. Towards Efficient and Scalabale Omniscient Debugging for Model Transformations. In *14th Workshop on Domain-Specific Modeling*, pages 13–18. ACM, 2014.

[44] Bas Cornelissen, Danny Holten, Andy Zaidman, Leon Moonen, Jarke J. Van Wijk, and Arie Van Deursen. Understanding execution traces using massive sequence and circular bundle views. *IEEE International Conference on Program Comprehension*, pages 49–58, 2007.

[45] M. Crane and J. Dingel. UML vs. classical vs. Rhapsody statecharts: not all models are created equal. *International Journal on Software and Systems Modeling (SoSyM)*, 6(4):415–435, 2007.

[46] Arnaud Cuccuru, Chokri Mraidha, François Terrier, and Sébastien Gérard. Templatable metamodels for semantic variation points. In *ECMDA-FA*, 2007.

[47] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. In *ICMT*, 2009.

[48] Juan de Lara and Esther Guerra. Generic meta-modelling with concepts, templates and mixin layers. In *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS)*. Springer, 2010.

[49] Juan de Lara and Esther Guerra. From types to type requirements: genericity for model-driven engineering. *International Journal on Software and Systems Modeling (SoSyM)*, 2011.

[50] Juan de Lara, Esther Guerra, and Jesús Sánchez-Cuadrado. Abstracting modelling languages: A reutilization approach. In *Advanced Information Systems Engineering*, pages 127–143. 2012.

[51] James Dean. Power to the people in energy revolution: Eco-batteries will slash household bills. *The Times*, May 2015.

[52] Julien Deantoni, Papa Issa Diallo, Ciprian Teodorov, Joël Champeau, and Benoit Combemale. Operational Semantics of the Model of Concurrency and Communication Language. Technical report.

[53] Julien Deantoni, Papa Issa Diallo, Ciprian Teodorov, Joël Champeau, and Benoit Combemale. Towards a Meta-Language for the Concurrency Concern in DSLs. In *Design, Automation and Test in Europe Conference and Exhibition (DATE 2015), PI*, Grenoble, France, March 2015.

[54] Julien Deantoni and Frédéric Mallet. ECL: the Event Constraint Language, an Extension of OCL with Events. Technical Report RR-8031, INRIA, 2012.

[55] Julien Deantoni and Frédéric Mallet. TimeSquare: Treat your Models with Logical Time. In *TOOLS*, volume 7304 of *Lecture Notes in Computer Science*, pages 34–41. Springer, May 2012.

[56] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. Melange: A Meta-language for Modular and Reusable Development of DSLs. In *8th International Conference on Software Language Engineering (SLE 2015)* , Pittsburg, USA, October 2015. ACM.

[57] Papa Issa Diallo, Joël Champeau, and Vincent Leilde. Model based engineering for the support of models of computation: The cometa approach. In *MPM*, 2011.

[58] Jürgen Dingel, Zinovy Diskin, and Alanna Zito. Understanding and improving UML package merge. *SoSym*, 7(4):443–467, 2008.

[59] Matthew B. Dwyer and Sebastian Elbaum. Unifying verification and validation techniques: relating behavior and properties through partial evidence. *Proceedings of the FSE/SDP workshop on Future of software engineering research (FoSER '10)*, pages 93–97, 2010.

[60] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. Xbase: Implementing Domain-specific Languages for Java. In *11th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 112–121. ACM, 2012.

[61] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity – the Ptolemy approach. *Proc. of the IEEE*, 91(1), 2003.

[62] Matthew Emerson and Janos Sztipanovits. Techniques for metamodel composition. In *OOPSLA Workshop on Domain Specific Modeling*, pages 123–139, 2006.

[63] UN World Commission on Environment and Development. Our common future, 1987. See http://en.wikipedia.org/wiki/Sustainability.

[64] S. Erdweg, P. G. Giarrusso, and T. Rendel. Language composition untangled. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*, LDTA '12, pages 7:1–7:8. ACM, 2012.

[65] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Sugarj: Library-based syntactic language extensibility. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 391–406. ACM, 2011.

[66] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. The State of the Art in Language Workbenches - Conclusions from the Language Workbench Challenge. In *SLE*, volume 8225 of *Lecture Notes in Computer Science*, pages 197–217. Springer, 2013.

[67] Erik Ernst. Family polymorphism. In JørgenLindskov Knudsen, editor, *Object-Oriented Programming (ECOOP 2001)*, volume 2072 of *Lecture Notes in Computer Science*, pages 303–326. Springer Berlin Heidelberg, 2001.

[68] R. Eshuis and R. Wieringa. Tool Support for Verifying UML Activity Diagrams. *IEEE Trans. Softw. Eng.*, 30(7):437–447, 2004.

[69] Esper. Espertech, 2009.

[70] Jacky Estublier, German Vega, and AncaDaniela Ionita. Composing domain-specific languages for wide-scope software engineering applications. In *Proc. of MODELS'05*, pages 69–83, 2005.

[71] Moritz Eysholdt and Heiko Behrens. Xtext: Implement your language faster than the quick and dirty way. In *Proc. of OOPSLA '10 Companion*, pages 307–309, 2010.

[72] Patrick Farail, Pierre Gaufillet, Agusti Canals, Christophe Le Camus, David Sciamma, Pierre Michel, Xavier Crégut, and Marc Pantel. The TOPCASED project: a Toolkit in OPen source for Critical Aeronautic SystEms Design. In *Embedded Real Time Software (ERTS'06)*, Toulouse, 25-27 January 2006.

[73] J-M. Favre, D. Gasevic, R. Lämmel, and E. Pek. Empirical language analysis in software linguistics. In *Software Language Engineering*, volume 6563 of *Lecture Notes in Computer Science*, pages 316–326. Springer, 2011.

[74] Franck Fleurey, Benoit Baudry, Robert France, and Sudipto Ghosh. A generic approach for automatic model composition. In *AOM Workshop at Models*, 2007.

[75] Franck Fleurey, Brice Morin, Arnor Solberg, and Olivier Barais. MDE to manage communications with and between resource-constrained systems. In *Proc. of MODELS'11*, pages 349–363, 2011.

[76] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM TOPLAS*, 29(3), 2007.

[77] Martin Fowler. One language http://martinfowler.com/bliki/OneLanguage.html, 2007.

[78] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 2010.

[79] Robert France and Bernhard Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *FOSE'07*. IEEE, 2007.

[80] Lars-ake Fredlund, Bengt Jonsson, and Joachim Parrow. An implementation of a translational semantics for an imperative language. In *CONCUR*, volume 458 of *Lecture Notes in Computer Science*, pages 246–262. Springer, 1990.

[81] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A practical guide to SysML: the systems modeling language*. 2014.

[82] K.B. Gallagher and J.R. Lyle. Using program slicing in software maintenance. *Software Engineering, IEEE Transactions on*, 17(8):751–761, 1991.

[83] Erich Gamma, Richard Helm, and Ralph Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[84] Marc Geilen, Twan Basten, and Sander Stuijk. Minimising buffer requirements of synchronous dataflow graphs with model checking. In *DAC*, pages 819–824. ACM, 2005.

[85] Debasish Ghosh. *DSLs in Action*. Manning Publications Co., Greenwich, CT, USA, 2010.

[86] Andy Gill. Domain-specific Languages and Code Synthesis Using Haskell. *ACM Queue*, 12(4):30:30–30:43, April 2014.

[87] David Gries. *The science of programming*, volume 198. Springer, 1981.

[88] Clément Guy, Benoit Combemale, Steven Derrien, and Jean-Marc Jézéquel. On Model Subtyping. In Antonio Valecillo, editor, *Proceedings of the 8th European Conference on Modelling Foundations and Applications (ECMFA 2012)*, number 7349 in Lecture Notes in Computer Science, pages 400–415. Springer, 2012.

[89] David Harel and Hillel Kugler. The rhapsody semantics of statecharts (or, on the executable core of the uml). In Hartmut Ehrig, Werner Damm, Jörg Desel, Martin Große-Rhode, Wolfgang Reif, Eckehard Schnieder, and Engelbert Westkämper, editors, *Integration of Software Specification Techniques for Applications in Engineering*, volume 3147 of *Lecture Notes in Computer Science*, pages 325–354. Springer Berlin Heidelberg, 2004.

[90] David Harel and Amnon Naamad. The STATEMATE semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(Issue 4):293–333, 1996.

[91] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *Computer*, 37(10):64–72, 2004.

[92] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[93] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[94] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. Polymorphic embedding of dsls. In *7th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 137–148. ACM, 2008.

[95] Zhenjiang Hu, Andy Schürr, Perdita Stevens, and James F. Terwilliger. Bidirectional transformation "bx" (dagstuhl seminar 11031). *Dagstuhl Reports*, 1(1), 2011.

[96] Paul Hudak. Modular domain specific languages and tools. In *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society, June 1998.

[97] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical assessment of mde in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 471–480. ACM, 2011.

[98] D. Jackson. Alloy: A Lightweight Object Modelling Notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[99] Axel Jantsch. *Modeling Embedded Systems and SoCs*. Morgan Kaufmann Publishers Inc., 2004.

[100] Cédric Jeanneret, Martin Glinz, and Benoit Baudry. Estimating footprints of model operations. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 601–610. IEEE, 2011.

[101] Jean-Marc Jézéquel, Olivier Barais, and Franck Fleurey. Model driven language engineering with kermeta. In João M. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'09)*, volume 6491 of *Lecture Notes in Computer Science*, pages 201–221. Springer, 2011.

[102] Jean-Marc Jézéquel, Benoît Combemale, Olivier Barais, Martin Monperrus, and François Fouquet. Mashup of metalanguages and its implementation in the kermeta language workbench. *Software and System Modeling*, 14(2):905–920, 2015.

[103] H. Kagdi, J.I. Maletic, and A. Sutton. Context-Free Slicing of UML Class Models. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pages 635–638. IEEE Computer Society, 2005.

[104] Gabor Karsai, Aditya Agrawal, Feng Shi, and Jonathan Sprinkle. On the use of graph transformations for the formal specification of model interpreters. *Journal of Universal Computer Science*, 9, 2003.

[105] U. Kastens and W.M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31(7):601–627, 1994.

[106] Lennart C.L. Kats and Eelco Visser. The spoofax language workbench: rules for declarative specification of languages and ides. *ACM SIGPLAN Notices*, 45:444–463, October 2010.

[107] Steven Kelly, Kalle Lyytinen, Matti Rossi, and Juha-Pekka Tolvanen. MetaEdit+ at the Age of 20. In *Seminal Contributions to Information Systems Engineering*, pages 131–137. Springer, 2013.

[108] Steven Kelly and Risto Pohjonen. Worst Practices for Domain-Specific Modeling. *IEEE Software*, 26(4):22–29, 2009.

[109] Mickael Kerboeuf and Jean-Philippe Babau. A DSML for reversible transformations. In *OOPSLA Workshop on Domain-Specific Modeling*, 2011.

[110] A. Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 1 edition, 2008.

[111] Donald E Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, 1968.

[112] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. Merging models with the epsilon merging language (EML). In *ACM/IEEE Models/UML*, 2006.

[113] DS Kolovos, LM Rose, and Nicholas Matragkas. A research roadmap towards achieving scalability in model driven engineering. In *BigMDE'13*, 2013.

[114] B. Korel, I. Singh, L. Tahat, and B. Vaysburg. Slicing of state-based models. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 34–43, 2003.

[115] Holger Krahn, Bernhard Rumpe, and Steven Volkel. Monticore: Modular development of textual domain specific languages. In *Objects, Components, Models and Patterns*, volume 11 of *LNBIP*, pages 297–315. Springer Berlin Heidelberg, 2008.

[116] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: a framework for compositional development of domain specific languages. *JSTT*, 12(5):353–372, 2010.

[117] A Kusel, J Schönböck, M Wimmer, G Kappel, W Retschitzegger, and W Schwinger. Reuse in model-to-model transformation languages: are we there yet? *SoSyM*, pages 1–36, 2013.

[118] Sabine Kuske, Martin Gogolla, Ralf Kollmann, and Hans-Jörg Kreowski. An Integrated Semantics for UML Class, Object and State Diagrams Based on Graph Transformation. In *IFM'02*, volume 2335 of *Lecture Notes in Computer Science*, pages 11–28. Springer, 2002.

[119] Thomas Kühne. On model compatibility with referees and contexts. *International Journal on Software and Systems Modeling (SoSyM)*, 2012.

[120] K. Lano and S. Kolahdouz-Rahimi. Slicing of UML Models Using Model Transformations. In *Model Driven Engineering Languages and Systems*, volume 6395 of *Lecture Notes in Computer Science*, pages 228–242. Springer Berlin Heidelberg, 2010.

[121] K. Lano and S. Kolahdouz-Rahimi. Slicing Techniques for UML Models. *Journal of Object Technology*, 10:11:1–49, 2011.

[122] Florent Latombe, Xavier Crégut, Benoît Combemale, Julien Deantoni, and Marc Pantel. Weaving Concurrency in eXecutable Domain-Specific Modeling Languages. In *8th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2015)*, Pittsburg, USA, October 2015. ACM.

[123] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing domain-specific design environments. *Computer*, 34(11):44–51, Nov 2001.

[124] Edward A. Lee. Overview of the Ptolemy project. Technical Memorandum UCB/ERL no M03/25, University of California at Berkeley, 2003.

[125] Edward A. Lee and Alberto L. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 17(12):1217–1229, 1998.

[126] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.

[127] B.H. Liskov and J.M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.

[128] Amory B Lovins. The negawatt revolution. *Across the Board*, 1990.

[129] Frédéric Mallet, Julien DeAntoni, Charles André, and Robert de Simone. The clock constraint specification language for building timed causality models - application to synchronous data flow graphs. *ISSE*, 6(1-2):99–106, 2010.

[130] Slavisa Markovic and Thomas Baar. Semantics of OCL specified with QVT. *International Journal on Software and Systems Modeling (SoSyM)*, 7(4):399–422, 2008.

[131] Tanja Mayerhofer and Manuel Wimmer. The TTC 2015 Model Execution Case. In *8th Transformation Tool Contest (TTC)*. CEUR, 2015.

[132] John McCarthy. Towards a mathematical science of computation. *Information processing*, 62:21–28, 1962.

[133] Stephen J. Mellor and Marc Balcer. *Executable UML - A Foundation for Model-Driven Architecture*. Addison-Wesley, 2002.

[134] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152, 2006.

[135] Marjan Mernik. *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*. IGI Global, 2013.

[136] Marjan Mernik. An object-oriented approach to language compositions for software language engineering. *Journal of Systems and Software*, 86(9):2451 – 2464, 2013.

[137] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[138] Marjan Mernik and Viljem Zumer. Reusability of formal specifications in programming language description. In *8th Annual Workshop on Software Reuse, WISR8*, pages 1–4, 1997.

[139] Metamodel Zoos. http://www.emn.fr/z-info/atlanmod/index.php/Zoos, last access: april 2015.

[140] Bertrand Meyer. *Eiffel: the language*. Prentice-Hall, Inc., 1992.

[141] Bertrand Meyer. Multi-language programming: how .NET does it. *Software Development (3-part article Part 1: Polyglot Programming; Part 2: Respecting other object models; Part 3: Interoperability: at what cost, and with whom?)*, May, June and July 2002.

[142] Leo A. Meyerovich and Ariel S. Rabkin. Socio-PLT: Principles for Programming Language Adoption. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! '12, pages 39–54. ACM, 2012.

[143] Leo A Meyerovich and Ariel S Rabkin. Empirical analysis of programming language adoption. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 1–18. ACM, 2013.

[144] Robin Milner. *A calculus of communicating systems*. Springer, 1982.

[145] Robin Milner. *Communication and concurrency*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1995.

[146] Marvin Minsky. Matter, mind, and models. *Semantic Information Processing*, pages 425–432, 1968.

[147] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving Executability into Object-Oriented Meta-Languages. In *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278. Springer, 2005.

[148] Gunter Mussbacher, Daniel Amyot, Ruth Breu, Jean-Michel Bruel, Betty Cheng, Philippe Collet, Benoit Combemale, Robert France, Rogardt Heldal, James Hill, Jörg Kienzle, Matthias Schöttle, Friedrich Steimann, Dave Stikkolorum, and Jon Whittle. The Relevance of Model-Driven Engineering Thirty Years from Now. In Juergen Dingel and Wolfram Schulte and Isidro Ramos and Silvia Abrahão and Emilio Insfran, editor, *17th International*

Conference on Model Driven Engineering Languages and Systems (MODELS 2014), volume 8767 of *Model-Driven Engineering Languages and Systems*, page 18, Valencia, Spain, September 2014. Springer International Publishing Switzerland.

[149] L. Northrop, P. Feiler, R. P. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, D. Schmidt, K. Sullivan, and K. Wallnau. Ultra-Large-Scale Systems - The Software Challenge of the Future. Technical report, Software Engineering Institute, 2006.

[150] Object Management Group, Inc. *Meta Object Facility (MOF) 2.0 Core Specification*, 2006.

[151] Object Management Group, Inc. *Unified Modeling Language (UML) 2.1.2 Superstructure*, 2007.

[152] Object Management Group, Inc. *MOF 2.0 Query/ View/ Transformation (QVT) Specification*, 2008.

[153] Object Management Group, Inc. *UML Object Constraint Language (OCL) 2.4 Specification*, 2014.

[154] OMG. *UML Testing Profile 1.0 Specification*, 2005.

[155] OMG. *Semantics of a Foundational Subset for Executable UML Models (fUML), Version 1.0*, 2011.

[156] V. Papailiopoulou, D. Potop-Butucaru, Y. Sorel, R. De Simone, L. Besnard, and J. Talpin. From design-time concurrency to effective implementation parallelism: The multi-clock reactive case. In *ESLsyn*, pages 1–6, 2011.

[157] Craig M. Pease and James J. Bull. *Scientific Decision-Making*. Online E-Book.

[158] C. A. Petri. Introduction to general net theory. In *Advanced Course: Net Theory and Applications*, pages 1–19, 1975.

[159] Gordon D Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 1981.

[160] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *5th Colloquium on International Symposium on Programming*, pages 337–351. Springer, 1982.

[161] J. Raul Romero, Jose E. Rivera, Francisco Duran, and Antonio Vallecillo. Formal and Tool Support for Model Driven Engineering with Maude. *Journal of Object Technology, TOOLS EUROPE 2007*, 6(9):187–207, 2007.

[162] Tiark Rompf and Martin Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. *SIGPLAN Not.*, 46(2):127–136, October 2010.

[163] G. Rozenberg, editor. *Handbook of graph grammars and computing by graph transformation: volume I. foundations*. World Scientific Publishing Co., Inc., 1997.

[164] Jesús Sánchez Cuadrado and Jesús García Molina. Approaches for model transformation reuse: Factorization and composition. In *ICMT*, 2008.

[165] João Saraiva. Component-based programming for higher-order attribute grammars. In *Proc. of GPCE*, pages 268–282, 2002.

[166] Douglas C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2):25–31, February 2006.

[167] Eric Schulte, Zachary Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. Software mutational robustness and proactive diversity. Technical report, Technical Report. University of New Mexico. http://www. cs. unm. edu/˜ forrest/classes/readings/mutation. pdf, 2011.

[168] Dana S Scott and Christopher Strachey. *Toward a mathematical semantics for computer languages*. Oxford University Computing Laboratory, Programming Research Group, 1971.

[169] Ed Seidewitz. Uml with meaning: Executable modeling in foundational uml and the alf action language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 61–68, New York, NY, USA, 2014. ACM.

[170] Sagar Sen, Naouel Moha, Benoit Baudry, and Jean-Marc Jézéquel. Meta-model Pruning. In *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, 2009.

[171] Sagar Sen, Naouel Moha, Vincent Mahé, Olivier Barais, Benoit Baudry, and Jean-Marc Jézéquel. Reusable model transformations. *International Journal on Software and Systems Modeling (SoSyM)*, 11(1), 2010.

[172] A. Shaikh, R. Clarisó, U.K. Wiil, and N. Memon. Verification-driven slicing of UML/OCL models. In *Proceedings of the IEEE/ACM international conference on Automated Software Engineering*, ASE '10, pages 185–194. ACM, 2010.

[173] A. Shaikh, U.K. Wiil, and N. Memon. Evaluation of Tools and Slicing Techniques for Efficient Verification of UML/OCL Class Diagrams. *Adv. Soft. Eng.*, 2011:5:1–5:18, 2011.

[174] M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Software Engineering*, 1995.

[175] Macneil Shonle, Karl Lieberherr, and Ankit Shah. XAspects: An extensible system for domain-specific aspect languages. In *Companion of the 18th annual ACM SIGPLAN conference OOPSLA*, pages 28–37. ACM, 2003.

[176] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 124–134, New York, NY, USA, 2011. ACM.

[177] B. Solomon, D. Ionescu, M. Litoiu, and M. Mihaescu. Model-driven engineering for autonomic provisioned systems. In *COMPSAC'08*.

[178] H. Stachowiak. *Allgemeine Modelltheorie*. Springer-Verlag, 1973.

[179] Jim Steel. *Model Typing*. PhD thesis, Université de Rennes 1, April 2007.

[180] Jim Steel and Jean M. Jézéquel. On model typing. *International Journal on Software and Systems Modeling (SoSyM)*, 6(4), 2007.

[181] Andreas Stefik and Susanna Siebert. An Empirical Investigation into Programming Language Syntax. *Trans. Comput. Educ.*, 13(4):19:1–19:40, November 2013.

[182] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[183] W. Sun, R. France, and I. Ray. Rigorous Analysis of UML Access Control Policy Models. In *IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY)*, pages 9–16, 2011.

[184] Wuliang Sun, Benoit Combemale, Robert B. France, Arnaud Blouin, Benoit Baudry, and Indrakshi Ray. Using Slicing to Improve the Performance of Model Invariant Checking. *Journal of Object Technology*, page 28, 2015.

[185] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. Generic model transformations: Write once, reuse everywhere. In *ICMT*, 2011.

[186] J. Thomas and K. Cook, editors. *Illuminating the Path: Research and Development Agenda for Visual Analytics*. IEEE Press, 2005.

[187] Juha-Pekka Tolvanen and Matti Rossi. MetaEdit+: defining and using domain-specific modeling languages and code generators. In *Companion of the 18th annual ACM SIGPLAN conference OOPSLA*, pages 92–93. ACM, 2003.

[188] Laurence Tratt. Domain Specific Language Implementation via Compile-time Meta-programming. *ACM Trans. Program. Lang. Syst.*, 30(6):31:1–31:40, October 2008.

[189] Edoardo Vacchi and Walter Cazzola. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures*, 2015.

[190] Edoardo Vacchi, Walter Cazzola, Benoit Combemale, and Mathieu Acher. Automating Variability Model Inference for Component-Based Language Implementations. In Patrick Heymans and Julia Rubin, editors, *18th International Software Product Line Conference (SPLC 2014)*, Florence, Italie, September 2014. ACM.

[191] Edoardo Vacchi, Walter Cazzola, Suresh Pillay, and Benoit Combemale. Variability Support in Domain-Specific Language Development. In Richard F. Paige Martin Erwig and Eric van Wyk, editors, *6th International Conference on Software Language Engineering (SLE 2013)*, Lecture Notes in Computer Science, Indianapolis, USA, 2013. Springer-Verlag.

[192] Matias Vara Larsen, Julien Deantoni, Benoit Combemale, and Fr'ed'eric Mallet. D3.1.2 - Language Composition Operator. Technical report, 2014. http://gemoc.org/wp-content/uploads/2015/07/D3.1.2.pdf.

[193] Matias Vara Larsen and Arda Goknil. Railroad Crossing Heterogeneous Model. In *GEMOC workshop*, 2013.

[194] Matias Ezequiel Vara Larsen, Julien Deantoni, Benoit Combemale, and Frédéric Mallet. A Behavioral Coordination Operator Language (BCOoL). In *18th International Conference on Model Driven Engineering Languages and Systems (MODELS 2015)*, August 2015.

[195] Dániel Varró and András Pataricza. Generic and meta-transformations for model transformation engineering. In *UML*, 2004.

[196] Markus Voelter. Language and IDE modularization, extension and composition with MPS. *Generative and Transformational Techniques in Software Engineering*, 2011.

[197] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[198] Markus Voelter, Daniel Ratiu, Bernd Kolb, and Bernhard Schätz. mbeddr: instantiating a language workbench in the embedded software domain. *Autom. Softw. Eng.*, 20(3):339–390, 2013.

[199] Markus Voelter and Konstantin Solomatov. Language modularization and composition with projectional language workbenches illustrated with MPS. In Mark van den Brand, Brian Malloy, and Steffen Staab, editors, *Third International Conference on Software Language Engineering (SLE 2010)*, Lecture Notes in Computer Science. Springer, 2010.

[200] Markus Völter and Eelco Visser. Language extension and composition with language workbenches. In *Proc. of the OOPSLA companion*, pages 301–304. ACM, 2010.

[201] Dean Wampler. Polyglot programming http://www.polyglotprogramming.com/.

[202] M. P Ward. Language-oriented programming. *Software-Concepts and Tools*, 15(4):147–161, 1994.

[203] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[204] Jon Whittle, John Hutchinson, and Mark Rouncefield. The state of practice in model-driven engineering. *Software, IEEE*, 31(3):79–85, 2014.

[205] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, J. Schoenboeck, and Wieland Schwinger. From the heterogeneity jungle to systematic benchmarking. In *MoDELS Workshops*, 2010.

[206] Manuel Wimmer, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, Wieland Schwinger, Jesús Cuadrado, Esther Guerra, and Juan de Lara. Reusing model transformations across heterogeneous metamodels. In *International Workshop on Multi-Paradigm Modeling*, 2011.

[207] Glynn Winskel. Event structures. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency*, Lecture Notes in Computer Science. Springer, 1987.

[208] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT press, 1993.

[209] Sun Wuliang, Benoit Combemale, Steven Derrien, and Robert France. Contract-Aware Substitutability of Modeling Languages. In *9th European Conference on Modelling Foundations and Applications (ECMFA 2013)*, LNCS, Montpellier, France, 2013. Springer.

[210] Bernard P. Zeigler, Herbert Praehofer, and Tag G. Kim. *Theory of Modeling and Simulation, Second Edition*. Academic Press, 2 edition, January 2000.

[211] Srđan Živković and Dimitris Karagiannis. Towards metamodelling-in-the-large: Interface-based composition for modular metamodel development. In *Enterprise, Business-Process and Information Systems Modeling*, pages 413–428. Springer, 2015.

List of Figures

Abstract

Software engineering faces new challenges with the advent of modern software-intensive systems such as complex critical embedded systems, cyber-physical systems and Internet of things. Application domains range from robotics, transportation systems, defense to home automation, smart cities, and energy management, among others. Software is more and more pervasive, integrated into large and distributed systems, and dynamically adaptable in response to a complex and open environment. As a major consequence, the engineering of such systems involves multiple stakeholders, each with some form of domain-specific knowledge, and with an increasingly use of software as an integration layer.

Model-Driven Engineering (MDE) aims at reducing the accidental complexity associated with developing complex software-intensive systems through the use of modeling techniques that support separation of concerns and automated generation of system artifacts from models. Separation of concerns is founded on the exploitation of different *domain-specific modeling languages* (DSMLs), each providing constructs based on abstractions that are specific to a concern of a system. As such, DSMLs are "the heart and soul" of MDE, and have major consequences on the industrial development processes.

The integration of domain-specific concepts and best practices development experience into DSMLs can significantly improve software and systems engineers productivity and system quality. Yet, the development of DSMLs has only been recently recognized as a significant software engineering task itself. Indeed, the development of DSMLs is a challenging task which requires specialized knowledge. This recently resulted in the emergence of *Software Language Engineering* (SLE), defined as the application of systematic, disciplined, and measurable approaches to the development, use, deployment, and maintenance of languages.

MDE encompasses the identification of added-value domain knowledge, and SLE covers the technical activities to develop DSMLs. However, the interplay between both is still challenging: how to turn application-level and business knowledge into added-value DSMLs? How to scale-up with their multiplication due to the various stakeholders and application domains of interest? Answering these questions is at the core of my research activities for 10 years. The challenge is twofold: to capture domain-specific knowledge into added-value DSMLs while supporting their possible reuse and customization in other contexts; and to support the separation of concerns through the use of multiple DSMLs while ensuring their automatic and correct integration.

In this *habilitation à diriger des recherches* (HDR), I review a decade of research work in the fields of MDE and SLE. I propose contributions to support a *language-oriented modeling*, with the particular focus on enabling early validation & verification (V&V) of software-intensive systems. I first present foundational concepts and engineering facilities which help to capture the core domain knowledge into the various heterogeneous concerns of DSMLs (aka. *metamodeling in the small*), with a particular focus on executable DSMLs to automate the development of dynamic V&V tools. Then, I propose structural and behavioral DSML interfaces, and associated composition operators to reuse and integrate multiple DSMLs (aka. *metamodeling in the large*).

In these research activities I explore various breakthroughs in terms of modularity and reusability of DSMLs. I also propose an original approach which bridges the gap between the concurrency theory and the algorithm theory, to integrate a formal concurrency model into the execution semantics of DSMLs. All the contributions have been implemented in software platforms – the language workbench *Melange* and *the GEMOC studio* – and experienced in real-world case studies to assess their validity. In this context, I also founded *the GEMOC initiative*, an attempt to federate the community on the grand challenge of *the globalization of modeling languages*.