HAL
open science

Towards Formal System Modeling: Making Explicit and Formal the Concurrent and Timed Operational Semantics to Better Understand Heterogeneous Models

Julien Deantoni

▶ **To cite this version:**

Julien Deantoni. Towards Formal System Modeling: Making Explicit and Formal the Concurrent and Timed Operational Semantics to Better Understand Heterogeneous Models. Embedded Systems. Université Côte d'Azur, CNRS, I3S, France, 2019. tel-02427962

HAL Id: tel-02427962

https://inria.hal.science/tel-02427962v1

Submitted on 4 Jan 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Towards Formal System Modeling

Making Explicit and Formal the Concurrent and Timed Operational Semantics to Better Understand Heterogeneous Models

Julien Deantoni

Habilitation à Diriger les Recherches

Habilitation committee:	Alain Girault, Inria Grenoble Rhône-Alpes,	reviewer
	Hans Vangheluwe, University of Antwerp,	reviewer
	Reinhard Von Hanxleden, University of Kiel,	reviewer
	Frédéric Mallet, University of Cote d'Azur,	examiner
	Jean-Paul Rigault, University of Cote d'Azur,	examiner
	Benoit Combemale, University of Toulouse Jean Jaurès,	invited

Contents

1 Introduction 1
- 1.1 Context . . . 1
- 1.2 Objectives and Challenges . . . 2
- 1.3 Facts About the Context of my Research . . . 5
  - 1.3.1 Research projects . . . 5
  - 1.3.2 Main invited seminar/workshops . . . 6
  - 1.3.3 PhD student supervision . . . 6
  - 1.3.4 Softwares . . . 7
- 1.4 Wrap Up and Outline of the Document . . . 8

2 Modeling Concurrent and Timed Execution of Models 9
- 2.1 Introduction . . . 9
- 2.2 Clock Constraint Specification Language . . . 10
  - 2.2.1 Multiform logical time . . . 11
  - 2.2.2 Instants and Time Structure . . . 11
  - 2.2.3 Clocks . . . 12
  - 2.2.4 Clock Constraints Specification . . . 13
  - 2.2.5 Definition of a specific subset of instants . . . 14
  - 2.2.6 Clock relations . . . 15
  - 2.2.7 Clock expressions . . . 16
  - 2.2.8 Temporal evolutions . . . 17
  - 2.2.9 CCSL libraries . . . 17
- 2.3 Comparing CCSL and temporal logics . . . 17
  - 2.3.1 Property Specification Language . . . 18
  - 2.3.2 Comparing PSL and CCSL . . . 19
  - 2.3.3 Sum-up about CCSL and PSL . . . 20
- 2.4 TimeSquare: the CCSL tool . . . 21
  - 2.4.1 CCSL Implementation . . . 22
  - 2.4.2 Simulation . . . 22
  - 2.4.3 Back-ends . . . 23
  - 2.4.4 Analysis of existing traces . . . 25
  - 2.4.5 Trace Reconciliation . . . 26
  - 2.4.6 Sum-up about TimeSquare . . . 26
- 2.5 Using CCSL in different domains . . . 27
  - 2.5.1 Behavior model for SDF . . . 27
  - 2.5.2 Behavior model of AADL . . . 31
  - 2.5.3 behavior model of Repetitive Structure Modeling (RSM) models and their execution platform . . . 35
- 2.6 Towards Multi-View Behavior Modeling . . . 37
  - 2.6.1 Background and related work . . . 38
  - 2.6.2 PRISMSYS: A modeling multi-view framework for systems . . . 39
  - 2.6.3 PRISMSYS Execution Semantics . . . 40
  - 2.6.4 PRISMSYS Model Co-Simulation . . . 43
  - 2.6.5 Case Study: CPU thermal controller . . . 44
- 2.7 Conclusion . . . 46

3 Modeling Concurrent and Timed Operational Semantics of Languages 47
3.1 Introduction . . . 48
3.2 Ingredients of a Concurrency-Aware Executable DSML . . . 49
3.2.1 Background Knowledge . . . 49
3.2.2 Language Units Identification . . . 49
3.2.3 Reifying Language Units Coordination . . . 51
3.3 A Language Workbench to Design and Implement Concurrency-Aware Executable DSMLs . . . 53
3.3.1 Abstract Syntax Design . . . 53
3.3.2 Domain Specific Actions Design . . . 54
3.3.3 Model of Concurrency and Communication + DSE Design . . . 55
3.4 Integration of the approach in the GEMOC Studio . . . 65
3.4.1 Overview of the GEMOC Studio . . . 65
3.4.2 Overview of the Execution Framework . . . 65
3.4.3 Interface of the Execution Framework . . . 66
3.4.4 The concurrent and timed Execution Engines . . . 67
3.4.5 Runtime Services linked to the concurrent engine . . . 67
3.5 Validation and Discussion . . . 69
3.6 Related Work . . . 70
3.7 Conclusion about the Modeling of Concurrent and Timed Operational Semantics . . . 71

4 Coordination Patterns between Heterogeneous Languages 73
4.1 Introduction . . . 73
4.2 Background on Existing Approaches . . . 74
4.2.1 Composition Approaches . . . 74
4.2.2 Coordination Approaches . . . 77
4.2.3 Leason Learned From Existing Approaches . . . 82
4.3 Language Behavioral Interface . . . 83
4.4 B-COoL . . . 85
4.4.1 Overview . . . 85
4.4.2 Abstract Syntax of B-COoL . . . 86
4.4.3 Execution Semantics . . . 88
4.5 Validation of the Approach . . . 90
4.5.1 Definition of Coordination Operators between the TFSM and fUML Language . . . 90
4.5.2 Use of Coordination Operators in a Surveillance Camera System . . . 92
4.5.3 Use of the Coordination Specification . . . 93
4.5.4 Comparison with Existing Approaches . . . 93
4.6 Implementation . . . 93
4.7 Conclusion . . . 95

5 Conclusion 99

6 Perspectives 103
6.1 Perspectives About CCSL evolutions/reasoning . . . 103
6.1.1 Exploitation of the CCSL clock graph . . . 103
6.1.2 Extension transition system and/or temporal logic for CCSL model checking . . . 105
6.1.3 Multi Physical Dimensions Associated to Simulation Speedup . . . 106
6.2 Perspective About the Modeling of Structural Operational Semantics . . . 108
6.2.1 Formal specification or abstraction of rewriting rules . . . 108
6.2.2 Clarification of the links between various “views” over the semantics . . . 108
6.2.3 Integration of the various metalanguages used for SOS modeling . . . 109
6.2.4 Helping Writing/Debugging The Operational Semantics . . . 110
6.3 Perspectives About Heterogeneous Modeling and Simulation . . . 111
6.3.1 Support for Data Coordination . . . 111
6.3.2 Understand the models “Greyification” for Correct and Formal Coordination . . . 112
6.4 Thoughts About Formal System Engineering . . . 114

Bibliography 117

Chapter 1

Introduction

Contents

1.1 Context 1
1.2 Objectives and Challenges 2
1.3 Facts About the Context of my Research 5
1.4 Wrap Up and Outline of the Document 8

1.1. Context

We are more and more interacting with non-natural complex systems, thought up by the human brain like smartphones, industrial robots, vehicles or home automation. Both words *complex* and *system* must be defined to better understand the context of my research in the last 10 years.

According to the Merriam-Webster dictionary, a system is "*a regularly interacting or interdependent group of items forming a unified whole*". It comes from the Greek *sýstima* ,defined by the Centre for the Greek Language[1] as *"A set of bodies, things, concepts or processes that are interdependent, so that any change in one of them has an effect on one or all of the others"*.

It is easy to understand from these definitions that speaking about *system engineering* is speaking about an interdisciplinary approach that deals with a set of tangled elements (or parts). In a system the parts can include very different kinds of entities amongst which: different kinds of software (*e.g.*, a control software or a data processing software); different kinds of hardware (*e.g.*, a System on Chip architecture or a topology of communicating nodes); different kinds of physical entities (*e.g.*, an aircraft nacelle structure or the thermal dissipation of a CPU); different kinds of actors (*e.g.*, the pilot or the mechanic of an aircraft).

It is also important to note that due to the increasing size and complexity of modern systems, the parts of the systems are not specified in a unique model by using a unique language but are split into different views where each view focuses on a specific concern of one or more parts. For instance, a CPU can be described in different views by its instruction set, its internal hardware structure, its package or a set of physical properties like its temperature dissipation or its electric consumption. All these views are different focuses on the same part (*i.e.*, CPU) that are useful at different stages of the system life cycle.

Concerning the etymology of *complex*, it is composed of the Latin words "com" (meaning: "together") and "plex" (meaning: "woven"). It is a little bit redundant with the notion of system itself but emphasizes the intricate nature of the different concerns composing the system.

In my research I focused on software-intensive systems, defined by the IEEE 1471 [92] as: *"A software-intensive system is any system where software contributes essential influences to the design, construction, deployment, and evolution of the system as a whole"*.

Nowadays, there are various stakeholders involved in the development process of such systems. Each stakeholder focuses on a specific domain of expertise of the system as defined in the IEEE 42010 [93] standard about system and software engineering. Each stakeholder is developing model(s) by using dedicated languages tailored to their domains both in terms of the concepts handled but also of their behavioral semantics. Each model is referring to various parts represented according to the specific view they belong to.

[1] `http://www.greek-language.gr/greekLang/portal/about_us/index.html`

Finally, the system is then specified by the consistent composition of the different heterogeneous models developed by each of the stakeholders. The languages used also evolve along the development process. For instance the models developed to provide an operational architecture are not specified in the same language as the models that define the algorithms of the logical architecture. Anyway, the models developed in these heterogeneous languages must be consistent, not only in terms of syntax but also in terms of behavioral semantics. Supporting the coordinated use of such Domain Specific Modeling Languages (DSML) leads to what we call the globalization of modeling language [46].

1.2. Objectives and Challenges

The development of software-intensive systems is facing many challenges arising from both the growing complexity of the modern systems and the interdisciplinary nature of the approach. A general objective for my research in the last 10 years was to go towards a formal model based system engineering where various languages, syntactically and semantically adequate to a given task, and the relations between them and their conforming models are precisely defined to enable automatic reasoning as well as Verification and Validation (V&V) activities all along the development process.

Behind this general objective, there is the conviction that the use of appropriate languages[2] at the appropriate level of abstraction together with an appropriate development process and supporting tooling can help to solve the socio technical coordination needed in interdisciplinary developments [133].

To achieve the objective describe below, there are a large variety of sub challenges. The goal of this section is not to provide an exhaustive list of them but rather to present some challenges I addressed, sometimes partially in my research work.

Challenge #1 When a behavioral model is defined, its structure is explicitly provided and manipulated by different actors (being tools or users). However, the way it behaves is usually not part of the model itself. At best, the behavioral semantics of the modeling language is available in a non-ambiguous form. However, most often, it is either written in natural language or encoded (in an opaque way) in a tool. This makes difficult the consistent manipulation/analysis of the model by different tools. What is missing is a language agnostic and explicit description of the model behavior. It could then be used as a golden reference by other tools to avoid different interpretation of the model or used by generic tools to reason on the model (see Figure 1.1).

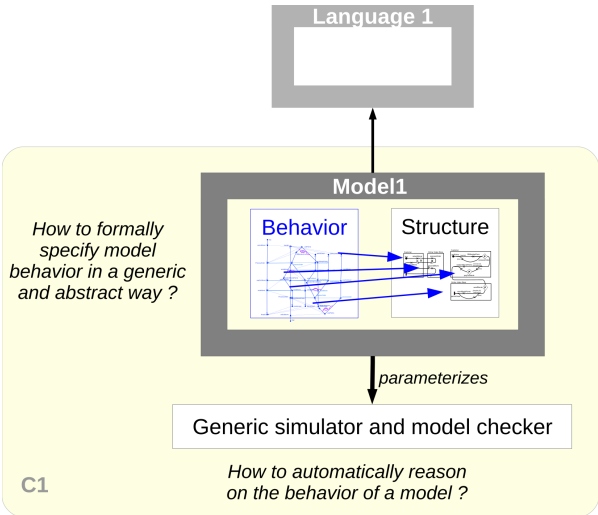

Figure 1.1: Overview of Challenge #1

- From a scientific point of view, the behavior of a model must not be specified by one potential totally ordered trace but by the (possibly infinite) set of all its acceptable traces (*i.e.*, by the traces defined by the behavioral semantics of the language). This is a crucial point to avoid making a

[2] of course we consider in this document that a language is defined by both a syntax and a semantics

priori restrictions at the early stage of the development process. It requires to find the appropriate level of abstraction for the specification of the model behavior. Also, this specification must be formal to avoid a lame behavior to be used as reference.

- From a technical point of view, it requires to provide a representation of the behavior which is associated to the model structure in order to provide a consistent set of artifacts, which represents the model structure together with the way it behaves. Based on such set of artifacts, it must be possible to run some simulations and/or V&V activities; independently of the development stage the model belongs to.

The beginning of Chapter 2 presents the most significant pieces of work I realized to address this challenge.

Challenge #2 There are usually many heterogeneous models to specify a system. These models refer to different parts of the system at different abstraction levels. In order to understand the emerging behavior of the system, it is important to understand the nature of the various correspondences that exist between all these models as well as their semantics impact/role with respect to the behavior of related models. it is needed for early V&V and rigorous refinement process and will make possible, at the system level, to understand the impacts of a decision made on a specific model on the other models (see Figure 1.2).

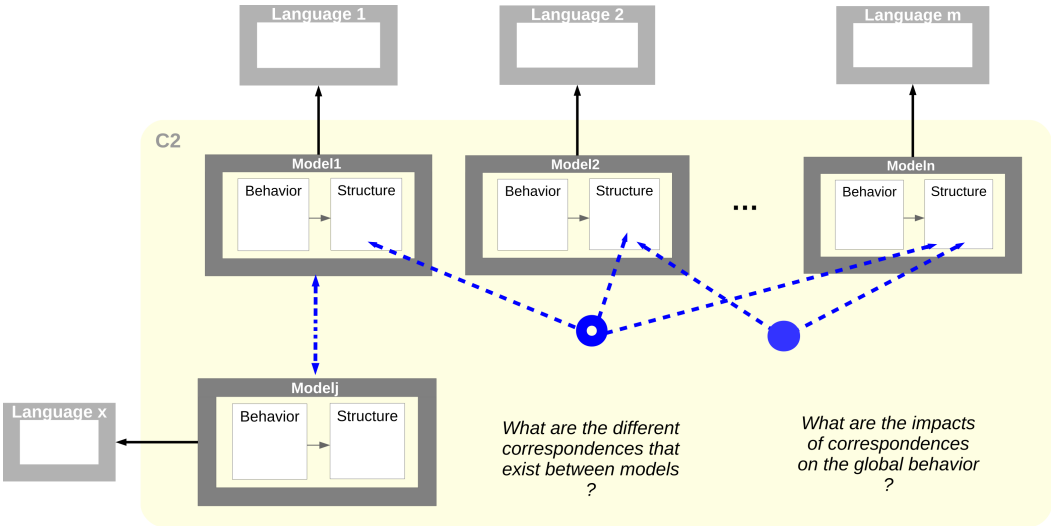

Figure 1.2: Overview of Challenge #2

- From a scientific point of view, it means that all the correspondences between models should be characterized precisely and in a systematic way. This characterization may provide appropriate (formal) annotations to specify the semantics of such correspondences. This study should encompass both "horizontal" correspondences, *i.e.*, correspondences between models at the same abstraction level; and "vertical" correspondences, *i.e.*, correspondences between a model and the model or the set of models that refines it. It should then be possible to manipulate, infer, or generate a "glue" (*e.g.*, synchronizations) or some relations (*e.g.*, the (bi)simulation relation or the preserving of some properties) between the behaviors of different models (whose elements are) linked by some correspondences.
- From a technical point of view, it should be possible to specify correspondences between models specified in different languages. It should also be possible to access to the associated glue or relations. From the semantics of the correspondences, a technical environment should allow the co-simulation or co-validation of heterogeneous models linked by some "horizontal" correspondences; and should allow the verification of the relations or properties inferred from the "vertical" correspondences between models.

The end of Chapter 2 summarizes some of the work I realized to address this challenge.

Challenge #3 Nowadays, when developing a language, there exist metalanguages that can be used to specify the syntax of the language and from which many of the tooling is automatically generated/provided in a generic way. This is for example the case when using Xtext[3]. It requires a specification of the language grammar and can automatically generate from it a parser, a linker, a typechecker and advanced editors with syntactic coloration, completion and quick fixes. Equivalent facilities do no exist for the behavioral semantics specification. There exist few metalanguages to fully specify the behavioral semantics of a language (either operational, denotational or axiomatic). The main drawback of existing approaches is their inappropriateness to make explicit and platform/language independent the temporal and concurrent aspects of the behavioral semantics, specially when the system is non-deterministic[4]. There is a need to make explicit these concerns in the specification of the behavioral semantics of the language to ease taming challenge #1 and to enable the automatic generation of tooling like for instance compilers, interpreters, model checkers or omniscient debuggers (see Figure 1.3).

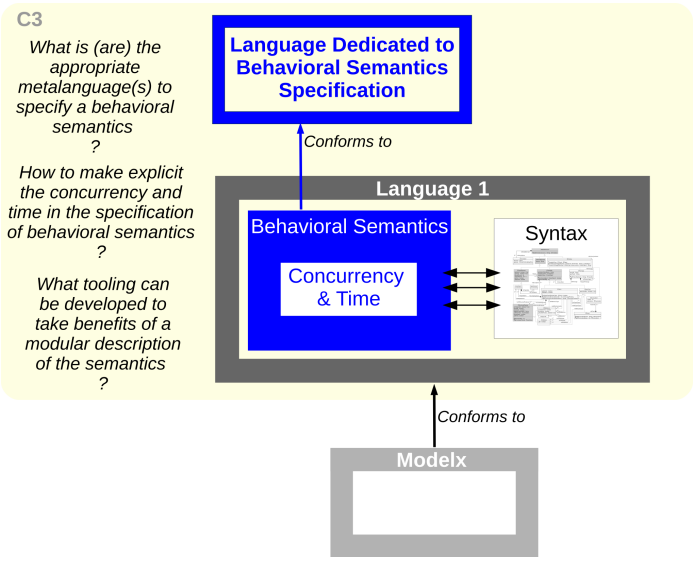

Figure 1.3: Overview of Challenge #3

- From a scientific point of view, this means that new metalanguage(s) must be defined to make concurrent and temporal aspects explicit in the definition of a behavioral semantics. This requires understanding what are the appropriate and minimal set of concepts to be manipulated to specify concurrency and time.
- From a technical point of view, this means that the new metalanguages must be seamlessly integrated with the other metalanguages (for instance used to define the syntax of the language). It also requires new tooling of the metalanguage(s) to bring benefits of using the proposed metalanguage(s).

Chapter 3 presents the most significant pieces of work I realized to address this challenge.

Challenge #4 Dealing with the various models of a system makes its development complex since a system designer has to deal with a set of heterogeneous models but also with the interaction between them. In this context, manually creating the correspondences between a large set of complex models is necessary (see challenge #2) but is a tedious and error prone task. Current approaches to automate the coordination are bound to a fixed set of coordination patterns and are not appropriated to the addition of models written in new languages. Additionally, existing approaches encode the coordination pattern(s) into a tool thus limiting reasoning on the global system behavior. What is missing is a way to specify coordination patterns between languages in a formal way so that correspondences on models can be generated automatically for an arbitrary set of models (see Figure 1.4).

[3] `http://eclipse.org/xtext`

[4] it is important to realize that there are many programs/models that are non deterministic from a trace point of view but functionally deterministic in the sense that all the acceptable traces produce the same outputs from a same set of inputs and initial conditions.

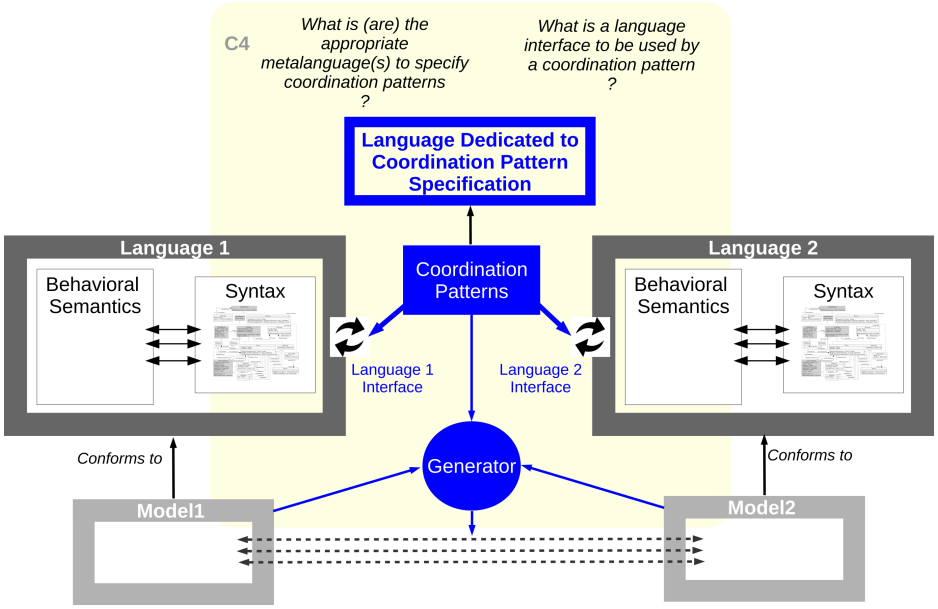

Figure 1.4: Overview of Challenge #4

- From a scientific point of view, It is required to provide a system designer with a dedicated language, which can be used to specify coordination patterns between different languages. This means that language interface(s) must be clearly defined and exposed to the system designer. This interface must allow querying (parts of) the syntax and (parts of) the behavioral semantics of the language. The definition of the patterns must be flexible enough to fit any new language and the coordination must be done in a non restrictive way, like for the behavioral semantics of a language, allowing for non determinism if needed. Additionally, the semantics of the proposed language must be (formally) defined to allow unambiguous tooling. Finally, it can be interesting to understand if it makes sense to specify patterns eventually resulting in a specific kind of "vertical" correspondences.
- From a technical point of view, the language interface must be reified and accessible in the system designer pattern specification. It must also be possible to automatically apply the patterns on specific models to generate the coordination.

the beginning of chapter 4 presents the most significant pieces of work I realized to address this challenge.

Before elaborating on the different challenges in the next three chapters of this paper, I quickly overview the context in which my research was realized.

1.3. Facts About the Context of my Research

The challenges described in the previous section have been (sometimes partially) addressed in different research contexts but most often based on discussion with the Kairos team members (previously named Aoste), PhD students and other national and international researchers from the community. Before describing my research I give in this section an overview of the context in which it has been realized.

1.3.1. Research projects

GLOSE *(2018-...)*

- Globalization in System Engineering

- bilateral collaboration INRIA/Safran (`http://gemoc.org/glose`)

CLARITY *(2014-2017)*

- éCosystème pour la pLAte-foRme d'Ingénierie sysTème melodY
- LEOC project (`http://clarity-se.org`)

GEMOC *(2012-2016)*

- A Generic Framework for Model Execution and Dynamic Analysis
- ANR Project, Program INS (`http://www.gemoc.org/ins`)

HOPE *(2012-2016)*

- Hierarchically Organized Power/Energy management
- ANR Project, Program INS (`http://anr-hope.unice.fr/?lang=fr`)

RT-SIMEX *(2009-2012)*

- Retro-engineering and analysis of simulation and execution traces from real time embedded systems (Principal Investigator)
- ANR Project, Program INS (`http://www.rtsimex.org/`)

HELP *(2010-2013)*

- High lEvel modeling for Low Power systems
- ANR Project, Program INS (`http://www-verimag.imag.fr/PROJECTS/SYNCHRONE/HELP/`)

AS GEMOC *(2011)*

- Survey on Heterogeneous Modeling
- CNRS Specific Action from GDR GPL (`http://www.gemoc.org/as2011`)

1.3.2. Main invited seminar/workshops

2019 5d., CAMPaM : Computer Automated Multi-Paradigm Modelling Workshop *Bellairs - Barbados*

2018 2d., Co-simulation in MPM4CPS *Skopje - Macedonia*

2018 2d., Workshop on Multi Paradigm For Cyber Physical Systems *Riga - Latvia*

2017 1d., Optimizing Real-Time Systems *Paris - France*

2017 5d., CAMPaM : Computer Automated Multi-Paradigm Modelling Workshop *Bellairs - Barbados*

2015 5d., Concern-Oriented Reuse Workshop *Bellairs - Barbados*

2014 3d., Globalizing Domain-Specific Languages *Dagstuhl - Allemagne #14412*

2014 4d., ECNU LIAMA summer school: Hands on Logical Time for Event Based Semantics Specification *Shangai - China*

2011 4d., Modeling time and constraints in MARTE *Real Time Summer school, demo of TimeSquare*

2007 4d., Software architecture for embedded real time systems *Real Time Summer school*

1.3.3. PhD student supervision

2018–... Giovanni Liboni

- *Composition of Cyber Models for Efficient and Accurate Co-Simulation*
- CIFRE with Safran Tech
- Co-supervised with Frédéric Mallet

2012–2016 Matias Ezequiel Vara Larsen

- *Behavioral Composition of Executable Models*
- founded on the GEMOC project
- Co-supervised with Frédéric Mallet

2010–2013 Carlos Ernesto Gomez Cardenas

- *Modeling Functional and Non-Functional Properties of Systems in a Multi-View Approach*
- grant from the French government
- Co-supervised with Frédéric Mallet

1.3.4. Softwares

Since my arrival in the Kairos team, I always promoted the development of research softwares to demonstrate the applicability of our research work. Additionally, it helps creating a common technological background in the team, helping thus the creation of new prototypes. I'm now in charge of maintaining most of these softwares. While interesting it is unfortunately time consuming and it is sometimes difficult to choose between maintenance and new development/research work.

1.3.4.1. TimeSquare

Authors: Nicolas Chleq, Julien Deantoni, Frédéric Mallet, Benoît Ferrero, Charles André.

TimeSquare is a software environment for the modeling and analysis of timing constraints in embedded systems. It relies specifically on the Time Model of the Marte UML profile, and more accurately on the associated Clock Constraint Specification Language (CCSL) for the expression of timing constraints.

`http://timesquare.inria.fr`

1.3.4.2. BCOol

Authors: Julien Deantoni, Matias Vara Larsen, Benoît Combemale, Didier Vojtisek.

BCOoL is a tool-supported meta-language dedicated to the specification of language coordination patterns to automatically coordinates the execution of, possibly heterogeneous, models. `http://www.gemoc.org`

1.3.4.3. MoCCML

Authors: Julien Deantoni, Didier Vojtisek, Joël Champeau, Benoît Combemale, Stephen Creff.

The MoCCML / Concurrency provides components and engines supporting concurrency and/or time in execution semantics.

`http://timesquare.inria.fr/moccml`

1.3.4.4. GEMOC Studio

Authors: Didier Vojtisek, Benoît Combemale, Cédric Brun, François Tanguy, Joël Champeau, Julien Deantoni, Xavier Crégut.

The GEMOC Studio is an eclipse package that contains components supporting the GEMOC methodology for building and composing executable Domain-Specific Modeling Languages (DSMLs). It includes two workbenches: The GEMOC Language Workbench: intended to be used by language designers (aka domain experts), it allows to build and compose new executable DSMLs. The GEMOC Modeling Workbench: intended to be used by domain designers to create, execute and coordinate models conforming to executable DSMLs. The different concerns of a DSML, as defined with the tools of the language workbench, are automatically deployed into the modeling workbench. They parametrize a generic execution framework that provides various generic services such as graphical animation, debugging tools, trace and event managers, timeline, etc.

`http://gemoc.org/studio.html`

1.4. Wrap Up and Outline of the Document

Before quickly describing the outline of this document, I have to admit that it is difficult to evaluate the impact of the several discussions I had on my research directions. I think it is really important to take time to exchange idea and debate with different people. I also think that the latin sentence "Audi ut discas" (Listen in order to learn) is essential in research. Consequently, all the research results presented in this document have been realized in collaboration with different persons, from the Kairos team, of course, but also from other national and international teams. I was really pleased to work with all of them and I hope I will be able to continue in this direction.

In order to present my work, I chose to address the challenges presented previously in order in the document. Chapter 2 presents the work on CCSL, its formal definition and its use as an explicit behavior model. Chapter 3 presents how we promote logical time for the formal specification of the concurrent and timed aspects of an operation semantics. Chapter 4 presents the rationale and definition of coordination patterns; from which coordination of heterogeneous models can be done, both for heterogeneous model execution and formal reasoning. Chapter 5 draws a short conclusion before, in chapter 6 detailing some perspectives on the continuation of my work.

Chapter 2

Modeling Concurrent and Timed Execution of Models

Contents

2.1 Introduction 9
2.2 Clock Constraint Specification Language 10
2.3 Comparing CCSL and temporal logics 17
2.4 TimeSquare: the CCSL tool 21
2.5 Using CCSL in different domains 27
2.6 Towards Multi-View Behavior Modeling. 37
2.7 Conclusion 46

2.1. Introduction

Nowadays, a program or a model is a purely syntactic artifact (typically a (set of) file(s)), which represents the structure of the program. This structure being in a specific format, it can be read by a user (*i.e.*, a human or a tool) that understand this syntax. In order to manipulate the structure, the user usually *knows* its behavioral semantics. For instance the C compiler *knows* the behavioral semantics of the C language and it can consequently compile the program into a corresponding set of CPU instructions. It is more accurate to state that the C compiler encodes a specific understanding of the behavioral semantics of the C language, defined in natural language in an ANSI standard. Indeed, using different C compilers targeting a same CPU can result in different sets of instructions due to different optimizations or different understanding of the semantics. In order for different users to have the same understanding of the model, it is require to agree on the way it behaves. This is usually not a real issue for popular general purpose language like C, since compiler writers put long term efforts in the writing of the compiler and the user community is large enough to detect bugs and deviance with regards to other compilers or to the standard. However, it became an issue with the emergence of many (interacting) Domain Specific Languages (DSLs), for which the access to the behavioral semantics is not necessarily provided and for which the user community and existing tools are restricted. Additionally, there exist many DSLs which are based on a similar syntax (*e.g.*, state machines) but with different behavioral semantics. For instance in [13], the authors have shown that an exact same model executed in two different tools results in two strongly different behavioral semantics. This is also a well known mechanism in UML where semantic variation points exist.

In the targeted domain (*i.e.*, CPS), applications are often interacting repeatedly with their external environment. The main design concern goes with handling data or control flow propagation, which frequently includes streaming and pipelined processings. In contrast, the actual data values and computation contents are of lesser importance (for design, not for correctness). Application models such as process networks, reactive components and state/activity diagrams are used to represent the structure

and the behavior of such applications. In order to complete the traditional structure of such models with an abstract specification of the way they behave, we promote the use of logical time as a unifying notion of time to capture both causal dependencies and physical time relationships. Logical time was first introduced by Lamport to represent the execution of distributed systems [113]. It has been extended and used in distributed systems to check the communication and causality path correctness [71]. At the same period, Logical time has also been used in synchronous languages [19, 23] for its polychronous and multiform nature, *i.e.*, based on several time references.

In the synchronous programming approach, the notion of logical clock has proved to be adaptable to any level of description, from very flexible causal time descriptions to very precise scheduling descriptions [32]. A simple and classical use of multiform logical time for specification is "*Task 1 executes twice as often as Task 2*". The instant at which $Task$ 2 executes is taken as a time reference to specify the execution of $Task$ 1. An event is then expressed relative to another one, that is used as a reference. No reference to physical time is given. Furthermore, if the (physical) duration between two successive executions of *Task1* is given, the instants at which *Task2* must execute can be deduced. Relying only on the logical specification at early stages avoids over-specifications induced by using physical time references while the execution platform is not fully determined, yet. Actually, physical time is a particular case of logical time where a physical clock is taken as a reference. By using such logical time, we were able to specify very different kind of behaviors (*e.g.*, both non deterministic and deterministic behaviors, both synchronous and asynchronous ones). By doing so, our goal was to associate an explicit formal semantics to the model structural elements so that it can be referred to as a golden model by external tools and make the model amenable to formal analysis, code generation or synthesis. An explicit and formal semantics within the model is fundamental to prevent different analysis tools from giving a different semantics to the same model. We proposed a language, called Clock Constraint Specification Language (CCSL), specifically devised to equip a given model with a logical time model specifying its behavior in an abstract way. CCSL is a formal domain specific language originally introduced in the MARTE profile [161] defined by the OMG[1]. CCSL have been formally defined, tooled and used to specify the behavior of various models, written in different languages.

In this chapter, we start with the definition of CCSL and specifically the major works I did on the definition/extension of the CCSL language. Then, expressiveness and tooling is discussed. Finally, we selected different models from different domains to illustrate possible uses of CCSL ([81, 83, 125, 145, 146]). These examples range from high level specification languages to the effect of allocating functions on a specific hardware platform.

2.2. Clock Constraint Specification Language

This section is an up to date excerpt of the following paper:

> Julien Deantoni, Charles André, and Régis Gascon. CCSL denotational semantics. Research Report RR-8628, , November 2014. URL `https://hal.inria.fr/hal-01082274`

CCSL is a language to impose dependence relationships between instants of different clocks. These dependences are specified by *Clock constraints*. The constrained elements are clocks. A clock constraint imposes relationships between instants of its constrained clocks. CCSL formal semantics was first defined in [6], however, this semantics was missing the crucial notion of death and birth of the clocks, introduced in section 2.2.3. Additionally, we extended the notion of constraints with the notion of death and birth, so that it opened the door to CCSL mode automata where a set of constraints can be activated/deactivated according to its belonging to a specific mode. In order to overview CCSL, we first introduce multi form logical time. Then, to understand clock constraints, we define relations on instants, clocks, life intervals and finally clock constraints.

[1] `http://omg.org`

2.2.1. Multiform logical time

CCSL was initially introduced in the MARTE Time model, which deals with both discrete and dense times. In MARTE, a *clock* gives access to a *time structure* made of *time bases*, which are themselves ordered sets of *instants*. A clock can be either *dense* or *discrete*. This section about CCSL focuses on the structural relations between clocks and these relations do not differentiate between dense and discrete clocks. However, some relations only apply to discrete clocks and others apply to both discrete and dense clocks.

Leslie Lamport [113] introduced logical clocks in the late 70's. The logical clocks associate numbers (logical timestamps) with events in a distributed system, such that there exists a consistent total ordering of all the events of the system. These clocks can be implemented by counters with no actual timing mechanisms. In the 80's, synchronous languages [18] introduced their own concept of logical time. This logical time shares with Lamport's time the fact that they need not actually refer to physical time. Logical time only relies on (partial or total) ordering of instants. In what follows, we consider logical time in the sense of synchronous languages. In the synchronous language Signal [19], a *signal* s is an infinite totally ordered sequence $(s_t)_{t\in\mathbb{N}}$ of typed elements. Index t denotes a *logical instant*. At each logical instant of *its* clock, a signal is present and carries a unique value. Signal is a multi-clock (or polychronous) language: it does not assume the existence of a *global clock*. Instead, it allows multiple logical clocks. Signal composition is ruled by operators which are either mono-clock operators (composing signals defined on a same clock) or multi-clock operators (allowing composition of signals having different clocks).

A logical clock can be associated with any event. This point of view has been adopted in the MARTE time model [161, Chap. 10]. A logical clock "ticks" with each new occurrence of its associated event. Synchronous languages like Esterel exploit this property. In an Esterel program, time may be counted in seconds, meters, laps... (see Berry's RUNNER program [24] which describes the training of a runner). This variety of events supporting time leads to the concept of *multiform time*. For instance, the electronic ignition is driven by the angular position of the crankshaft rather than by a chronometric time (see our studies of a knock controller in a 4-stroke engine [5, 145]).

In this section, we consider both dense and logical clocks and their relationships through *clock constraints*. Because the notion of instant and the ordering of such instants are the core of the proposition, we first introduce the relations we defined on the instants.

2.2.2. Instants and Time Structure

Instants are the basic elements which are manipulated all along this section. An instant can be seen as an event occurrence, occurring at a specific (logical) time. Expliciting the fact that an instant as a specific logical time implies that instants can be partially ordered. To do so, we use specific instant relations. An instant relation between two instants reflects a specific ordering between the considered instants. We defined five kinds of instant relations, derived from two main relations[2].

Causal and chronological relations over a set of instant are defined in a *time structure*. A time structure $\mathcal{T}$ is a triple $\langle \mathcal{I}, \prec, \equiv \rangle$ made of a set of instants $\mathcal{I}$ and two main relations on instants satisfying the conditions below.

- The *precedence relation* $\prec$ is a strict order relation on I (irreflexive, asymmetric and transitive)
- The *coincidence relation* $\equiv$ is an equivalence relation (reflexive, symmetric and transitive). It reflects the fact that two instants have the exactly same logical time.
- The precedence and coincidence relations satisfy the following properties: for every $i_1, i_2, i_3 \in I$
  - if $i_1 \prec i_2$ and $i_2 \equiv i_3$ then $i_1 \prec i_3$,
  - if $i_1 \prec i_2$ and $i_1 \equiv i_3$ then $i_3 \prec i_2$.

From these two relations, one can define additional relations associated to any time structure:

[2] there are different ways to express the kernel relations. We chose here the one used in the main technical report [6]

- The *causality relation* $\preccurlyeq$ is defined as the union of the precedence relation and the coincidence relation.

$$\forall i_1, i_2 \in I,\ i_1 \preccurlyeq i_2 \Leftrightarrow (i_1 \prec i_2 \text{ or } i_1 \equiv i_2).$$

  Note that $\preccurlyeq$ is not really a partial order. By definition, this relation is reflexive and transitive, but it is not anti-symmetric. If $i_1 \preccurlyeq i_2$ and $i_2 \preccurlyeq i_1$ we can deduce that $i_1 \equiv i_2$ but coincidence does not imply equality.

- The *exclusion relation* $\#$ is defined as the union of the precedence relation and its converse relation. Formally, we have

$$\forall i_1, i_2 \in I,\ i_1 \# i_2 \Leftrightarrow (i_1 \prec i_2 \text{ or } i_2 \prec i_1).$$

  This imposes that instants i_1 and i_2 must not be coincident.

- Finally, the independence relation $\|$ corresponds to the absence of other relations.

$$\forall i_1, i_2 \in I,\ i_1 \| i_2 \Leftrightarrow \text{ neither } (i_1 \prec i_2) \text{ nor } (i_2 \prec i_1) \text{ nor } (i_1 \equiv i_2).$$

  This relation can be used to express concurrency between two instants.

The graphical representations of these different instant relations, used later in the section, is given in Table 2.1.

instant relation	symbol	graphical representation
precedence	$\prec$	
causality	$\preccurlyeq$	
coincidence	$\equiv$	

Table 2.1: Instant relations

2.2.3. Clocks

This model is the one implemented in TimeSquare, the CCSL simulation tool. This model considers that a clock is an ordered set of instants. A clock has a lifetime delimited by a birth instant and a death instant.

Formally, a *Clock* c is a 5-tuple $\langle \mathscr{I}_c, \prec_c, c^{\uparrow}, c^{\downarrow}, \equiv_{c^{\downarrow}} \rangle$ where $\mathscr{I}_c$ is a possibly infinite set of instants, $c^{\uparrow} \notin \mathscr{I}_c$ is the birth instant of c, and $c^{\downarrow} \notin \mathscr{I}_c$ is the death instant of c, $\equiv_{c^{\downarrow}}$ is a coincidence relation and $\prec_c$ is an order relation on $\mathscr{I}_c \uplus \{c^{\uparrow}, c^{\downarrow}\}$ satisfying the following conditions:

- The restriction of $\prec_c$ on $\mathscr{I}_c$ is a strict order relation.
- All instants of $\mathscr{I}_c$ are strictly ordered ($\prec_c$ is total):

$$\forall i, j \in \mathscr{I}_c,\ (i \neq j) \Rightarrow (i \prec_c j) \text{ or } (j \prec_c i).$$

- The birth strictly precedes all the other instants of the clock:

$$\forall i \in \mathscr{I}_c \uplus \{c^{\downarrow}\},\ c^{\uparrow} \prec_c i.$$

- Every instant precedes the death but the last one that may coincide with it:

$$\forall i \in \mathscr{I}_c \uplus \{c^{\uparrow}\},\ ((i \prec_c c^{\downarrow}) \vee (i \equiv_{c^{\downarrow}} c^{\downarrow}))$$

The set of instants $\mathscr{I}_c$ represents the occurrences or *ticks* of the clock c. This is why birth and death do not belong to this set. A clock can have a finite or infinite number of instants. If $\mathscr{I}_c$ is infinite then the death instant is not necessary.

A *discrete-time clock* c is a clock with a discrete set of instants $\mathscr{I}_c$. In that case, $\mathscr{I}_c$ can be indexed by natural numbers in a fashion that respects the ordering $\prec_c$: we define $\text{idx}_c : \mathscr{I}_c \to \mathbb{N}^{\star}$ ($\mathbb{N}^{\star} = \mathbb{N} \setminus \{0\}$) such

that $\forall i \in \mathscr{I}_c$, $\mathrm{idx}(i) = k$ iff i is the kth instant in $\mathscr{I}_c$ wrt. $\prec_c$. By convention we will consider that the first instant of c is indexed by 1.

For any discrete time clock $c \triangleq \langle \mathscr{I}_c, \prec_c, c^{\uparrow}, c^{\downarrow}, \equiv_{c^{\downarrow}} \rangle$, $c[k]$ denotes the kth instant in $\mathscr{I}_c$ (*i.e.*, $\mathrm{idx}_c(c[k]) = k$). For any instant $i \in \mathscr{I}_c$ of a discrete time clock c, $^{\circ}i$ is the unique immediate predecessor of i in $\mathscr{I}_c \uplus \{c^{\uparrow}\}$. We assume that the predecessor of $c[1]$ is the birth $c^{\uparrow}$. Similarly we denote the unique immediate successor of i in $\mathscr{I}_c$ as i°, if any.

The number of instants preceding a specific instant i of a clock c is retrieved by $\chi(c)@i$. It only makes sense on a discrete clock and it always returns ∞ on a dense clock. Note that the use of this function needs i to be strictly ordered on the set of instants it refers (*i.e.*, $\mathscr{I}_c$ in the previous example) (cf. equation 2.1.0). From the previous definition we give the following lemmas:

$$
\begin{aligned}
&\langle \mathscr{I}, \prec, \equiv \rangle \models \chi(c) && \Leftrightarrow \\
&\quad 0)\ (i \in \mathscr{I}) \wedge (\mathscr{I}_c \subseteq \mathscr{I}) \wedge (\exists j, k \in \mathscr{I}_c \uplus \{c^{\uparrow}\}) \wedge (j \prec k) \wedge (j \prec i) \wedge (i \prec k) && \wedge \\
&\quad 1)\ \big(c \in C \wedge \chi(c)@i = k\big) \Leftrightarrow c[k] \preccurlyeq i \prec c[k+1] && \wedge \\
&\quad 2)\ \big(c \in C, \forall k \in [1; |\mathscr{I}_c|]\big) \Leftrightarrow \chi(c)@c[k] = k-1 && \wedge \\
&\quad 3)\ \chi(c)@c^{\uparrow} = 0 && \wedge \\
&\quad 4)\ \chi(c)@c^{\downarrow} = |\mathscr{I}_c|
\end{aligned}
\tag{2.1}
$$

2.2.4. Clock Constraints Specification

We[3] have introduced the concept of *clock constraints* in the MARTE specification (chapter 9) and also a dedicated language for expressing such constraints: CCSL [161, Annex C.3]. This language is non normative (the MARTE profile implementors are not obliged to support it). The semantics of CCSL given in the MARTE specification is informal. A first formal semantics, based on mathematical expressions has been proposed in a paper [123] and a research report [7], which is an extended version of [123]. A precise definition of the syntax of a *kernel* of CCSL along with a structural operational semantics was first defined in [6] and has been extended in [57] with the notion of birth and death as defined in this section. [57] is one possible operational semantics of the denotational semantics defined here. This semantics is the golden reference for the CCSL constraint solver implemented in TimeSquare [51][4], the software environment that supports CCSL.

Clock constraints are classified into two main categories:

1. clock relations that constrain the order of the instant of two or more existing clocks;
2. clock expressions that define a new clock from a set of parameters.

As for clocks, every relation and expression has a timelife defined by the instants of its *birth* and its *death*. We define in more details these elements in the following.

A CCSL specification $\mathscr{S}$ is a triple $\langle C, Expr, Rel \rangle$ such that C is a set of clocks, *Expr* is a set of *clock definitions* and *Rel* is a set of *clock relations*. Both *Expr* and *Rel* state constraints on the elements of C.

The models of CCSL specification are time structures. Let $\mathscr{S} = \langle C, Expr, Rel \rangle$ be a CCSL specification and $\mathscr{T} = \langle I, \prec, \equiv \rangle$ a time structure. Informally, $\mathscr{T}$ satisfies the specification $\mathscr{S}$ iff the following conditions holds:

- The set of instants I includes all the instants of the different clocks in C.
- The orders of the different clocks in C are preserved by $\prec$.
- The different constraints induced by the elements of *Expr* and *Rel* are satisfied.

The satisfaction of the different clock relations and definitions will be defined in the following sections and depends of the relations $\prec$ and $\equiv$.

[3] members of the former AOSTE team, which became Kairos
[4] `http:timesquare.inria.fr`

2.2.5. Definition of a specific subset of instants

Since relations and expressions are alive from their birth instant and possibly until their death instant, it is useful to define the set of instants of a clock between these two instants. An interval of clock c (*i.e.*, a subset of $\mathscr{I}_c$) characterized by two instants α and β is denoted as $\mathscr{I}_c^{\alpha..\beta}$. It is defined by the following equation:

$$\mathscr{I}_c^{\alpha..\beta} \triangleq \{i \in \mathscr{I}_c | (\alpha \preccurlyeq i) \wedge \neg(\beta \prec i)\} \tag{2.2}$$

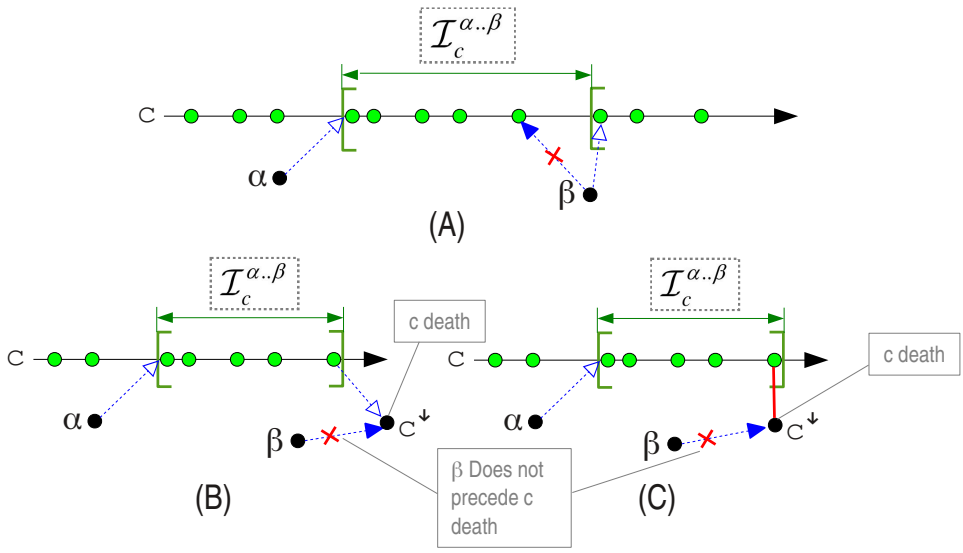

Figure 2.1: Clock interval definitions (taken from [56])

The interval starts with the first instant of $\mathscr{I}_c$ such that α precedes this instant. The interval ends at the first instant of $\mathscr{I}_c$ which is preceded by β; this instant *is not in* the interval. The latter constraint is expressed by $\neg(\beta \prec i)$ for all instants of $\mathscr{I}_c$ in the interval. Note that since $\preccurlyeq$ is a partial order, $\neg(\beta \prec i)$ is not equivalent to $(i \preccurlyeq \beta)$.

When β is not given, the interval (denoted as $\mathscr{I}_a^{\alpha..}$) is defined as follows:

$$\mathscr{I}_c^{\alpha..} \triangleq \{i \in \mathscr{I}_c | (\alpha \preccurlyeq i)\} \tag{2.3}$$

Fig. 2.2 illustrates various cases: without the clock death (Fig. 2.1.A) or with the clock death (Fig. 2.1.B and C).

These intervals are used in the following to represent the instants to be considered during the life of a specific relation or expression. Consequently, the set of instants of a clock c during the life of a relation r is noted $\mathscr{I}_c^r$. Similarly the set of instants of a clock c during the life of an expression is noted $\mathscr{I}_c^{ce}$ where ce is the clock defined by the expression. More generally, we denote $\mathscr{I}_c^x$ the set of instants of c that is defined on an interval bounded by $x^\uparrow$ and, either $c^\downarrow$ or $x^\downarrow$. More formally, $\mathscr{I}_c^x$ is defined by the following equation:

$$\mathscr{I}_c^x \triangleq \{(\exists x^\downarrow \in x \Rightarrow \mathscr{I}_c^x = \mathscr{I}_c^{x^\uparrow..x^\downarrow}) \vee (\nexists x^\downarrow \in x \Rightarrow \mathscr{I}_c^x = \mathscr{I}_c^{x^\uparrow..})\} \tag{2.4}$$

Predicate dies_in Let dies_in be a predicate on $clock \times constraint$ such that

$$c \text{ dies_in } (r) \Leftrightarrow (r^\uparrow \preccurlyeq c^\downarrow) \wedge \big((\exists r^\downarrow \in r \wedge \exists c^\downarrow \in c) \Rightarrow \neg(r^\downarrow \prec c^\downarrow)\big) \tag{2.5}$$

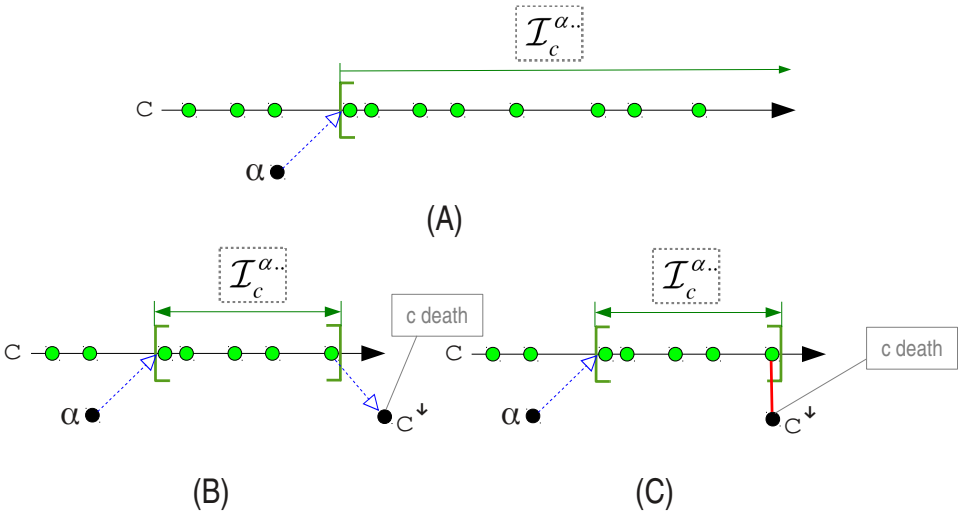

Figure 2.2: Clock interval when only α is given (taken from [56])

2.2.6. Clock relations

Specifying a full time structure using only instant relations is not realistic. Moreover a set of instants is usually infinite, thus forbidding an enumerative specification of instant relations. Hence the idea to extend relations to clocks. CCSL kernel defines five basic clock relations. In the following definitions, a and b stand for Clocks. For the sake of conciseness, mathematical definition is given only in the SubClock relation. Other definitions can be found in [56].

For each of them the definition is split into a first part named '(a)' defining the semantics of the relation and a second part '(b)' specifying the relations between deaths of clocks in the relations.

Subclocking: $a \overset{r}{\boxed{\subset}} b$ is a synchronous clock relation. a is said to be a sub-clock of b, and b a super-clock of a. The r represents the entity that gives the birth and the death of the relation (*i.e.*, the interval on which the relation applies). If r is not specified, the relation applies for the whole system life.

$$\begin{aligned} &\langle \mathscr{I}, \prec, \equiv \rangle \models a \overset{r}{\boxed{\subset}} b \Leftrightarrow \\ &(a) \quad \forall i_a \in \mathscr{I}_a^r, \exists i_b \in \mathscr{I}_b^r,\ i_a \equiv i_b \quad \wedge \\ &(b) \quad b \text{ dies_in } (r) \Rightarrow (a^{\downarrow} \preccurlyeq b^{\downarrow}). \end{aligned} \tag{2.6}$$

Equation 2.6-a defines the constraints between instants of $\mathscr{I}_a$ and $\mathscr{I}_b$. It means that each instant of a must coincide with an instant of b. Equation 2.6-b states that a must die when the superclock dies .

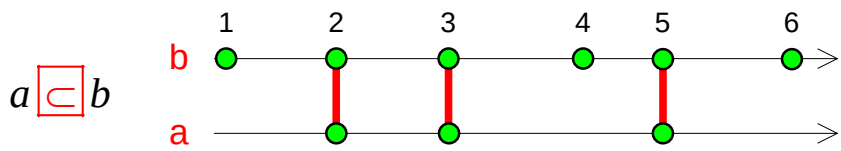

Figure 2.3: Example of subclocking.
(For a sake of simplicity $\chi(a)@r^{\uparrow} = 0$ and $\chi(b)@r^{\uparrow} = 0$)

Note that the death of a coincides with the one of b if b dies in r and a is not killed by another constraint in the time structure. Because of the possibly to be killed by another constraint, the death of a precedes the death of b.

All relations are defined to support both dense and discrete clocks. In this habilitation, for the sake of simplicity, the other relations are only informally defined and their definition works only for discrete clocks:

- *Equality:* $a \boxed{=} b$ is a special case of subclocking where the coincidence mapping is a bijection. $\forall k \in \mathbb{N}^\star, a[k] \equiv b[k]$. a and b are "synchronous". Clock deaths are also synchronous.
- *Precedence:* $a \boxed{\preccurlyeq} b$ means $\forall k \in \mathbb{N}^\star, a[k] \preccurlyeq b[k]$. a is said to cause b. b dies when a died and b cannot tick anymore.
- *Strict precedence:* $a \boxed{\prec} b$ is similar to the previous one but considering the strict precedence instead. $\forall k \in \mathbb{N}^\star, a[k] \prec b[k]$. b dies if a died and bb can not tick anymore.
- *Exclusion:* $a \boxed{\#} b$ means that a and b have no coincident instants. Death instants are also independent.

2.2.7. Clock expressions

They allow definitions of new clocks from existing ones. While formally defined in [56], we only present informally the clock expressions here.

There are classical clock expressions like the union and intersection operators respectively denoted by $a + b$ and $a * b$, the strict sample operators $a \Downarrow b$ or the preemption $a \lightning b$. A clock expression defines an implicit clock. The instants of the clock corresponding to the union $a + b$ coincide with an instant of a, of b, or with both and every instant of a and b coincides with an instant of $a + b$. The intersection $a * b$ has a set of instants that can be mapped to the instants of a that coincides with some instant of b. The sample expression $a \Downarrow b$ represents the subclock of b that ticks if a has occurred at least once since the last tick of b. The preemption $a \lightning b$ coincides with all the instants of a until the first occurrence of b and then the implicit clock dies. The informal semantics of the expressions above is illustrated in Fig. 2.4.

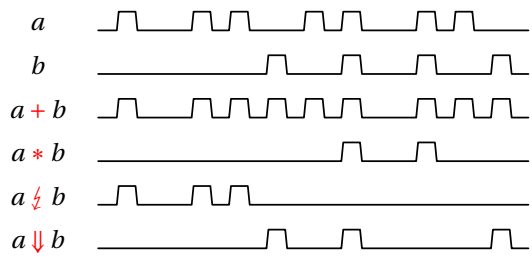

Figure 2.4: Illustration of some CCSL expressions

Here is a quick informal description of other existing CCSL expressions:

Awaiting a^n, where $n \in \mathbb{N}^\star$, is a synchronous clock expression. This expression waits for the nth strictly future tick of a. On this occurrence, the implicit clock ticks and dies. As one has to count the occurrences of a, the clock must be discrete.

Sup $a \vee b$ defines a clock that is the fastest among all the clocks slower than a and b. In other terms, the resulting clock always coincides with the slowest clock among a and b, in an opportunistic manner. Death of the implicit clock is simultaneous with the death of the clock it follows.

Inf $a \wedge b$ defines a clock that is the slowest among all the clocks faster than a and b. The resulting clock always coincides with the fastest clock among a and b, in an opportunistic manner. Death of the implicit clock is simultaneous with the death of the clock it follows.

Defer $a \overset{(w)}{\rightsquigarrow} b$ is a mixed clock expression that deals with multiple future scheduled ticks. w is an integer word defined, conjointly with notations over w,in [56]. Basically w is an integer word (*i.e.*, a finite or infinite word: $w \in \mathbb{N} \cup \mathbb{N}^\omega$). Defer is a complex expression when used with complex integer word. A simplistic explanation is to consider that each tick of the clock a specifies that a tick of the resulting clock will coincide after n ticks of b; where n is taken in w. if $w = 3^\omega$ and $a = b$ then the resulting clock coincides with clock a except for the 3 first ticks of a. When w is a integer repeated infinitely we often refer to this constraint as $delayedFor$. For the $Defer$ expression to make sense, a and b must be discrete.

Concatenation $a \bullet b$ defines a clock that coincides with a until its death. Once a is dead, then the resulting clock coincides with b. When b dies, then the implicit clock dies.

2.2.8. Temporal evolutions

A CCSL specification imposes a complex ordering on instants. We do not explicitly represent this time structure. We compute possible *runs* instead. A run is a sequence of steps. Each step is a set of clocks that are simultaneously fired without violating any clock constraints. When a discrete clock ticks (or fires), the index of its current instant is incremented by 1. The computation of a step is detailed in a technical report [6] that provides a syntax and an operational semantics for a kernel of CCSL. Here, we just sketch this process. An example of such run is represented graphically on Figure 2.9.

Using the semantics of CCSL, from a clock constraint specification $\mathscr{S}$ we derive a logical representation of the constraints $[\![\mathscr{S}]\!]$. This representation is a Boolean expression on a set of Boolean variables $\mathscr{V}$, in bijection b with the set of clocks $\mathscr{C}$. Any valuation $v : \mathscr{V} \to \{0,1\}$ such that $[\![\mathscr{S}]\!] = (v) = 1$ indirectly represents a set of clocks F that respects all the clock constraints: $F = \{c \in \mathscr{C} \mid v(b(c)) = 1\}$. F is a possible set of simultaneously fireable clocks. Most of the time, this solution is not unique. Our solver supports several policies for choosing one solution.

2.2.9. CCSL libraries

CCSL specifications are executable specifications. However, the expressiveness of the kernel CCSL is limited, for instance by the lack of support for parameterized constructs. The full CCSL overcomes these limitations through libraries. A library is a collection of parameterized constraints, using constraints from one or many other libraries. The primitive constraints, which constitute the kernel CCSL, are grouped together in the *kernel library*. The operational semantics is explicitly defined only for the constraints of the kernel library. Each user-defined constraint is structurally expanded into kernel constraints, thus defining its operational semantics.

As a very simple example, we define the *alternates* relation, denoted $\boxed{\sim}$. The user-defined relation consequently has two clock variables (v1, v2). This definition contains two instances of the precedes relation one instance of the defer expression whose definitions are given in the kernel library. The textual specification of the alternates relation is the following:

$$\begin{aligned}\text{def}\ \boxed{\sim}\ (\text{clock } v1, \text{clock } v2) \triangleq\ & (v1 \prec v2)\\ & \mid (v1DeferByOne \triangleq v1 \overset{(1^{\omega})}{\rightsquigarrow} v1)\\ & \mid (v2 \prec v1DeferByOne)\end{aligned}$$

We can also define in libraries user defined expressions. For instance, one interesting user defined expression is $filteredBy$ (∇). $a\ filteredBy\ w$ where w is an integer word is a recursive expression defined by:

$$\text{def}\ \nabla\ \text{clock } c1, \text{integerWord } w1) \triangleq (c1\hat{}head(w1)) \bullet (c1\ \nabla\ queue(w))$$

where $head(w)$ returns the first element of w and $queue(w)$ returns w minus its first element.

CCSL libraries are used later (Section 2.5) to group domain specific constraints. Section 2.5 also provides examples of behavior model specification realized in order to obtain a model made up with both an explicit structure and an explicit behavior. The next section focuses on the expressiveness of CCSL with regards to temporal logics.

2.3. Comparing CCSL and temporal logics

This section is an up to date excerpt of the following paper:

Régis Gascon, Frédéric Mallet, and Julien DeAntoni. Logical time and temporal logics: Comparing UML MARTE/CCSL and PSL. In Carlo Combi, Martin Leucker, and Frank Wolter, editors, *Eighteenth International Symposium on Temporal Representation and Reasoning, TIME 2011, Lübeck , Germany, September 12-14, 2011*, pages 141–148, Lübeck , Germany, September 2011. IEEE. ISBN 978-1-4577-1242-5. doi: 10.1109/TIME.2011.10. URL `https://doi.org/10.1109/TIME.2011.10`

CCSL is intended to be used at various modeling levels following a refinement strategy. It should allow both coarse, possibly non-deterministic, infinite, unbounded specifications at the system level but also more precise specifications from which code generation, schedulability and formal analysis are possible.

In the domain of hardware electronic systems, which is one of the subdomains targeted by CCSL, formal verification is an important step of the development. To allow simulation and formal verification of such systems, the IEEE Property Specification Language (PSL [150]) provides a formal notation for the specification of electronic system behavior, compatible with multiple electronic system design languages (VHDL, Verilog, SystemC, SystemVerilog).

A question arises: Is CCSL expressive enough to express properties usually modeled in PSL?

The main contribution of this section is the comparison of PSL and CCSL expressiveness. The first result is that neither of these languages subsume the other one. Consequently, we identified the CCSL constructs that cannot be expressed in temporal logics and propose restrictions to these operators so that they become tractable in temporal logics. Conversely, we also identified the class of PSL formulas that can be encoded in CCSL. Using this information, we show that translations between large fragments of CCSL and PSL can be defined. Because direct modular translation is more tedious, we use an automaton-based approach. Though translation from PSL to automata is a well studied topic (see e.g., [35]), similar transformation for CCSL specifications was a new and interesting result. However it is not presented in this section and the reader must refer to [77] if curious.

The rest of this section is organized as follows. In Sect. 2.3.1 we introduce PSL and determine which kind of properties cannot be expressed in each language. Then, we provide concluding remarks and future work.

2.3.1. Property Specification Language

The IEEE standard PSL [150] has been designed to provide an interface to hardware formal verification. Its temporal layer is a textual language to build temporal logic expressions. PSL assertions can then be validated by model-checking or equivalence checking techniques. The underlying linear-time logic in PSL extends LTL with regular expressions and sugaring constructs. However PSL remains as expressive as ω-regular languages. As it would be tedious to consider the sugaring operators of PSL in formal reasoning, we use the minimal core language defined in [35].

Let VAR be a set of propositions (Boolean variables) that aims at representing signals of the system. PSL atomic formulas are called *Sequential Extended Regular Expressions* (SERE). SEREs are basically regular expressions built over the Boolean algebra:

$$b ::= x \mid \overline{x} \mid b \wedge b \mid b \vee b$$

where $x \in \mathrm{VAR}$ is a Boolean variable. We also consider the standard implication and equivalence operators $\Rightarrow$ and $\Leftrightarrow$ that can be defined from the grammar above[5]. The set of SEREs is defined by:

$$r ::= b \mid r \cdot r \mid r \cup r \mid r^*$$

where b is a Boolean formula. The operators have their usual meaning: $r_1 \cdot r_2$ is the concatenation, $r_1 \cup r_2$ the union and r^* is the Kleene star operator. From these regular expressions, PSL linear properties[6] are defined by:

$$\phi ::= r \mid \phi \wedge \phi \mid \neg\phi \mid \mathsf{X}\phi \mid \phi \mathsf{U} \phi \mid r \rightarrowtail \phi.$$

[5] $x \Rightarrow y$ is equivalent to $\overline{x} \vee y$ and $x \Leftrightarrow y$ to $(x \Rightarrow y) \wedge (y \Rightarrow x)$.

[6] The PSL standard also defines a branching time part that we do not consider here.

where r is a SERE. The operators X (next) and U (until) are the classical temporal logic operators. We also use the classical abbreviations $\mathsf{F}\phi \equiv \top\mathsf{U}\phi$ (eventually) and $\mathsf{G}\phi \equiv \neg\mathsf{F}\neg\phi$ (always). The formula $r \rightarrowtail \phi$ is a "suffix conjunction" operator meaning that there must exist a finite prefix satisfying r and that ϕ must be satisfied at the position corresponding to the end of this prefix (with a one-letter overlap between the prefix and the suffix).

The semantics of PSL is defined in such a way that properties can be interpreted over infinite words as well as finite or truncated words. This is important for applications like simulation or bounded model-checking. Similarly to CCSL, the models of PSL are finite or infinite sequences over elements of 2^{VAR} that represent the set of variables that holds at each position.

For every $X \in 2^{\mathrm{VAR}}$ and $p \in \mathrm{VAR}$, we note $X \models_b p$ iff $p \in X$ and $X \models_b \overline{p}$ iff $p \notin X$. The remaining of the Boolean satisfaction relation $\models_b$ is standard. SEREs refer to a finite (possibly empty) prefix of the model. So σ is supposed to be finite in a SERE satisfaction relation (which is not the case in a PSL satisfaction relation). The SERE satisfaction is defined by induction as follows:

- $\sigma \models_{re} b$ iff $|\sigma| = 1$ and $\sigma(0) \models_b b$,
- $\sigma \models_{re} r_1 \cdot r_2$ iff there are σ_1, σ_2 s.t. $\sigma = \sigma_1\sigma_2$ and $\sigma_1 \models_{re} r_1$ and $\sigma_2 \models_{re} r_2$.
- $\sigma \models_{re} r_1 \cup r_2$ iff $\sigma \models_{re} r_1$ and $\sigma \models_{re} r_2$.
- $\sigma \models_{re} r^*$ iff either $\sigma = \epsilon$ or there exist $\sigma_1 \neq \epsilon$ and σ_2 s.t. $\sigma = \sigma_1\sigma_2$, $\sigma_1 \models_{re} r$ and $\sigma_2 \models_{re} r^*$.

Finally, the satisfaction of PSL properties is defined by:

- $\sigma \models_{psl} \neg\phi$ iff $\sigma \not\models_{psl} \phi$,
- $\sigma \models_{psl} \phi_1 \wedge \phi_2$ iff $\sigma \models_{psl} \phi_1$ and $\sigma \models_{psl} \phi_2$,
- $\sigma \models_{psl} \mathsf{X}\phi$ iff $|\sigma| > 1$ and $\sigma^1 \models_{psl} \phi$,
- $\sigma \models_{psl} \phi_1\mathsf{U}\phi_2$ iff there is $0 \leq i < |\sigma|$ s.t. $\sigma^i \models_{psl} \phi_2$ and for every $0 \leq j < i$ we have $\sigma^j \models_{psl} \phi_1$,
- $\sigma \models_{psl} r \rightarrowtail \phi$ iff there is a finite prefix $\sigma_1\alpha$ of σ ($\alpha \in 2^{\mathrm{VAR}}$ is a single letter) s.t. $\sigma = \sigma_1\alpha\sigma_2$, $\sigma_1\alpha \models_{re} r$ and $\alpha\sigma_2 \models_{psl} \phi$,
- $\sigma \models_{psl} r$ iff for every finite prefix σ_1 of σ there is a finite word σ_2 s.t. $\sigma_1\sigma_2 \models_{re} r \rightarrowtail \top$.

2.3.2. Comparing PSL and CCSL

The CCSL semantics we consider in this section is restricted. In the general definition, models of CCSL specifications do not need to be totally ordered. However, under this restriction CCSL and PSL share common models. So we can compare the classes of properties they can express.

Let S be a CCSL specification over a set of variables $V_S \subseteq \mathrm{VAR}$ and ϕ a PSL formula over a set of variables $V_\phi \subseteq \mathrm{VAR}$. We will say that S is encoded by (or simulated by) ϕ s.t. $V_S \subseteq V_\phi$ iff every model of ϕ is also a model of S and every model of S can be extended on V_ϕ to a model of ϕ. The converse simulation relation is a bit different. CCSL models have the properties that one can add an unbounded amount of empty states between two relevant states and left the satisfaction unchanged. This can easily be proved by induction on the structure of a CCSL specification.

Lemma 1 *Let S be a CCSL specification. For every model σ satisfying S and every $0 \leq i \leq |\sigma|$ the model σ' defined by*

$$\begin{aligned}
&\sigma'(j) = \sigma(j) \textit{ for every } j < i\\
&\sigma'(i) = \emptyset\\
&\sigma'(j) = \sigma(j-1) \textit{ for every } i < j \leq |\sigma| + 1
\end{aligned}$$

also satisfies S.

This property is a consequence of the multiclock aspect of CCSL. It is not possible to completely link the execution of a CCSL specification to a global clock. However, the states where no clocks occur are *irrelevant* from the CCSL point of view as they do not make the system evolve. So it is not really a

problem to discard them. Actually, this is what is done in the CCSL simulator called TimeSquare.We will say that ϕ is simulated by S s.t. $V_\phi \subseteq V_S$ iff every model of S *with no irrelevant states* is also a model of ϕ and every model of ϕ can be extended to a model of S.

Some CCSL relations or expressions implicitly introduce unbounded counters. For instance, one has to store the number of occurrences of the clocks c_1 and c_2 (or the difference between them) to encode the precedence relation $c_1 \prec c_2$. The corresponding language is made of all the words such that every finite prefix contains more occurrences of c_1 than c_2. Such a language is neither regular nor ω-regular and cannot be encoded in PSL. The same remark holds for the expressions $c_1 \wedge c_2$ and $c_1 \vee c_2$. On the other hand, CCSL relations and expressions only state safety constraints. As a specification is a conjunction of such constraints the result is always a safety property. CCSL cannot express liveness like the reachability property $\mathsf{F}p$. For finite executions, there is also no way to express that the model must have a next position which can be stated by $\mathsf{X}\top$ in PSL. To summarize, the preliminary comparison of expressiveness of CCSL and PSL gives the following results:

(I) There are PSL formulas that cannot be encoded in CCSL.

(II) There are CCSL specifications that cannot be encoded in PSL.

It is now clear that PSL and CCSL are not comparable in their whole definition. We have shown in [77] that restricting the properties of each language according to the observations above is enough to obtain large fragments that can express the same set of properties. To prove that, we defined an automatic way to encode a property in each of these fragments into the other. Our method was based on intermediate translation into automata.

2.3.3. Sum-up about CCSL and PSL

In this section, we over-viewed how we compared the formal languages CCSL and PSL. Both languages can be used to specify behavioral properties in the domain of hardware electronic systems but at different design levels. These results contribute to clarify their role when addressing this domain by comparing the type of properties that each language can express.

We have first identified the CCSL constructs that cannot be expressed in PSL and the class of PSL formulas that cannot be stated in CCSL. Considering the results of these observations, we have defined fragments of CCSL and PSL that can be encoded into each other. A sufficient condition to translate CCSL specifications into PSL is to bound the integer counters used to count the number of occurrences of clocks. More precisely, the relative advance of the clocks put in relation by these CCSL constructs must be bounded. Conversely, CCSL cannot express the class of liveness properties but can express every PSL safety property.

Note that CCSL cannot express liveness because it has not been designed for this purpose. However, it could be interesting to capture all the expressive power of PSL in a higher-level description language. A first step to fill the gap could be to identify what kind of temporal logic can also take care of multi-clock and synchronous aspects.

2.4. TimeSquare: the CCSL tool

This section is an up to date excerpt of the following papers:

> Julien DeAntoni and Frédéric Mallet. Timesquare: Treat your models with logical time. In Carlo A. Furia and Sebastian Nanz, editors, *Objects, Models, Components, Patterns - 50th International Conference, TOOLS 2012, Prague, Czech Republic, May 29-31, 2012. Proceedings*, volume 7304 of *Lecture Notes in Computer Science*, pages 34–41. Springer, 2012. ISBN 978-3-642-30560-3. doi: 10.1007/978-3-642-30561-0_4. URL `https://doi.org/10.1007/978-3-642-30561-0_4`

> Kelly Garcés, Julien Deantoni, and Frédéric Mallet. A Model-Based Approach for Reconciliation of Polychronous Execution Traces. In *SEAA 2011 - 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, Oulu, Finland, August 2011. IEEE. URL `https://hal.inria.fr/inria-00597981`

> Julien Deantoni, Frédéric Mallet, Frédéric Thomas, Gonzague Reydet, Jean-Philippe Babau, Chokri Mraidha, Ludovic Gauthier, Laurent Rioux, and Nicolas Sordon. RT-simex: retro-analysis of execution traces. In Kevin J. Sullivan Gruia-Catalin Roman, editor, *SIGSOFT FSE*, volume ISBN 978-1-60558-791-2, pages 377–378, Santa Fe, États-Unis, 2010. doi: 10.1145/1882291.1882357

> Julien Deantoni, Frédéric Mallet, Charles André, and Frédéric Thomas. Logical time @ work: the RT-Simex project. In *Sophia Antipolis Formal Approach*, Sophia, France, April 2011. URL `https://hal.inria.fr/inria-00587151`

In this section, we overview the main functionality of TimeSquare. TimeSquare is developed for 10 years in collaboration with temporary engineers like Nicolas Chleq or Benoit Ferrero as well as with many students. I'm the main maintainer of TimeSquare and I supervised most of the students who worked on it. TimeSquare is based on eclipse and integrated in a model driven approach (EMF) so that, by using models, it allows the specification of model behavior with formal annotation. CCSL concrete syntax is based on Xtext[7] so that the user directly constructs an EMF model while typing. This model can be parsed easily to provide useful information like the clock tree graph that represents the polychronous specification in a graphical way [173]. Keeping the specification as a model enables a better integration with a model driven approach because the model, the formal language and the solver are in the same technological space. The main benefits is the ability to link specification as well as results directly to the models. The feedback to the user is then greatly improved compared with transformational techniques, which translate a model to an existing formal language. The output of TimeSquare is also an EMF model that defines a specific partial order of events, which represents the trace of the simulation. It is important to notice that a single simulation provides a partial order and consequently captures several possible executions (total orders). It is possible to subscribe to specific events during the construction of this trace by using the extension point mechanism provided by eclipse. By using such an approach it is possible to write user- or domain-specific back-ends.

The architecture of TimeSquare is shown in Figure 2.5. Straight arrows indicate the model flows, whereas dashed arrows represent the links between two models. The trace model is directly linked to CCSL model elements, which in turns are linked to other (EMF) model elements.

The section organization follows this architecture. Subsection 2.4.1 describes the CCSL concrete syntax tooling, Subsection 2.4.2 explains how the solver produces the trace model according to possibly user-defined simulation policies. Subsection 2.4.3 details the back-end mechanism and overviews some of the major existing back-ends. Finally, subsection 2.4.4 overviews how we aligned execution traces with simulation traces.

[7] `http://www.eclipse.org/Xtext/`

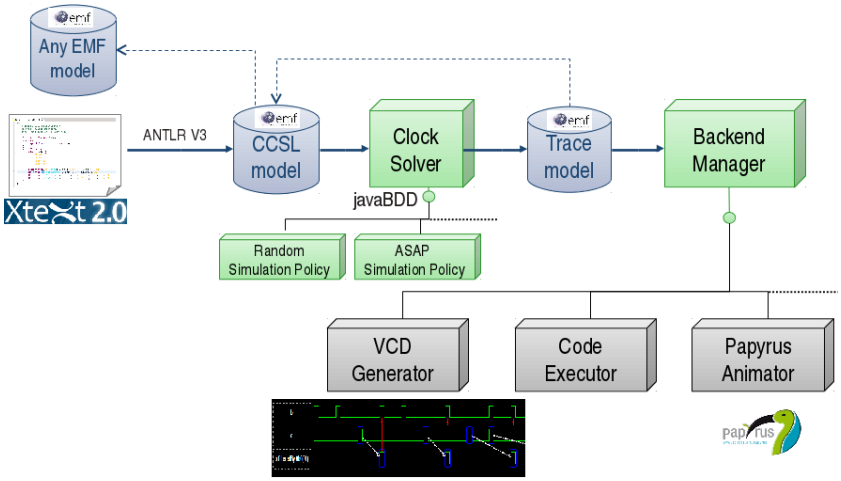

Figure 2.5: Big Picture of the TimeSquare Architecture (taken from [51])

2.4.1. CCSL Implementation

The clock constraint specification language (CCSL) complements structural models by defining formally a set of kernel clock constraints, which apply infinitely many instant relations. The operational semantics of CCSL constraints was initially defined in a technical report [6] and completed recently in [57]. Some recurrent constraints from a specific domain can be complex. To ease the application of such complex constraints, libraries of user-defined constraints can be built by composing existing constraints. This language and the library mechanism is defined in a metamodel accessible here: `http://timesquare.inria.fr/resources/metamodel`. This metamodel can be instantiated from two different classes depending on whether the user wants to create a CCSL specification or a library. Because using the ecore reflective editor provided by EMF is not suitable for any user, we created a textual concrete syntax using XText. XText allows us to easily parse the text file in order to create the corresponding EMF model and also to make link to external existing models. Direct links between the CCSL specification and the (structural) EMF model are important to help the user in the specification of constraints and the creation of a coherent specification (completion, detection of errors on the fly, tips, etc). Two kinds of model can be imported in a CCSL specification: external libraries and EMF-based models. If a library is imported, the Xtext editor will automatically propose, as a completion mechanism, the relations and the expressions from the library. It will also check the parameters provided and propose some changes if a problem is detected. Such customization of the editor greatly eases the users in their formal specification. In order to allow static analysis as, for instance the one described in [173], TimeSquare provides a clock graph that depicts the synchronous/asynchronous relations between clocks. A simple CCSL specification, the corresponding and synchronized EMF model in the outline and the associated clock graph are represented on Figure 2.6.

If an EMF model is imported in a CCSL specification, all the elements from the model that own a "name" property will be accessible and possibly constrained. Figure 2.7 shows a part of the previous CCSL specification where an import from a UML model is done. It allows enriching the *Clock* declaration with the structural element from the UML model (here subject to completion). The meaning of the link can also be specified (*send* on the figure).

2.4.2. Simulation

The formal operational semantics of CCSL constraints makes CCSL specifications executable. A *run* of a time system is an infinite sequence of *steps* (if no deadlocks are found by the solver). During a step, a Boolean decision diagram represents the acceptable sets of clocks that can tick. If the CCSL specification contains assertion(s), then the Boolean decision diagram also represents the state of the assertion (violated or not). Assertions never change the set of clocks that can tick. It has been used in the RT-Simex project in order to check if a specific execution trace is correct with regard to a CCSL

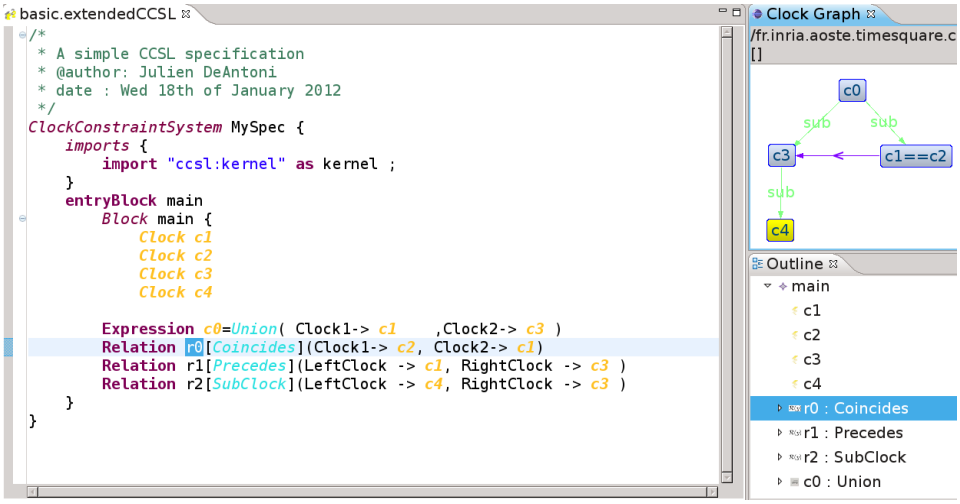

Figure 2.6: A simple CCSL specification and the associated clock graph (taken from [51])

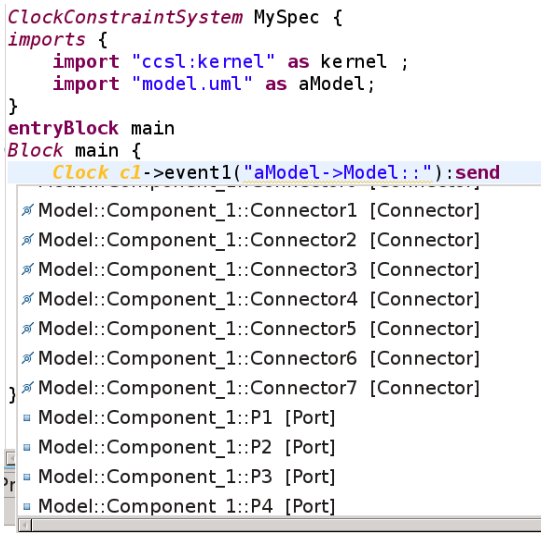

Figure 2.7: Link between a model (UML here) and a CCSL specification, helped by completion (taken from [51])

specification [54]. If the CCSL specification is deterministic, there exists a single set; if not, a simulation policy is used to choose amongst the possible solutions. TimeSquare offers a set of simulation policies (Random, As soon as possible, etc). It is possible for a user to add a new simulation policy by using a specific TimeSquare extension point. The choice of the simulation policy, the number of steps to compute as well as the choices about debugging information are integrated in the existing eclipse configuration mechanism so that a run or a debug (step by step) of a CCSL specification is accessible as in other languages like java by right clicking on the file and *run as...* Figure 2.8 shows the configuration box of a CCSL specification. The reader can notice some back-ends on the bottom of the screenshot. These back-ends are overviewed in the next section.

2.4.3. Back-ends

TimeSquare can be used in various model-driven approaches. Depending on the domain, users are interested in different feedback or analysis of the results. To allow an easy integration of TimeSquare in various domains, we implemented a back-end manager, which enables the easy addition of user-defined back-ends. The back-end manager receives the status of the clock (it ticks or not) at each simulation step. It also receives the status of relations (causalities and coincidences) as well as the status of

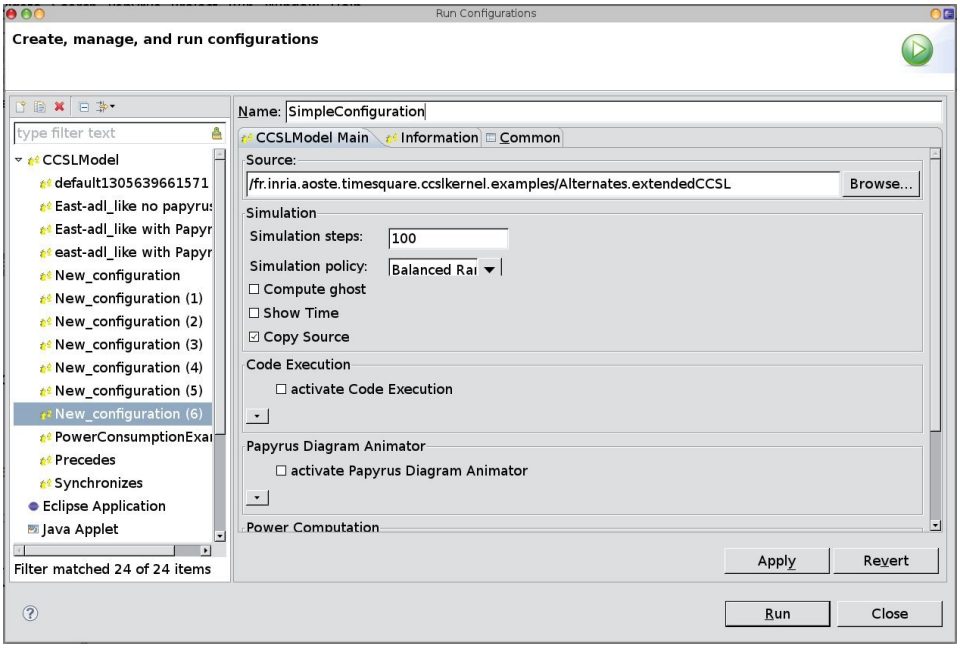

Figure 2.8: The configuration box of a CCSL specification (taken from [51])

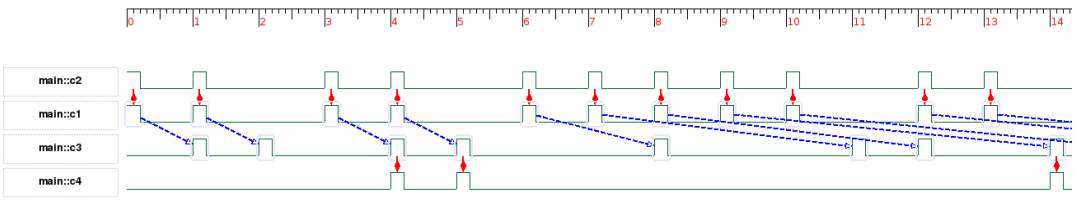

Figure 2.9: The extended VCD diagram back-end

the assertions (violated or not). By using a specific extension point, a developer can create a back-end that subscribes to some of these events. The registered back-ends are then noticed when the events they subscribed to occur during the simulation step. It has been used in very different ways but we present in this paper only the three main ones: the VCD diagram creator, the papyrus animator and the code executor.

2.4.3.1. VCD diagram creator:

VCD is an existing standard mainly used in the electronic design domain. It is very close to the UML timing diagram and represents the evolution of events (Clocks) *vs.* time evolution, represented horizontally. It classically represents a total order of events. Because TimeSquare provides a trace which is only partially ordered, the classical VCD features have been extended in order to graphically represent such partial order. On Figure 2.9, a simple VCD is represented. It results from the simulation of the CCSL specification represented on Figure 2.6 where the `c0` clock is hidden to simplify the reading. We can notice the optional presence of two kinds of link between the ticks of the clocks: blue arrows, which represents causalities (loosely synchronizations) and red links, which represents coincidences (strong synchronizations). The result is that the partial order is valid as long as the red links are not broken and the blue arrows never go back in time.

2.4.3.2. Papyrus diagram animator:

When a CCSL specification is linked to a UML model, the model is often represented graphically in a UML tool. Papyrus (`http://www.eclipse.com/Papyrus`) is an open source UML tool integrated with

eclipse EMF and GMF. The papyrus animator provides a graphical animation of the UML diagrams during the simulation. The kind of graphical animation depends on the "meaning" of the event linked to the UML model (send, reveive, start, etc). This animation provides a very convenient feedback to the user who wants to understand what happens in the model according to the constraints he wrote. Additionally to graphical animation, the Papyrus animator adds *comments* to the UML model elements that represent their activation trace, keeping this way a trace of the simulation directly in the UML model. The Papyrus animator is shown conjointly (and synchronized with) the VCD diagram on Figure 2.10.

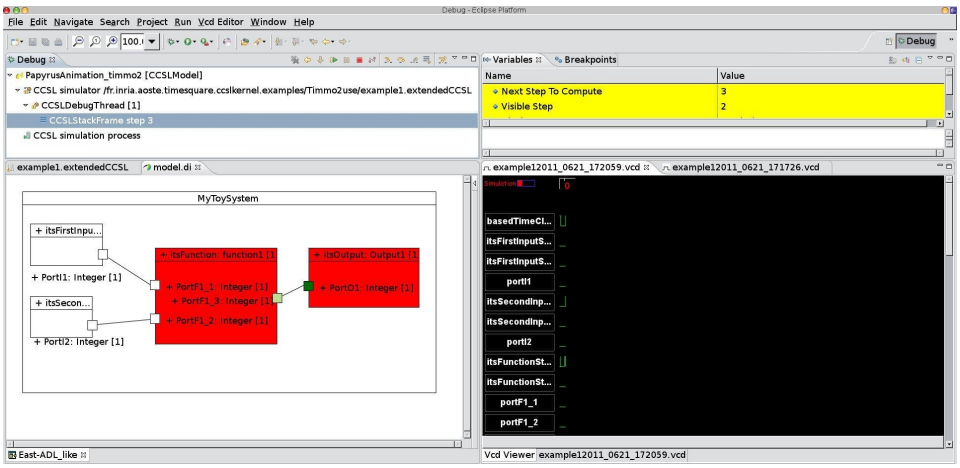

Figure 2.10: The animation of a UML model and the associated timing diagram in TimeSquare (taken from [51])

2.4.3.3. Code executor:

When a software is prototyped, it can be convenient to run some piece of code in order to provide application specific feedback. For instance we developed a simple digital filter by using UML composite structure in Papyrus and we added constraints on it representing its synchronizations (so that the diagram can be animated conjointly with the VCD diagram). To test our algorithm and ease the debugging of the synchronization in the model, we used the JAVA code executor. It allows the declaration of objects and the launch of specific methods of these objects when a desired event occurs (tick of a clock, etc). It can be used, as in the digital filter, to represent the data manipulation of the filter and to graphically represent the internal state of the memory. It can also be used to pop-up information windows when an assertion is violated, etc.

2.4.4. Analysis of existing traces

In the context of the ANR project RT-Simex, in collaboration with the Obeo company[8], we provided a CCSL library to specify real time constraints on UML models. The CCSL file was created automatically from the Obeo graphical environment (Sirius[9]). Then, some code was generated in order to observe the selected events (model elements associated with clocks). The code is then executed directly on the target platform and an OTF (Open Trace Format [108]) trace is produced. An OTF trace is a fully ordered trace. Therefore, it can be transformed into a synchronous CCSL model. The clock calculus engine of TimeSquare is used to verify that the generated trace is a refinement of the initial CCSL specification. If it does not conform to the specification, the violation is reported along with debug information (like the violated constraint, the step when it happened, the state of other constraints...). The two CCSL models involved do not have the same role. The one extracted from the trace is the actual behavior, the other one is not used to enforce a behavior (as usual CCSL specifications do) but only to verify that the two models are consistent with each other. CCSL constraints are, in that situation, used as assertions.

[8] `http://obeo.fr`
[9] `http://eclipse.org/sirius`

2.4.5. Trace Reconciliation

When several programs execute concurrently on different platforms, each of them produce a trace. If the platform are completely unsynchronized, it is not possible to reconcile the traces together. However, the different parts usually interact with each other. The communications between the two parts are identified on the initial model and can be used as a loose synchronization information. For instance, a message is always sent before being received. This synchronization information can be expressed as a CCSL specification. Combining this specification with the CCSL models extracted from the traces produces a partial order that is analyzed once again with TimeSquare.

2.4.6. Sum-up about TimeSquare

In this section, we briefly presented TimeSquare. It is a model-based tool well integrated in the Model Driven Development process. Its goal is to ease the use of the formal declarative language CCSL. Additionally, we wanted to develop it by using model driven technology; on the one hand it helped in the development of our tool and on the other hand it put the tool in the same technological space than the model under development. The main benefit is the pretty feedback offered to the users during the simulation. A video demonstration is available from the TimeSquare website (in French): `http://timesquare.inria.fr/`. Finally, while not presented here, it also supports exhaustive simulation, Java code generation, Sirius graphical animation, a form of runtime analysis through the generation of VHDL or Esterel observers or the integration with the MARTE profile in Papyrus[10].

We are currently investigating various development directions including (but not limited to): generation of code without relying on Binary Decision Diagrams to enable the embedded of code in platform with small memory footprint; the analysis of potential deadlock (*i.e.*, specification for which some solutions are correct and some others lead to a deadlock) based on the recognition of bad smell patterns on the clock graph; the improvement of the library mechanism to support the definition of constraints with arbitrary numbers of parameters; the addition of the notion of mode allowing to switch between different sets of constraints according to some condition.

[10] see this draft video to have an idea:`https://youtu.be/ZshCHlWs0Xc`

2.5. Using CCSL in different domains

This section is an up to date excerpt of the following papers:

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. doi: 10.1007/s11334-009-0109-0. URL `https://doi.org/10.1007/s11334-009-0109-0`

Frédéric Mallet, Charles André, and Julien DeAntoni. Executing AADL models with UML/MARTE. In *14th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2009, Potsdam, Germany, 2-4 June 2009*, pages 371–376. IEEE Computer Society, 2009. ISBN 978-0-7695-3702-3. doi: 10.1109/ICECCS.2009.10. URL `https://doi.org/10.1109/ICECCS.2009.10`

Calin Glitia, Julien DeAntoni, Frédéric Mallet, Jean-Vivien Millo, Pierre Boulet, and Abdoulaye Gamatié. Progressive and explicit refinement of scheduling for multidimensional data-flow applications using UML MARTE. *Design Autom. for Emb. Sys.*, 19(1-2):1–33, 2015. doi: 10.1007/s10617-014-9140-y. URL `https://doi.org/10.1007/s10617-014-9140-y`

2.5.1. Behavior model for SDF

This first example considers a purely logical case and builds a CCSL library for defining Synchronous Data Flow (SDF) [118] graphs. Subsection 2.5.1.1 recalls basics on Synchronous Data Flow (SDF) (syntax and execution rules). Section 2.5.1.2 proposes a modular CCSL specification to specify the domain specific CCSL constraints appropriated for the specification of SDF behavior models. Finally, an example SDF graph is built using our library. The semantics is given with CCSL and the syntax is built by a UML model with Papyrus. We apply the very same semantic model to two different UML diagrams. The first target is a UML activity, a popular notation to represent data flows. The second target is a UML state machine, whose concrete syntax is close to the usual representation of SDF graphs.

2.5.1.1. Synchronous Data Flow

Data Flow graphs are directed graphs where each node represents a function or a computation and each arc represents a data path. SDF is a special case of data flow in which the number of data samples produced and consumed by each node is specified *a priori*. This simple formalism is well suited for expressing multi-rate DSP algorithms that deal with continuous streams of data. This is a restriction of Kahn process networks [97] to allow static scheduling and ease the parallelization. SDF graphs are essentially equivalent to Computation graphs [98] which have been proven to be a special case of conflict-free Petri nets [148].

In SDF graphs, nodes (called *actors*) represent operations. Arcs carry *tokens*, which represent data values (of any data type) stored in a first-in first-out queue. Actors have inputs and outputs. Each input (resp. output) has a *weight* that represents the number of tokens consumed (resp. produced) when the actor executes. SDF graphs obey the following rules:

- An actor is *enabled* for execution when all inputs are enabled. An input is enabled when enough tokens are available on the incoming arc. Enough tokens means equal to or greater than the input weight. Actors with no inputs (Source actors) are always enabled. Enabling and execution do not depend on the token values, *i.e.*, the control is data-independent.
- When an actor executes, it always produces and consumes the same fixed amount of tokens, in an atomic way. It produces on each output exactly the number of tokens specified by the output

weight; these tokens are written into the queue of the outgoing arc. It consumes on each input exactly the number of tokens specified by the input weight, these tokens are read (and removed) from the queue of the incoming arc.

- *Delay* is a property of an arc. A delay of n samples means that n tokens are initially in the queue of the arc.

2.5.1.2. A CCSL library for SDF

With SDF graphs it is possible to construct locally a set of CCSL constraints for each model element. This section describes the library of CCSL relations built for this purpose.

The first stage is to identify which CCSL clocks must be defined to create a time structure conforming to the SDF semantics. For each *actor* A, one CCSL clock A is created. The instants of this clock represent the atomic execution instants of the operation related to the actor. For each *arc* T, two CCSL clocks $write$ and $read$ are created. Clock $write$ ticks whenever a token is written into the arc queue. Clock $read$ ticks whenever a token is read (and removed) from the queue. Note that, the actual number of available tokens is not directly represented and must be computed, if required, from the difference in the number of read and write operations. No specific clocks are created for inputs and outputs.

The second stage is to apply the right CCSL clock constraints so that the result of the clock calculus can be interpreted to apprehend the behavioral semantics of the SDF graph.

Actors do not require any specific constraint.

Tokens are written into and read from the arc queues. A *token* cannot be read before having been written, hence, for a given arc, the ith tick of $write$ must strictly precede the ith tick of $read$. The kernel CCSL relation strict precedence models such a constraint: $write \boxed{\prec} read$. When $delay > 0$, $delay$ tokens are initially available in the queue, which means that the ith read operation gets the data written at the $(i - delay)^{\text{th}}$ write operation, for $i > delay$. The $delay$ previous read operations actually get tokens initially available and that do not match an actual write operation. CCSL operator delayedFor can represent such a delay (Eq. 2.7). To represent SDF arcs, we propose to create in a library, a new relation definition, called $token$. Such a relation has three parameters: two clocks ($write$ and $read$) and an integer ($delay$). The $token$ relation applies the adequate constraint (Eq. 2.7). Note that, when $delay = 0$, Eq. 2.7 reduces to $write \boxed{\prec} read$.

$$\begin{aligned} &\text{def } token(\text{clock } write, \text{ clock } read, \text{ int } delay) \triangleq \\ &\quad write \boxed{\prec} (read \text{ delayedFor } delay) \end{aligned} \tag{2.7}$$

Inputs require a packet-based precedence (keyword by in Eq. 2.8). A relation definition, called $input$, has three parameters. The clock $actor$ represents the actor with which the input is associated. The clock $read$ represents the reading of tokens on the incoming arc. The strictly positive integer $weight$ represents the input weight.

$$\begin{aligned} &\text{def } input(\text{clock } actor, \text{ clock } read, \text{ int } weight) \triangleq \\ &\quad (read \text{ by } weight) \boxed{\prec} actor \end{aligned} \tag{2.8}$$

Here again, the packet-based precedence can be built with the filtering operator (Eq. 2.9). When $weight = 1$, it reduces to $read \boxed{\prec} actor$.

$$\left(read \triangledown \left(0^{weight-1}.1\right)^{\omega}\right) \boxed{\prec} actor \tag{2.9}$$

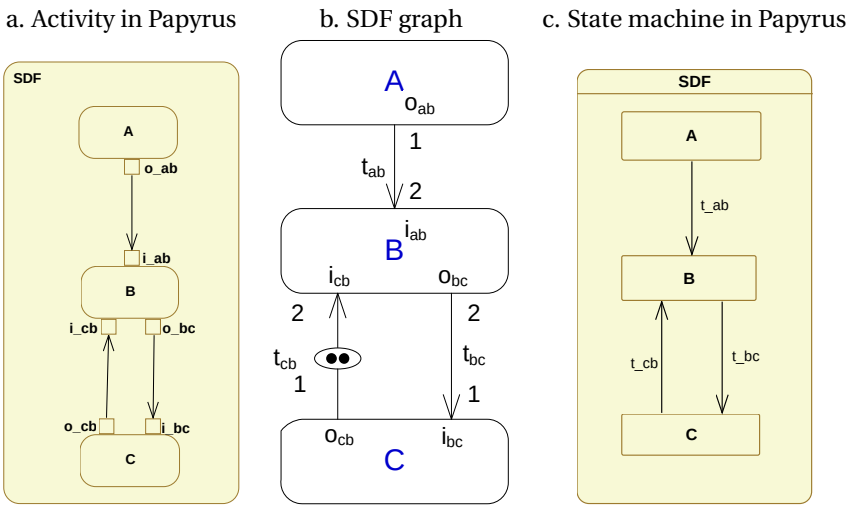

Figure 2.11: An example of SDF graph

Outputs are represented by the relation definition $output$, which has three parameters. The clock $actor$ represents the actor with which the output is associated. The clock $write$ represents the writing of tokens to the outgoing arc. The strictly positive integer $weight$ represents the output weight. CCSL operator filteredBy is used (Eq. 2.10). When $weight = 1$, Eq. 2.10 simplifies into $actor \boxed{=} write$.

$$\begin{aligned} &\text{def } output(\text{clock } actor, \text{ clock } write, \text{ int } weight) \triangleq \\ &\quad actor \boxed{=} \left(write \nabla \left(1.0^{weight-1}\right)^{\omega}\right) \end{aligned} \tag{2.10}$$

Arcs can globally be seen as a conjunction of one output, one token and one input. The library relation definition arc (Eq. 2.11) defines a complete set of constraints for an arc with a delay $delay$, from an actor $source$, with an output weight out to another actor $target$, with an input weight in. In that case, CCSL clocks $write$ and $read$ are local clocks.

$$\begin{aligned} &\text{def } arc(\text{int } delay, \text{ clock } source, \text{ int } out, \text{ clock } target, \text{ int } in) \triangleq \\ &\quad \text{clock } read, write \\ &\quad output(source, write, out) \\ &\quad \mid token(write, read, delay) \\ &\quad \mid input(target, read, in) \end{aligned} \tag{2.11}$$

2.5.1.3. Applying the SDF semantics

The previous subsection defines, for each SDF model element, the CCSL clocks that must be created and the clock constraints to apply. This section illustrates the use of our CCSL library for SDF. Our purpose is to explicitly add to an existing model the SDF semantics. In this example, we use UML activities (Fig. 2.11.a) and state machines (Fig. 2.11.c) to build with the Papyrus editor a simple SDF graph (Fig. 2.11.b). Our intent is NOT to extend the semantics of UML activities and state machines but rather to use Papyrus as a mere graphical editor to build graphs, *i.e.*, nodes connected by arcs. Our proposal is to attach CCSL clocks to some UML elements represented by the modeling editor. Papyrus gives the concrete graphical syntax and the clock constraints give explicitly the expected execution rules.

By instantiating elements of our CCSL library for SDF, these two models can be given the execution semantics as classically understood for SDF graphs. The idea is to add the semantics within the model explicitly without being implicitly bound to some external description. In our case, the semantics is given by the CCSL specification and by explicit associations between CCSL clocks and model elements.

When using the syntax of activities, CCSL clocks that represent actors are associated with actions and CCSL clocks that represent readings from (resp. writings to) the arc queues are associated with input (resp. output) pins. All other CCSL clocks are left unbound and are just used as local clocks with no direct interpretation on the model. When using the syntax of state machines, only CCSL clocks that represent actors are bound to the states and other clocks are left unbound.

Eq. 2.12 uses our library to give to the models on parts a and c the same semantics as understood when considering the SDF graph in the middle part (b). Clocks are named after the model elements with which they are associated, *i.e.*, the clock for actor A is named A. However, this rule is for clarity only and the actual association is done explicitly in the CCSL model.

$$\mathscr{S} \triangleq arc(0,A,1,B,2) \mid arc(0,B,2,C,1) \mid arc(2,C,1,B,2) \tag{2.12}$$

a CCSL specification encodes, in intention, all the schedules acceptable for the model. Fig. 2.12 shows one possible execution of this specification produced by TimeSquare. Intermediate clocks are hidden and only actors are displayed. The relative execution of the actors is what matters when considering SDF graphs.

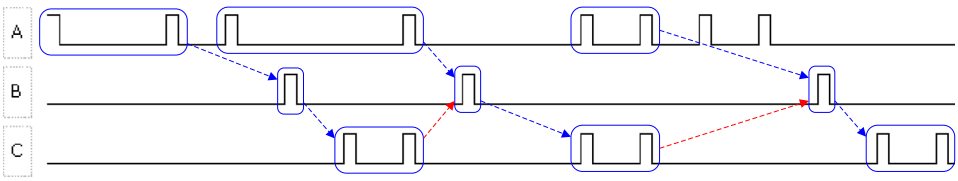

Figure 2.12: A simulation output with TimeSquare.

2.5.1.4. Adding constraints to the SDF Behavior Model

It is of course possible to extend the behavior model with other constraints. For instance the behavior model realized so far for the SDF model in Figure 2.11 represents an infinity of acceptable schedules since actor A is a source and can consequently be executed infinitely many times without any execution of other actors. What is possible is to bound the number of token that can be put in an arc. In this case we can create a new constraint *boundedArc* defined in equation 2.13. It is similar to the *arc* constraint excepted that we add a constraint to specify that the relative advance of *write* on *read* is bounded

$$\begin{aligned}
&\text{def } boundedArc(\text{int } delay, \text{ clock } source, \text{ int } out, \text{ clock } target, \text{ int } in, \text{int} maxToken) \triangleq \\
&\quad \text{clock } read, write \\
&\quad output(source, write, out) \\
&\quad \mid token(write, read, delay) \\
&\quad \mid input(target, read, in) \\
&\quad \mid read \boxed{\prec} write \text{ delayedFor } maxToken
\end{aligned} \tag{2.13}$$

By using the *boundedArc* constraint for source actors, we bound the set of acceptable schedules and make possible to generate the scheduling space representing all acceptable schedules in extensions. Figure 2.13.a and 2.13.b represent two different state spaces when the model is equipped with a boundedArc constraint for arc t_{ab}. On figure 2.13.a the bound is set to 2 and on Figure 2.13.b it is set to 3. While these pictures are not intended to be read, we can see that the scheduling space grows with the storage capacity of the Arc. In these scheduling space, each edge is a set of clocks that tick simultaneously. Every cyclic path is an then infinite schedule of the model. If a state has no outgoing edge, then it represents a path to a deadlock of the system. Such deadlocks are detected during the construction of the scheduling space.

There are many other ways to extends the model behavior, for instance by adding a notion of time or by considering the non atomic execution of Actors.

a. arc t_{ab} bounded to 2 b. arc t_{ab} bounded to 3

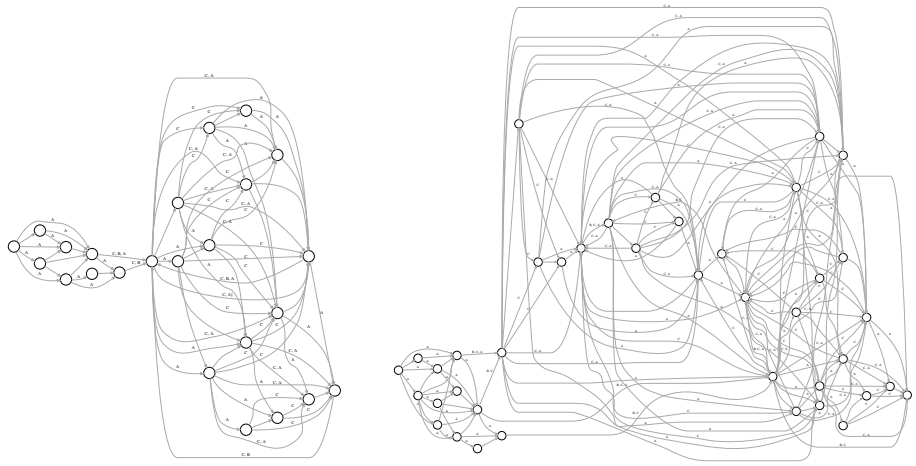

Figure 2.13: Examples of bounded state spaces

2.5.2. Behavior model of AADL

In this second example, we consider AADL and use UML MARTE profile to specify its structure (*i.e.*, software components, execution platform components, express the binding relationships) and CCSL to specify its rich behavior. Our intent is to allow a UML MARTE representation of AADL models and to make it explicitly equipped with the AADL semantics.

2.5.2.1. AADL

Modeling elements AADL supports the modeling of application software components (thread, subprogram, and process), execution platform components (bus, memory, processor, and device) and the *binding* of software onto execution platform. Each model element (software or execution platform) must be defined by a type and comes with at least one implementation.

The latest AADL specification acknowledge that MARTE should be used to provide a UML-based front-end to AADL models and the MARTE specification provides a full annex on the matter [69]. However, even though the annex gives full details on syntactic equivalences between MARTE stereotypes and AADL concepts, it does not say much about the semantic equivalence. Once again we believe that augmenting the model structure with a model of its behavior is of prime importance for correct model manipulation.

AADL application software components Threads are executed within the context of a process, therefore the process implementations must specify the number of threads they execute and their interconnections. Type and implementation declarations also provide a set of properties that characterizes model elements. For threads, AADL standard properties include the dispatch protocol (periodic, aperiodic, sporadic, background), the period (if the dispatch protocol is periodic or sporadic), the deadline, the minimum and maximum execution times, along with many others.

We have created a UML library to model AADL application software components [120]. AADL threads are modeled using the stereotype SwSchedulableResource from the MARTE Software Resource Modeling sub-profile. Its meta-attributes deadlineElements and periodElements explicitly identify the actual properties used to represent the deadline and the period.

AADL flows AADL end-to-end flows explicitly identify a data-stream from sensors to the external environment (actuators). Fig. 2.14 shows an example previously used [70] to discuss flow latency analysis with AADL models.

This flow starts from a sensor (Ds, an aperiodic device instance) and sinks in an actuator (Da, also aperiodic) through two process instances. The first process executes the first two threads while the last thread is executed by the second process. The two devices are part of the execution platform and communicate via a bus (db1) with two processors (cpu1 and cpu2), which host the three threads with several possible bindings. All processes are executed by either the same processor, or any other combination. One possible binding is illustrated by the dashed arrows. The component declarations and implementations are not shown.

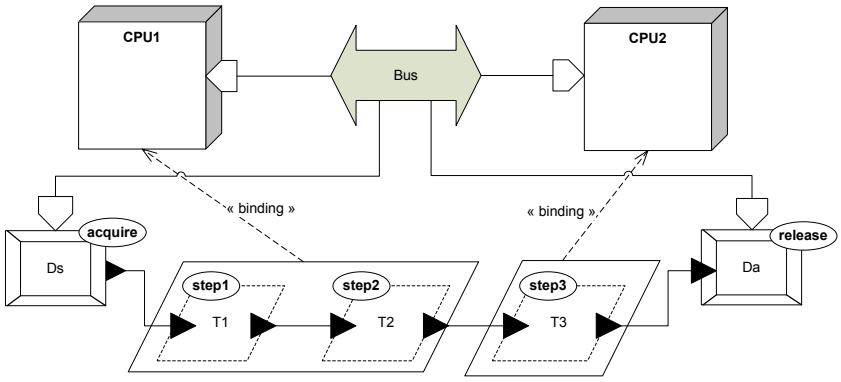

Figure 2.14: The example in AADL (taken from [125])

AADL ports There are three kinds of ports: *data*, *event* and *event-data*. Data ports are for data transmissions without queueing. Connections between data ports are either *immediate* or *delayed*. Event ports are for queued communications. The queue size may induce transfer delays that must be taken into account when performing latency analysis. Event data ports are for message transmission with queueing. Here again the queue size may induce transfer delays. In our example, all components have data ports represented as a filled triangle. We have omitted the ports of the processes since they are required to be of the same type than the connected port declared within the thread declaration and are therefore redundant.

UML components are linked together through ports and connectors. No queues are specifically associated with connectors. The queueing policy is better represented on a UML activity diagram that models the algorithm. A UML activity is the specification of parameterized behavior as the coordinated sequencing of actions. The sequencing is determined by *token flows*. A token contains an object, datum, or locus of control. A token is stored in an activity *node* and can move to another node through an *edge*. Nodes and edges have flow rules that define their semantics. In UML, an *object node* (a special activity node) can contain 0 or many tokens. The number of tokens in a object node can be bounded by setting its property upperBound. The order in which the tokens present in the object node are offered to its outgoing edges can be imposed (property ordering). FIFO (First-In First-Out) is a predefined ordering value. So, object nodes can be used to represent both event and event-data AADL communication links. The token flow represents the communication itself. The standard rule is that only a single token can be chosen at a time. This is fully compatible with the AADL dequeue protocol OneItem. The UML representation of the AADL dequeue protocol AllItems is also possible. This needs the advanced activity concept of *edge weight*, which allows any number of tokens to pass along the edge, in groups at one time. The weight attribute specifies the minimum number of tokens that must traverse the edge at the same time. Setting this attribute to the unlimited weight (denoted '*') means that *all* the tokens at the source are offered to the target.

To model data ports, UML provides «datastore» object nodes. In these nodes, tokens are never consumed thus allowing multiple readings of the same token. Using a data store node with an upper bound equal to one is a good way to represent AADL data port communications.

2.5.2.2. Describing AADL models with MARTE

AADL flows with MARTE We choose to represent the AADL flows using a UML activity diagram. Fig. 2.15 gives the activity diagram equivalent to the AADL example described in Fig. 2.14. The diagram was built

with Papyrus (`https://www.eclipse.org/papyrus/`), an open-source UML graphical editor.

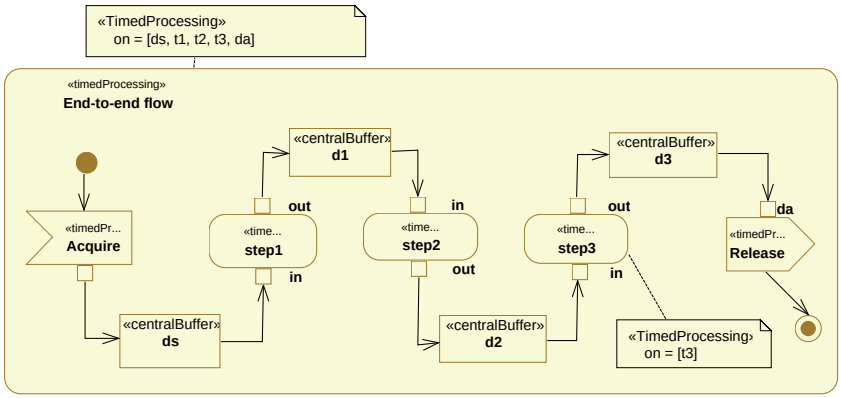

Figure 2.15: End to end flow with UML and MARTE (taken from [125])

As discussed previously, object nodes are used to represent the queues between two tasks. To ease the understanding of the encoding in CCSL, we consider in the following that the tasks are atomics (*i.e.*, we do not take into account neither the start and finish of a tasks nor its worst case execution time. This is however done in [126]. This UML diagram is *untimed* and we use MARTE Time Profile to add time information. This diagram is *a priori* polychronous since each AADL task is independent of the other tasks. The first action to describe the time behavior of this model is to build five logical clocks (ds, $t1$, $t2$, $t3$, da). This is done in two steps. Firstly, a *logical, discrete,* clock type called AADLTask is defined. Then, five instances of this clock type are built. Secondly, the five clocks must be associated with the activity, which is done by applying the stereotype TimedProcessing. As shown in Fig. 2.15, this stereotype is applied to the whole activity but also to the actions. In our case, each action is associated with a different clock. In AADL, the same association is done when binding a subprogram to a task.

Five aperiodic tasks The five clocks are *a priori* independent. The required time behavior is defined by applying clock constraints to these five clocks. The clock constraints to use differ depending on the dispatch protocols of the tasks. *Aperiodic* tasks start their execution when the data is available on their input port in. This is the case for devices, which are aperiodic. The *alternation* relation ($\boxed{\sim}$) can be used to model asynchronous communications. For instance, action Release starts when the data from Step3 is available in d3. $t3$ is the clock associated with Step3 and da is the clock associated with Release. The asynchronous communication is represented as follows: $t3 \boxed{\sim} da$. Fig. 2.16 represents the execution proposed by TimeSquare with only aperiodic tasks with the following constraints: $ds \boxed{\sim} t1$, $t1 \boxed{\sim} t2$, $t2 \boxed{\sim} t3$, $t3 \boxed{\sim} da$. The optional dashed arrows represent instant precedence relations induced by the applied clock constraints.

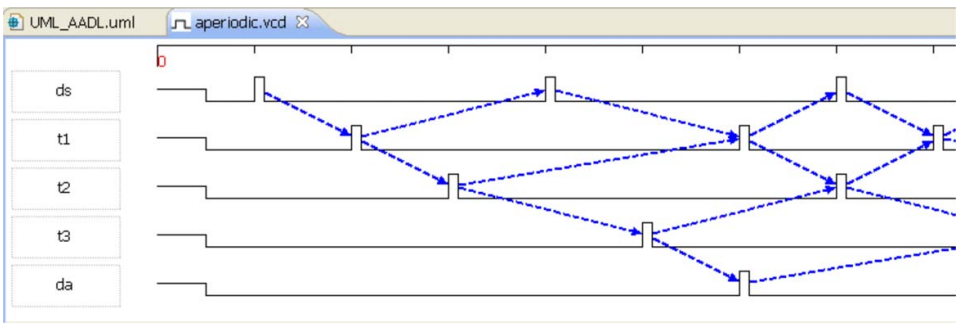

Figure 2.16: Five aperiodic tasks (taken from [125])

Note that this is only an abstraction of the behavior where the task durations are neglected. Additionally, we did not enforce a run to completion execution of the whole activity. Therefore, the behavior is pipelined and ds occurs a second time before the first occurrence of da. This is because the operator

$\boxed{\sim}$ is not transitive. An additional constraint ($ds \boxed{\sim} da$) would be required to ensure the atomic execution of the whole activity. Finally, this *run* is one possible behavior and certainly not the only one. Most of the time, and as in this case, clock constraints only impose a partial ordering on the instants of the clocks. Applying a simulation policy reduces the set of possible solutions. The one applied here is the *random policy* that relies on a pseudo-random number generator. Consequently, the result is not deterministic, but the same simulation can be replayed by restoring the generator *seed*.

Mixing periodic and aperiodic tasks Logical clocks are infinite sets of instants but we do not assume any periodicity, *i.e.*, the distance between successive instants is not relevant. The clock constraint isPeriodicOn allows the creation of a periodic clock from another one. This is a more general notion of periodicity than the general acceptation. A clock $c1$ is said to be periodic on another clock $c2$ with period P if $c1$ ticks every P^{th} ticks of $c2$. In CCSL, this is expressed as follows: $c1$ isPeriodicOn $c2$ period P offset δ.

To build a periodic clock with the usual meaning, the base clock must refer to the physical time, *i.e.*, it must be a chronometric clock. Then, we can discretize idealClk for that purpose and build c_{100}, a 100 Hz clock (Eq. 2.14).

$$c_{100} \boxed{=} idealClk \text{ discretizedBy } 0.01 \tag{2.14}$$

Figure 2.17 illustrates an execution of the same application when the threads $t1$ and $t3$ are periodic. $t1$ and $t3$ are harmonic and $t3$ is twice as slow as $t1$. Coincidence instant relations imposed by the specification are shown with vertical edges with a diamond on one end. Depending on the simulation policy there may also be some opportunistic coincidences. Clock ds is not shown at all in this figure since it is completely independent from other clocks.

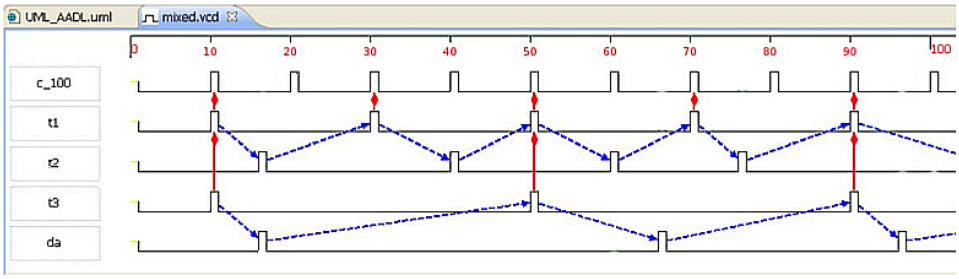

Figure 2.17: Mixing periodic and aperiodic tasks (taken from [125])

Note that, the first execution of $t3$ is synchronous with the first execution of $t1$ even before the first execution of $t2$. Hence, the task $step3$ has no data to consume. This is compatible with the UML semantics only when using data stores. The data stores are non-depleting so if we assume an initialization step to put one data in each data store, the data store can be read several times without any other writing. The execution is allowed, but the result may be difficult to anticipate and the same data will be read several times. When the task $t1$ is slower than $t3$, *i.e.*, when oversampling, some data may be lost.

Note that the complete CCSL specification for this configuration is available in another work [124].

To interpret the result of the simulation, the TimeSquare VCD viewer annotates the VCD with additional information derived from the CCSL specification. We have already discussed the instant relations (dashed arrows and vertical edges). Having a close look at clock da, we can see that its first occurrence is opportunistically coincident with the first occurrence of $t2$. However, the second occurrences of the two clocks are not coincident. Additionally, that specification is *conflict-free* but it may happen that the firing of one clock disables others. These are classical problems occurring when modeling with Petri nets and that appear with CCSL because we have defined precedence instant relations in addition to coincidence relations.

Finally, we have to take into account the execution platform and the allocation. In this example the tasks $t1$ and $t2$ are allocated to the same CPU so it is not possible to execute them simultaneously. The associated constraint is defined in equation 2.15. However, this constraint is no necessary in this specific case since $t1$ and $t2$ alternates. More elaborate execution platforms are considered in the next section.

$$t1 \boxed{\#} t2 \tag{2.15}$$

2.5.3. behavior model of Repetitive Structure Modeling (RSM) models and their execution platform

Computation-intensive embedded systems require the parallel execution of data-intensive computations on several computation units. Process networks [97] and data-flow graphs [82, 115, 117] are adequate to capture the data dependencies of such systems and to compute static schedules that optimize specific criteria such as communication buffer size. Being able to compute static schedules also reduces the scheduling overhead on target platforms, increases predictability, and allows precise performance estimations. Such a schedule must satisfy both the data dependencies and the execution constraints imposed by application functionality, execution platforms and the environment.

The execution ordering depends on constraints of different natures. In general, data dependency constraints induce only a partial execution ordering on the way an application deterministically achieves its associated functionality. In other words, whatever constraints are added to those already induced by data dependencies, the application always computes the same function. As the execution ordering constraints implied by the data dependencies must be respected to ensure that the application computes the right function, we refer to the schedule computed from data dependency constraints as the minimal execution ordering. All valid execution orderings are more constrained (*i.e.*, with less parallelism) than this one.

Indeed dataflow models usually assume an unlimited amount of resources. Taking into account the finite amount of resources available in the execution platform imposes new constraints that reduce the parallelism while still achieving the same function. The resulting execution ordering depends on the number of resources (processors, memories) and on their interconnection topology. A multiprocessor platform with a Network-on-Chip is likely to be more efficient for computation-intensive embedded systems than a uniprocessor platform using a bus. These two platforms will induce different schedules for a given application and hence lead to different execution orderings.

Beyond the data dependencies inherent to the application and execution platform, the environment can also impact the application schedule. Typically, the availability, *i.e.*, arrival ordering, of the inputs of an application from the environment via some sensors influences the scheduling of application tasks for an optimal execution. Similarly, an environment can impose some constraints on the outputs of an application, e.g. some production rate constraints on computed data to be sent to an actuator.

Embedded system design concepts: MARTE profile. The UML profile for Modeling and Analysis of Real-Time and Embedded systems, referred to as MARTE [140], has been recently adopted by the OMG. It extends the Unified Modeling Language (UML) [139] with concepts required to model embedded systems. The *General Component Modeling* (GCM) and *Repetitive Structure Modeling* (RSM) packages offer a support to capture the application functionality. GCM defines all basic concepts such as data flow ports, components and connectors. RSM provides concepts for expressing repetitive structures and regular interconnections. It is essential for the expression of parallelism, in both application modeling and execution platform modeling; and for the allocation of applications onto execution platforms. In the context of application modeling, ARRAY-OL [82] is the underlying Model of Computation and Communication (MoCC) which expresses data dependencies independently of the scheduling. In [82] the authors argue that ARRAY-OL is well suited to model multidimensional signal processing applications and compare this MoCC to others, mainly derivatives of Synchronous DataFlow (SDF) [115]. One of the interesting characteristics of ARRAY-OL is that it is more abstract than the derivatives of SDF. Indeed, it expresses data dependencies without any implied scheduling, thus leaving the choice of the schedule to later design stages. This could be viewed as a weak point of ARRAY-OL because an ARRAY-OL model is not executable but it is actually very useful in our opinion because it leaves all the options open to adapt the schedule to the environment and to the hardware architecture. Thus ARRAY-OL and MARTE RSM allow to express the minimal execution ordering whereas it is not generally the case with derivatives of SDF.

The *Hardware Resource Modeling* (HRM) package, which specializes the concepts of GCM into hardware devices such as processor, memory or buses allows the modeling of the execution platforms in

MARTE. The *Allocation* (Alloc) package allows the modeling of the space-time allocation of application functionality on an execution platform. Both the HRM and Alloc packages can be used with the RSM package to allow a compact modeling of repetitive hardware (*e.g.*, grids of processing elements) and data and computation distributions of a parallel application onto such a repetitive hardware.

The models described with the previous packages can be refined with temporal properties specified within the *Time* package. Such properties are typically clock constraints denoting some activation rate constraints about considered components. The concepts of the Time package are often used with the *Clock Constraint Specification Language* (CCSL) [6], which was initially introduced as a non-normative annex of MARTE.

Our contribution.

Data dependencies are explicitly captured by typical data-flow models. They are often constrained via specific compiler choices to obtain a particular scheduling according to which their corresponding generated code is executed. These scheduling choices, left to a compiler, may not necessarily be well-adapted to scheduling requirements supported by any execution platform. As a result, this does not enable a very flexible choice of execution platforms for a given data dependency specification beyond compiler scheduling constraints. For a better exploitation of compilers in accordance with execution platform constraints, we believe that scheduling constraints should be made explicit by refining given data dependency specifications so that typical execution constraints coming from the environment and platform can be explicitly captured. So the question we are addressing is how we can model all possible schedules and allow designers to select the most appropriate one with respect to the platform and environment induced constraints.

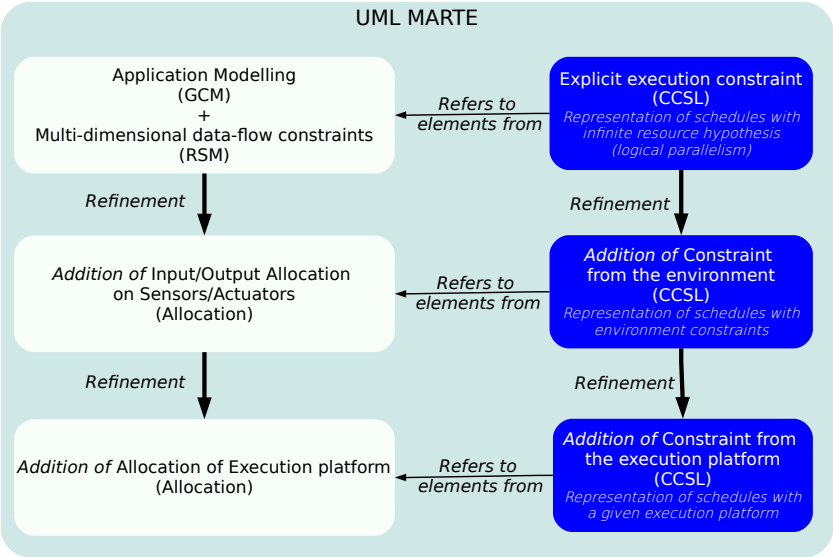

Figure 2.18: Big Picture of the contribution process

We propose a uniform framework, illustrated in Figure 2.18, based on MARTE, for designing computation-intensive embedded systems with the repetitive structure modeling concepts. The resulting descriptions are enriched with clock constraints that explicitly capture information about environment and execution platform properties of systems. The ultimate goal of our framework is to permit an easy high-level design of computation-intensive embedded systems with a possibility to get access to simulation and analysis tools, such as Gaspard2 [74] and TimeSquare[11] for early design validation. The usage of the proposed framework is illustrated on a radar application that aims at detecting from an aircraft the objects moving on the ground. In our opinion, the overall work presented in this paper is one of the first design methodologies about MARTE, which addresses various features of embedded systems by coherently combining a large set of MARTE concepts.

[11] `http://timesquare.inria.fr/`

2.6. Towards Multi-View Behavior Modeling

This section is an up to date excerpt of the following papers:

Carlos Gomez, Julien DeAntoni, and Frederic Mallet. Power Consumption Analysis Using Multi-View Modeling. *Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 235 – 238, 2013

Carlos Gomez, Julien DeAntoni, and Frederic Mallet. Multi-view Power Modeling Based on UML, MARTE and SysML. *Software Engineering and Advanced Applications (SEAA)*, pages 17 – 20, 2012

Amani Khecharem, Carlos Gomez, Julien Deantoni, Frédéric Mallet, and Robert De Simone. Execution of Heterogeneous Models for Thermal Analysis with a Multi-view Approach. In *FDL 2014 : Forum on specification and Design Languages*, Munich, Germany, October 2014. IEEE. URL `https://hal.inria.fr/hal-01060309`

The design of embedded systems requires a strict conformance to the non-functional requirements, such as heat dissipation, energy consumption, safety or time performance. These properties are usually addressed (*i.e.*, modeled and analyzed) by experts of different domains. Experts have their own languages and tools to describe the system modeled from their own point of view. Nevertheless, these expert's models are strongly connected; the behavior of a model could impact the behavior of the other ones. For instance, powering a component off in a power management model affects the functional behavior and the performances of the component in a functional description model.

The recent IEEE 42010 standard [93] proposes a common vocabulary to capture the different *views* of heterogeneous models. *Correspondences* are used to associate domain elements from different views. For instance, in a thermal view, if the temperature of a CPU is reaching the maximum acceptable temperature, the controller may reduce the clock frequency and voltage level (or turn the CPU off) to avoid damaging the CPU. This syntactic link between a non-functional property in a specific view and its impact on the other view should also be considered from a semantic point of view.

More generally, the behavior specification of an embedded system should explicitly take the non-functional properties into account to provide a correct representation of the system behavior. While it may eventually be implemented by using specific sensors, there is a need for early modeling and co-simulation of two types of behavior: one based on a discrete (logical) time representing the functional behavior of the system; and another one based on continuous time representing the evolution of non-functional properties and/or other physical phenomena.

PRISMSYS [84] proposes to tackle the semantics of the MoCs on top of traditional engineering models based on the UML (Unified Modeling Language [143]) or one of its specializations like SysML [142] or MARTE [141]. *PRISMSYS* [84] also proposes to keep the semantics of the MoC explicit and separate from the functional model to ease its extraction, modification and analysis. *PRISMSYS* implements the IEEE 42010 standard. It relies on UML and SysML to capture both functional and non-functional properties [85].

In this paper, we propose an operational framework for the joint simulation of both the discrete and continuous parts of the *PRISMSYS* model. The semantics of *PRISMSYS* models is described using the Clock Constraint Specification Language (CCSL [6, 8]). CCSL is a formal declarative language for discrete logical time specification amenable to analysis and simulation in the associated tool TimeSquare [51]. We have built a dedicated backend for TimeSquare to drive, not only the execution of the discrete part of the model, but also of the continuous part. This particular aspect is delegated to *Scilab* tool [154]. We illustrate the approach on a thermal manager and we present the model, explain the way the backend is built and give some execution results.

The section starts with discussing related works (Section 2.6.1) on both heterogeneous modeling and modeling of thermal aspects. Then, Section 2.6.2 briefly recalls the fundamentals of *PRISMSYS*. Section 2.6.3 describes the execution semantics of *PRISMSYS* for heterogeneous execution based on CCSL.

CCSL is introduced on-the-fly, with only the minimum material required. Finally, the implementation of TimeSquare's backend for Scilab is described in Section 2.6.4 and illustrated on a case study in Section 2.6.5.

2.6.1. Background and related work

2.6.1.1. Heterogeneous Modeling

The joint use of different Models of Computation (MoCs) [63, 94, 95] for a single system is known as *Heterogeneous Modeling*. A MoC (*e.g.*, Synchronous Data Flow, Finite State Machine or Continuous Time) makes explicit the execution semantics applied to a specific syntactic model. Several approaches and tools address the problem of modeling and simulation of heterogeneous models.

Hybrid automata [88] offer a theoretical framework to combine discrete and continuous phenomena. However, they do not offer any specific support for combining heterogeneous models (dataflow, equations, control) or for a clean separation of preoccupations or views.

Ptolemy II [64] relies on a generic actor model to capture hierarchical heterogeneous models. The model of computation (MoC) and its interactions with other MoCs are described in a so-called director that conforms to a predefined Java API. *Ptolemy II* is widely used and has inspired many derived products or evolutions. The main difference with our proposal is that we rely on UML and SysML to capture the structural information instead of a pure actor model and the synchronization and scheduling constraints are given explicitly in CCSL rather than being hidden inside the Java code and mixed with the functional description. Making the MoC explicit and separate allows for reasoning on its properties.

BIP [16] is an interesting alternative that supports explicit heterogeneous interaction models. The behavior is described using timed automata, then several interaction schemes can be used. Contrary to *BIP*, *PRISMSYS* does not require to use state-based representation and may also rely on data flow models or equational models to capture continuous aspects. This increase of expressiveness comes at the cost of decidability results.

There are several works that focus on the interactions between discrete and continuous time, like MatLab/Simulink or Scilab/Scicos. Amongst the solutions that support synchronous models of computation, Zelus [29] provides a nice integration with Lustre. However, in all these solutions, the functional and interaction models are intertwined and hidden as design choices inside the tool. Our proposal is to make the choices explicit in the model (like using a fixed-step solver with a given step) and use the model to drive the tools.

On the engineering side, SysML [142] introduces the notion of parametrics to capture *acausal* models. This is adequate to describe continuous functions or relations and *PRISMSYS* heavily rely on this new construct to capture the thermal and power-related information. Our proposal is to define an execution framework to execute such models.

2.6.1.2. Non-Functional (Thermal) Modeling

Non-functional properties are more and more important in many embedded systems, they must be consequently taken into account during the design process.

Most of the time, non-functional properties are modeled (and analyzed) by using specific tools. For instance, to analyze the impact of temporal properties on the system schedulability, experts in time performance use tools such as *Cheddar* [157] where the focus is on specific temporal properties while the other aspects of the application and its architecture are heavily abstract. Additionally, to study the thermal aspect of a system, temperature experts employ tools like *Hotspot* [91] or *Aceplorer* [61]. Once again, the tools give many details on the properties that deal with their specificity (in this case, the thermal equations), but they abstract strongly the other parts of the system.

In consequence, the study of several non-functional properties requires the duplication of some parts of the system model in these dedicated tools. Due to the strong abstraction of specific parts, these tools can not be used in stand alone. For instance, since *Aceplorer* does not model the system functional

behavior, it needs an execution scenario, generated from a functional system simulation, to activate the elements of its power model and consequently evaluate the power consumption of the system. In the same way, *Hotspot* is based on the *Compact Thermal Model* [167] which defines the model of a the thermal features of the system component and their thermal influence over their neighbors. This model does not represent the system activity needed by *Hotspot* to generate the temperature layout of the system.

One current problem comes from the use of a dedicated controller (*e.g.*, the power manager of an OMAP platform [160]) whose goal is to change the functional behavior in reaction to non-functional property evolution. For instance, because the power consumption of a chip is exponentially increasing with its temperature, a power controller can choose to slow down (or to stop) a chip when its temperature is exceeding a specific threshold. In this case, the functional simulation sets the chip activity according to the chip temperature, but the chip temperature in tools like *Aceplorer* or *Hotspot* require the chip activity. Therefore, there is a cyclic dependency between the execution of the tools that can be solved by a co-simulation of the domain specific models (or tools).

In consequence, it is important to define a central (multi-view) model where the behavioral dependencies are made explicit. The semantics of these dependencies can thus be taken into account to coordinate the domain specific models. In order to specify these dependencies, we define a system-level framework where the behavioral semantics of each parts take into account their link to the other parts of the system. We focused here on the co-simulation of these models to show the benefits of the approach. We named this framework *PRISMSYS*.

2.6.2. *PRISMSYS*: A modeling multi-view framework for systems

PRISMSYS is a multi-view framework for modeling systems, focused on the description and analysis of functional and non-functional aspects. In this framework, a view characterizes the aspects of a specific domain relevant for the system. For instance, time performance model, power model and temperature model are different views of the same system, and they characterize aspects such as time, power consumption and temperature evolution.

The Figure 2.19 depicts the main elements of *PRISMSYS*. The framework defines that a *view* contains up to three *sub-views*: *Control*, *Structural* and *Equational*. The structural sub-view states domain specific elements, which can possibly be an abstraction of elements that exist in another view. These elements define the non-functional properties relevant from the domain point of view. The behavior of these elements is described by Finite State Machine (FSM). The FSM abstracts the operation modes of the element in a specific domain and the transitions are sensitive to events generated from the control sub-view. We use UML components to represent these elements and MARTE Non-Functional Properties (NFP) to type the properties defined in the sub-view elements. The equational sub-view specifies the equations that characterize the properties stated in the structural sub-view. We use SysML Parametric Diagram to represent the equational sub-view. The control sub-view commands the execution of the elements specified in the structural sub-view. A way to specify these view is to use *controllers*, an element whose behavior is stated as a FSM. In contrast to the structural sub-view FSM, the transitions of the controller FSM are sensitive to the evaluation of a condition, *i.e.*, to a guard. Once the transition is fired, an effect is produced, sending an event to the corresponding FSM in the structural sub-view. To distinguish between both FSMs, we employ the term *mode-FSM* to reference the FSM specified in control sub-view.

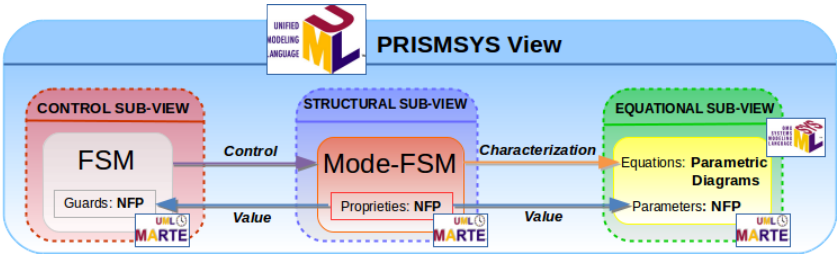

Figure 2.19: *PRISMSYS* View (taken from [101])

The consistency between different views is defined through *correspondences*. A correspondence is a syntactic association between elements of two different views, *e.g.*, the abstraction of an element from a first view characterizes this element from another point of view. Besides specifying associations between views, sub-views also own relationships between them. We named then *sub-correspondences*. For instance, *PRISMSYS* provides a state-equation association through the *characterization* sub-correspondence. This sub-correspondence was defined since it is not possible to specify when a SysML equation is active or not by activating a state in a FSM. *Equivalence* is another sub-correspondence used to bind the structural sub-views properties to the value calculated on the equational sub-view. In order to explicitly address the control events to the mode-FSMs, a *control* sub-correspondence is specified. The values tested in the FSM guards is extracted from the structural sub-view properties by using the *data* sub-correspondence.

A *PRISMSYS* view combines the execution of two kinds of MoCs: FSMs based on a discrete (logical) time, and the equations based on a Continuous Time. This MoC combination could be *homogeneous* (FSM and Mode-FSM) and *heterogeneous* (Mode-FSM and Continuous Time). In order to give an execution semantics to FSM, Mode-FSM, Continuous Time and the coordination between them, *PRISMSYS* defines the relevant events of the models together with constraints that specify the causal relations and synchronizations. Each relevant action in a MoC is expressed by an event and we use CCSL [4], an event constraint language, to represent the allowed sequence of actions as defined by the MoC.

2.6.3. *PRISMSYS* Execution Semantics

PRISMSYS behaviors, which initially are described by a static definition (meta-model), need a formal way to express their actions associated with their static elements. CCSL provides the needed concepts to express the actions of these behaviors allowing the association action-element and the behavioral analysis by simulation. First, we define the FSM execution semantics. Second, we state the execution semantics in Continuous Time. Finally, we describe the coordination between FSMs and Continuous Time.

2.6.3.1. Finite State Machine Semantics

In a FSM, there are various relevant events that occur during its execution. Most of the FSM concepts are associated with one or more events that describe a particular FSM execution change, *e.g.*, the entering in a state or the firing of a transition. These events are represented in CCSL by clocks. Table 2.2 summarizes the clocks defined to represent the activity in the FSM.

Clock	Action
$init$	Initializing the execution of FSM
s_{enter}	Entering into state s
s_{leave}	Leaving from state s
$fire_t_{ij}$	Firing the transition t_{ij} from s_i to s_j
$guard_{ij}$	Ticking once the t_{ij} guard evaluation is true
$trigger_{ij}$	Receiving a trigger event of t_{ij}
$effect_{ij}$	Generating an Event once t_{ij} is fired

Table 2.2: Clocks representing the relevant actions in a Finite State Machine.

We give a precision about the FSM model, we constrain that a transition t_{ij} is sensitive to either a $guard_{ij}$ or a $trigger_{ij}$, however not both of them. If a guard is defined, then $guard_{ij}$ clock is stated, else $trigger_{ij}$ is specified.

Once the FSM clocks are defined, we identify the constraints on these clocks to describe the FSM execution semantics. We start defining the activation of a specific state. A state is active between the corresponding entering and leaving occurrences. In other words, the state s is active when the kth occurrence of s_{enter} clock ticks and it stops being active when the kth occurrence of s_{leave} ticks. We specified that a state cannot be transitory, *i.e.*, the s_{enter} and s_{leave} ticks cannot be simultaneous.

Moreover, a state can not be activated if it is already active. Consequently, we specified an *Alternate* relation between s_{enter} and s_{leave} in CCSL for all the states of FSM as follows:

$$s_{enter} \boxed{\sim} s_{leave} \tag{2.16}$$

According to the execution semantics of FSM [36], a transition t_{ij} from a state s_i to a state s_j is fired if two conditions are achieved: s_i is active, and either the t_{ij} guard evaluation is true or $trigger_{ij}$ ticks. However, the instant when the guard is evaluated is not specified. Because it affects the timing properties, it is of primary importance for a state machine of an embedded system to define when the guard is evaluated. In *PRISMSYS*, we explicitly define a *chronometric clock* named *eval* that specifies the period when the guard is evaluated. This clock is the same for all the guards in the system and its period must be chosen according the non-functional property dynamic (see section 2.6.3.3). Hence, when the evaluation of the t_{ij} guard condition returns true, $guard_{ij}$ occurs. Considering that s_i is active and $guard_{ij}$ ticks, then the t_{ij} transition is fired. In contrast, if a transition is sensitive to a trigger event, the t_{ij} transition is fired when $trigger_{ij}$ ticks. We specify the relationship of these clocks by using CCSL expressions. We present the CCSL constraints that define when to fire the t_{ij} transition by $guard_{ij}$:

$$fire_t_{ij} \boxed{=} [(s_{i_{enter}} \Downarrow guard_{ij}) \lightning s_{i_{leave}}] \bullet fire_t_{ij}$$

this expression can be read as: a transition t_{ij} is fired if $guard_{ij}$ ticks while the state is active (*i.e.*, after a $s_{i_{enter}}$ occurrence and up to an occurrence of $s_{i_{leave}}$).

The state s_i stops being active (*i.e.*, $s_{i_{leave}}$ ticks) when one of its outgoing transitions occurs. The relationship between the leaving of a state and the firing of the outgoing transitions is specified as follows:

$$s_{i_{leave}} \boxed{=} \bigcup_{t \in t_{out_{s_i}}} fire_t$$

where $t_{out_{s_i}}$ is the set of outgoing transitions from s_i.

When a transition is fired, the targeted state is entered simultaneously, *i.e.*, the entering in the s_j state coincides with any of the incoming transition firing. We specify this coincidence relationship by:

$$s_{j_{enter}} \boxed{=} \bigcup_{t \in t_{in_{s_j}}} fire_t$$

where $t_{in_{s_j}}$ is the set of incoming transitions to s_j.

If the transition t_{ij} has an action, the $effect_{ij}$ clock occurs simultaneously with the transition firing. This relationship is specified by:

$$effect_{ij} \boxed{=} fire_t_{ij}$$

The UML state machines must have at least one initial state. We constrain them to have one and only one initial state. The clock *init* is specified to start the FSM execution by activating the initial state. We only need one tick in *init* to activate the initial state. Thus, we state $init$ in CCSL as follows:

$$init \boxed{=} init \nabla 1(0)^{w}$$

this equation forces $init$ to tick only once.

The $init$ clock must be associated with the initial state. Considering that s_{init} is the initial state of the FSM, we define its activation as follows:

$$s_{init_{enter}} \boxed{=} init$$

By using these rules, we defined the event based semantics of the UML state machine. It is then possible to reason about a specific state machine and to simulate it in TimeSquare, obtaining a timing diagram and/or a diagram animation. It is now mandatory to define the continuous time execution semantics.

2.6.3.2. Continuous Time Execution Semantics

In the considered systems, the non-functional properties evolves according to the time. There are two main families of continuous time solver, the fixed-step solvers and the variable step solver. A fixed-step solver evaluates the equation of the modeled process periodically with a predefined step size. The disadvantage of such solver is that they can hide some discontinuities in the modeled process if the step size is too big. The variable step solvers adapt the step size to the dynamic of the modeled process in order to avoid hiding such discontinuities. This is of great importance when the goal is to provide an accurate representation of the modeled process.

In our case, we model by equation the evolution of non-functional properties according to the time. One important goal of this work is to allow the prototyping of non-functional property managers. Eventually, these managers will retrieve the value of the corresponding non-functional properties by using (periodic) sensors. Consequently, we do not try to obtain the best representation of the non-functional property evolution, but rather a representative view of the process at specific point in time, as seen by the managers. Therefore, we can use either fixed-step or variable steps solvers but we do not use zero crossing functionality and we explicitly specify in the UML model the point in time when the value from the process are retrieved. To ease the implementation we used for now only fixed step solvers so that the points in time when the value from the process are retrieved are periodic and correspond to the solver fix step.

In the model, to specify the step size used by the solver, we employ a *chronometric clock* from MARTE. In CCSL, this is encoded by discretizing a dense clock (*i.e.*, a clock in which whatever pair of two instants you choose, there exists always one instant between these two instants). We use the physical time dense clock defined in CCSL and we name the discretized clock *step*. At each occurrence of *step*, the active equations specified in the model are evaluated by the continuous time solver. In CCSL, the *step* clock is defined by:

$$step \triangleq physicalTime \textbf{ discretizedBy } \Delta t \tag{2.17}$$

where Δt is the step size as defined in the UML model and used by the fixed-step solver.

Once the FSM and Continuous Time execution semantics are defined, we must coordinate the actions of both semantics to achieve a complete execution of a *PRISMSYS* view.

2.6.3.3. FSM and Continuous Time Coordination

We provided the event based semantics for the UML state machine (FSM) and for the SysML parametric diagram (CT). Control and structural sub-views follow the FSM semantics and the equational sub-view follows a Continuous Time semantics. Instead of providing a tailored semantics for a new model mixing both semantics, such as in an hybrid automata [36], we propose to define the semantics of the syntactic association between the different sub-views (named sub-correspondences in *PRISMSYS*). The coordination semantics is also specified in CCSL.

In the structural sub-view, the FSM are representing some mode automata. The transition between two modes is caused by the reception of an event sent from the control sub-view. Therefore, there is a correspondence between the output events generated from the control sub-view and the events associated with the mode transitions. This correspondence is syntactically represented by control connectors between control and structural sub-views.

From the behavioral semantic perspective, a control event generated from the FSM of the control sub-view can *cause* a mode transition in the mode automata of an element in the structural sub-view. This relation can be tailored to be synchronous, purely causal or even timed. Here we chose that the sending and the reception of an event occurrence is simultaneous. By using CCSL, if an event occurrence is transferred by a control connector from a transition t_{ij} to the trigger of a mode automata ($trigger_{mode_automata}$) we enforce the following coincidence relation:

$$effect_{ij} \boxed{=} trigger_{mode_automata}$$

we read this expression as a control event, produced during the transition in a control FSM ($effect_{ij}$), coincides with the trigger of a transition in the mode automata ($trigger_{mode_automata}$).

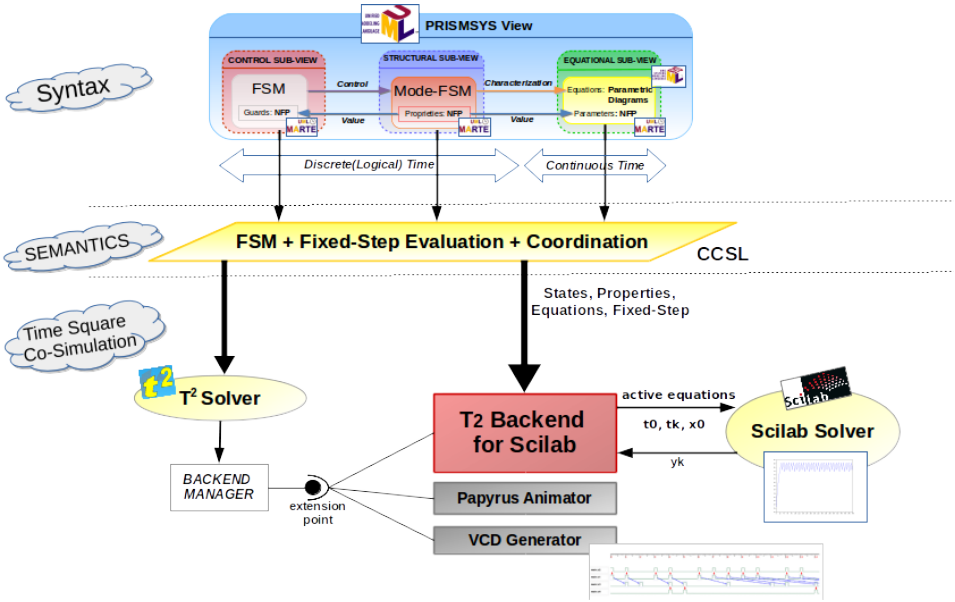

Figure 2.20: T2 BACKEND FOR SCILAB Overview on a *PRISMSYS* thermal view case study (taken from [101])

In *PRISMSYS*, the mode automata are used to specify the modes under which there are different law that govern the evolution of a non-functional property. For instance the instantaneous consumption of a CPU depends on its activity (*e.g.*, sleep, idle or active). These evolution are specified by different equations that are enabled or not according to the active state of a mode-FSM (see section 2.6.2). To represent this semantics, a mode-FSM and an equation are syntactically linked by a *characterization* sub-correspondence. When a mode becomes active (*i.e.*, the s_{enter} clock ticks), the associated equation is enabled. When the mode is left, (*i.e.*, the s_{leave} clock ticks), the corresponding equation is disabled. Of course, for the model to be consistent, a non functional property should be at least defined by one equation during a run.

We have two chronometric clocks in the system. The one used for the evaluation of the guard and the one that define the fix step of the solver. To avoid the evaluation of the guards without a new value computed by the equation, the chronometric clock used for the solver step must be the same or faster than the one used to evaluate the guards. We express this coordination in CCSL as:

$$step \boxed{\prec} eval \tag{2.18}$$

where $eval$ is the clock whose instants command the evaluation of the guards and $step$ is the fix step of the solver. If these two clocks are simultaneous, it is mandatory to evaluate the equations before the evaluation of the guards to respect the causality (even if instantaneous in this case). It guarantees that the non-functional properties used in the guards have the latest value resulting from equation evaluation.

The previous sections specified the execution semantics of *PRISMSYS*. The remainder of the paper presents the associated implementation.

2.6.4. *PRISMSYS* Model Co-Simulation

TimeSquare is an Eclipse and model-based environment for the specification, analysis and verification of causal and temporal constraints. Particularly, TimeSquare allows defining, analyzing and verifying a CCSL specification. Nevertheless, this tool is not adapted to evaluate equations (*e.g.*, power and temperature equations). Therefore, we use *Scilab* [154], an open source tool for numerical computation, as a fixed-step solver for the evaluation of the equations. To coordinate the TimeSquare solver with the *Scilab* solver, we implemented a TimeSquare backend that acts like a "*connector*" between these tools. We named it T2 BACKEND FOR SCILAB.

Figure 2.20 depicts an overview of this implementation. It shows three dependent domains: the one on top is the *PRISMSYS* model, detailed in Section 2.6.2. The one at bottom describes the TimeSquare and *Scilab* co-simulation and the one in the middle defines the *PRISMSYS* semantics to execute and to coordinate the behavior of the *PRISMSYS* model. We focus in this section on the block at the bottom, the T2 BACKEND FOR SCILAB.

TimeSquare offers the possibility to add user-defined backends that trigger specific actions on selected event occurrences or relations. These new backends can be added to the *backend manager* by using an eclipse extension point. During a simulation, the backend manager receives the status of each clock (it ticks or not) at each simulation step. It also receives the status of relations (causality and coincidence) as well as the status of the assertions (violated or not). A developer can create its own backend that subscribes to some of these events. TimeSquare is already distributed with some backends like the *VCD backend* that draws the timing diagram of the clocks during the simulation and the *Papyrus Animator* that animates the elements of an UML model specified in *Papyrus*[12]. Alike, by using the backend manager and connecting to the specific extension point, we implement T2 BACKEND FOR SCILAB as an *Eclipse* Plug-in to bind the TimeSquare solver module with the *Scilab* solver. This back-end receives the status of the clock (it ticks or not) at each simulation *step* and consequently configures the *Scilab* solver according to the information in the UML *PRISMSYS* model and drives the *Scilab* solver to evaluate the equations during the simulation. The user can visualize the value of the non-functional properties thanks to the *Scilab* graphic window called by the *Scilab* solver.

The solver configuration consists of the extraction of different equations from the model as well as the extraction of the chronometric clock that defines the solver step size. It is mandatory for the backend to register to the entering and leaving clock of all the mode-FSM states linked to an equation in order to add or remove it from the *Scilab* solver accordingly. This backend also commands the evaluation of the guards, therefore such pieces of information must also be extracted from the *PRISMSYS* model. During the simulation, the T2 BACKEND FOR SCILAB receives the status of each relevant clocks. When an entering state ($s_{i_{enter}}$) clock ticks, T2 BACKEND FOR SCILAB adds the equation(s) associated with the active state. When the *step* clock ticks, the active equations are evaluated by the *Scilab* solver. The result of the evaluation is plotted in a *Scilab* plot window and stored in the backend. For each *step* occurrence, the guards are evaluated. If a guard is evaluated to true, T2 BACKEND FOR SCILAB forces the associated guard clock occurrence in the TimeSquare solver (see $guard_{ij}$ in Section 2.6.3). Such occurrences can cause a change of the *controller* state, and then a mode change. We can note that the guard evaluation is done for each *step* occurrence and not necessarily immediately when the parameter exceeds the thresholds. It emphasizes the importance to select the appropriated solver step size in order to react soon enough to the evolution of the non-functional properties.

2.6.5. Case Study: CPU thermal controller

The case study shows how a thermal controller can be prototyped by using *PRISMSYS*. The temperature, a non-functional property, must be monitored in order to reduce the power consumption, extending the battery charge and fulfilling the temperature requirements. The CPU is the component on which we focus in this case study. We define a CPU muti-view model by using *PRISMSYS*. In Figure 2.21, we present a simplified version of the thermal view model. The figure depicts the three *PRISMSYS* behaviors (control, modes and equations).

The structural sub-view of this figure represents the abstraction of a CPU from a thermal point of view. It describes a non-functional property representing the temperature (T). The equational sub-view defines the equations that characterize the evolution of the CPU temperature. The Control sub-view specifies a *thermal controller*, which changes the activity of the CPU according to its temperature to avoid damages.

The CPU thermal behavior is represented by two modes: *COOLING* and *HEATING*. Each mode corresponds to a different temperature evolution. *COOLING* indicates the CPU temperature is decreasing. In contrast, *HEATING* means that the CPU temperature is increasing. In order to define the evolution of the temperature, an equation, defined in the equational sub-view, is associated (point 2 in Figure 2.21). Thus, we associate the *HEATING* mode with the equation $dT/dt = -0.1 * (T - 100)$ and the *COOLING*

[12] `http://www.eclipse.org/papyrus/`

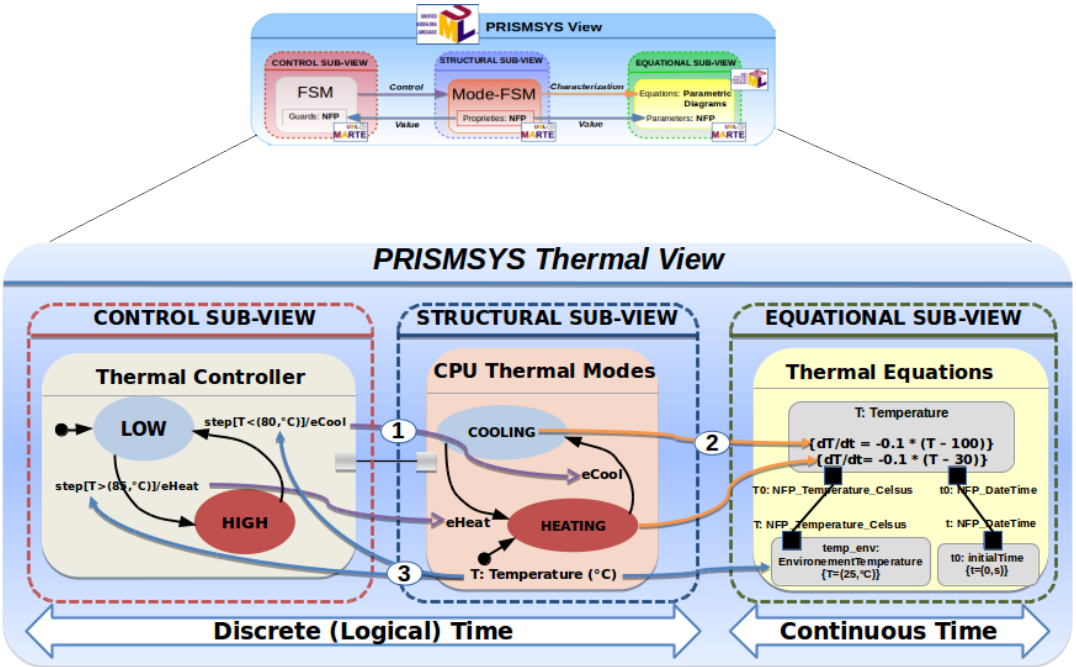

Figure 2.21: Case Study: CPU Thermal Controller (taken from [101])

mode with the equation $dT/dt = -0.1 * (T - 30)$ where 100 is the temperature maximum for the CPU and 30 is its minimal idle temperature. The unit of both temperatures is $^{\circ}C$.

The change between two modes occurs when the transition between them is fired. Transitions are sensitive to associated events sent from the control sub-view (point 1 in Figure 2.21). The control sub-view contains a *thermal controller* whose task is to manage the temperature evolution of the CPU. The *controller* behavior is another state machine of two states: *LOW* and *HIGH*. *LOW* represents that the CPU temperature is below the recommended temperature. *HIGH* expresses the temperature exceeds the maximum recommended value. In contrast to the CPU thermal FSM, the transition of the controller is sensitive to the evaluation of guards. In this case study, guards evaluate if the temperature surpasses $85^{\circ}C$ or is lower than $80^{\circ}C$ (point 3 in Figure 2.21). Once a transition is fired, a specific event is sent to the CPU thermal FSM in order to change its mode and thus activate the equation to be evaluated.

Figure 2.22 depicts the evolution (plotted in *Scilab*) of the temperature T according to the time t. When the simulation starts, *i.e.*, at t=0, T2 BACKEND FOR SCILAB extracts the clocks associated with modes *COOLING* and *HEATING* and the step clock. T2 BACKEND FOR SCILAB also identifies and extracts the thermal equations. In our thermal model, we assume that the ambient temperature is $25^{\circ}C$. Thus, the TimeSquare simulation starts at $25^{\circ}C$ and the activating *LOW* and *HEATING* states (which are the initial states) are activated. When the temperature exceeds $85^{\circ}C$, the transition *lo2hi* is fired, the state changes to *HIGH* and the *eCool* event is sent to the CPU Thermal Modes (other events, which set the CPU activity to idle in other views, are also sent but not represented here). Such an event causes the leaving of the *HEATING* mode to enter in the *COOLING*, consequently deactivating the equation $dT/dt = -0.6 * (T - 100)$ and activating the equation $dT/dt = -0.6 * (T - 30)$. In the next simulation steps, the associated *COOLING* equation is evaluated and the temperature decreases. When the temperature is lower than $80^{\circ}C$, the *controller* state machine changes to the *LOW* state and the cycle is repeated.

In Figure 2.22, we note that the temperature actually exceeds some how the threshold defined in the guard. This is due to the periodic monitoring of the temperature and the periodic evaluation of the guard. To limit such phenomenon, one could decrease the step size. However, this phenomenon has to be taken into account during the development of the system.

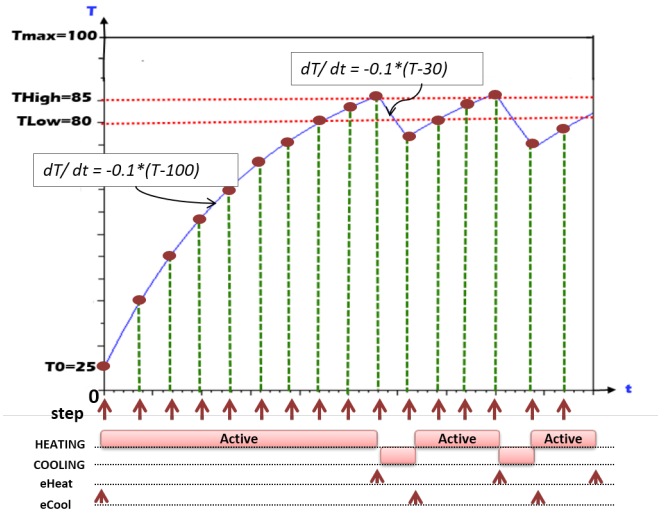

Figure 2.22: *PRISMSYS ThermalView* simulation in *Scilab* by using T2 BACKEND FOR SCILAB. (taken from [101])

2.7. Conclusion

In this section we described how we used logical time formal specification to augment the structure of a model with an explicit specification of its behavior. We presented CCSL, the formal declarative language we used to manipulate logical time. Its formal structure and semantics has been presented in detail and is followed by a comparison with temporal logics. Then the TimeSquare tool that supports, possibly exhaustive simulation of CCSL specification has been presented. After this presentation of CCSL and the associated tool, we illustrated how CCSL has been used in different domains, encompassing both application and execution platform, in order to equip the associated models with a model of their behavior. The behavior are then given in terms of how the actions on the elements of model are scheduled, typically to represent data causality. We believe this is of prime importance to avoid early restrictions on the partial ordering of actions, which can be later constrained by non functional constraints or the choice of a specific execution platform. These acceptable partial orderings can then be simulated or specified in extension by an exhaustive simulation (when finite). Additionally, associated to the code executor back-end of TimeSquare, the model can be executed and debugged.

The possibility to verify a model from the early stages of the development process, associated with the possibility to take into account the effect of external factors is an important benefit of having an explicit model of the possible execution path. Among the external factor, there is the deployment on a specific platform. We also investigated in the previous section the effects due to correspondences between different views of the system. While this last approach is still exploratory, it shown the possibility to define semantics of correspondences to reason about the coordination of heterogeneous systems. We are currently investigating how such mechanism could be extended to co-simulation in its more general terms than the one presented in section 2.6.

Chapter 3

Modeling Concurrent and Timed Operational Semantics of Languages

Contents

3.1 Introduction **48**
3.2 Ingredients of a Concurrency-Aware Executable DSML **49**
3.3 A Language Workbench to Design and Implement Concurrency-Aware Executable DSMLs **53**
3.4 Integration of the approach in the GEMOC Studio **65**
3.5 Validation and Discussion **69**
3.6 Related Work **70**
3.7 Conclusion about the Modeling of Concurrent and Timed Operational Semantics. . **71**

This chapter is an up to date, both partial and extended view of the following papers:

Benoit Combemale, Julien Deantoni, Matias Vara Larsen, Frédéric Mallet, Olivier Barais, Benoit Baudry, and Robert France. Reifying concurrency for executable metamodeling. In *the 6th International Conference on Software Language Engineering (SLE 2013)*, oct 2013

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)*, Grenoble, France, March 2015. URL `https://hal.inria.fr/hal-01087442`

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)*, Pittsburg, United States, 2015. ACM. URL `https://hal.inria.fr/hal-01185911`

Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale. Execution Framework of the GEMOC Studio (Tool Demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, page 8, Amsterdam, Netherlands, October 2016. URL `https://hal.inria.fr/hal-01355391`

3.1. Introduction

Software-intensive system development must tame an increasing complexity, in an increasing number of different inter related domains. As an element of the solution, the use of domain-specific modeling languages (DSMLs) helps in increasing productivity while providing effective support for separating concerns. DSMLs can make it easier for stakeholders from different domains (*e.g.*, experts in fault tolerance, security, communication) to participate in the design of a system, by providing linguistic concepts tailored to their specific needs. However, for a DSML to be an effective system design tool, it must be defined as precisely as possible and supported by sound analysis tools [43].

The specification, design and tooling of DSMLs leverage the rich state of the art in language theory. Several metamodeling environments support the specification of the syntax and the (static and dynamic) semantics of a DSML. These two elements of a DSML specify the domain-specific concepts, as well as the meanings of domain-specific actions that manipulate these concepts. Examples of metamodeling environments include Microsoft's DSL tools [1], Eclipse Modeling Framework (EMF) [2], Generic Modeling Environment (GME) [3], and MetaEdit+ [4]. A significant limitation of current metamodeling environments is the lack of support for explicitly modeling concurrency semantics. Concurrency semantics is currently defined implicitly in DSMLs that support concurrent execution in their models. It is typically embedded in the underlying execution environment supported by the language workbench used to implement the DSMLs (*e.g.*, if the language runs on top of a Java Virtual Machine, the semantics of Java threads defines concurrent behavior).

The lack of an explicit concurrency model has several drawbacks. It not only hinders a comprehensive understanding of the behavioral semantics, it also prevents developing effective concurrency-aware analysis techniques. For instance, knowing that a data-flow model (*e.g.*, an activity diagram) follows Kahn process networks semantics ensures *de-facto* properties like latency-insensitive functional determinism but imposes communications through unbounded FIFOs. Restricting the data-flow model to the Synchronous Data Flows semantics allows the computation of finite bounds on the communication buffer sizes. Furthermore, having an implicit concurrency model also prevents the distinction of semantic variants in a model. For example, the fUML specification identifies several semantic variation points. As stated in the fUML specification, some semantic areas "are not explicitly constrained by the execution model: The semantics of time, the semantics of concurrency, and the semantics of inter-object communications mechanism" [138]. The lack of an explicit model of concurrency, including time and communication, prevents one from understanding the impact of these variation points on the execution of a conforming model. Additionally, it prevents to consciously align the concurrency of an application with the parallelism provided by a specific platform.

In this chapter we reify concurrency as a metamodeling facility. We leverage formalization work on concurrency and time from concurrency theory, specifically, theoretical work on tagged structures [116] and on heterogeneous composition of models of computation [64, 94]. The primary contribution of this chapter is an approach supported by a language workbench for binding domain specific concepts and models of computation through an explicit event structure at the metamodel level. We illustrate these novel metamodeling facilities by designing a DSML specifying concurrent and timed finite state machines. We highlight the benefits and the flexibility of the approach by making a semantic variation on the concurrency specification of the DSML. We also provide pointers to other examples to show that our approach applies to different MoCs and DSMLs.

The chapter is organized as follows. Section 3.2 uses background on language and concurrency theories to identify the key ingredients of a concurrency-aware executable DSML, and to reify them as the association of four language units. Section 3.3 describes the language workbench built to implement the proposal, and the associated environment for concurrent model execution. Section 3.5 demonstrates and discusses the DSML implementation and execution environment obtained thanks to our language workbench. The approach is illustrated throughout the paper with the design, implementation and use of timed finite state machines. A comparison to related work and a conclusion follow.

[1] http://www.microsoft.com/en-us/download/details.aspx?id=2379
[2] http://www.eclipse.org/modeling/emf/
[3] http://www.isis.vanderbilt.edu/Projects/gme/
[4] http://www.metacase.com/mep/

3.2. Ingredients of a Concurrency-Aware Executable DSML

3.2.1. 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 [17, 99, 109, 149]; *axiomatic semantics*, where predicates on the execution state allow reasoning about the models expressed in the language and its correct evolution [86, 89, 171]; and *translational semantics* [73] 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 [155]). 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 [171]). 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 [132], CSP [90] and Petri Nets [147]. 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 [170] and tagged structures [116]. 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.

3.2.2. 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. 3.1). 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. 3.1). 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 [130]) have

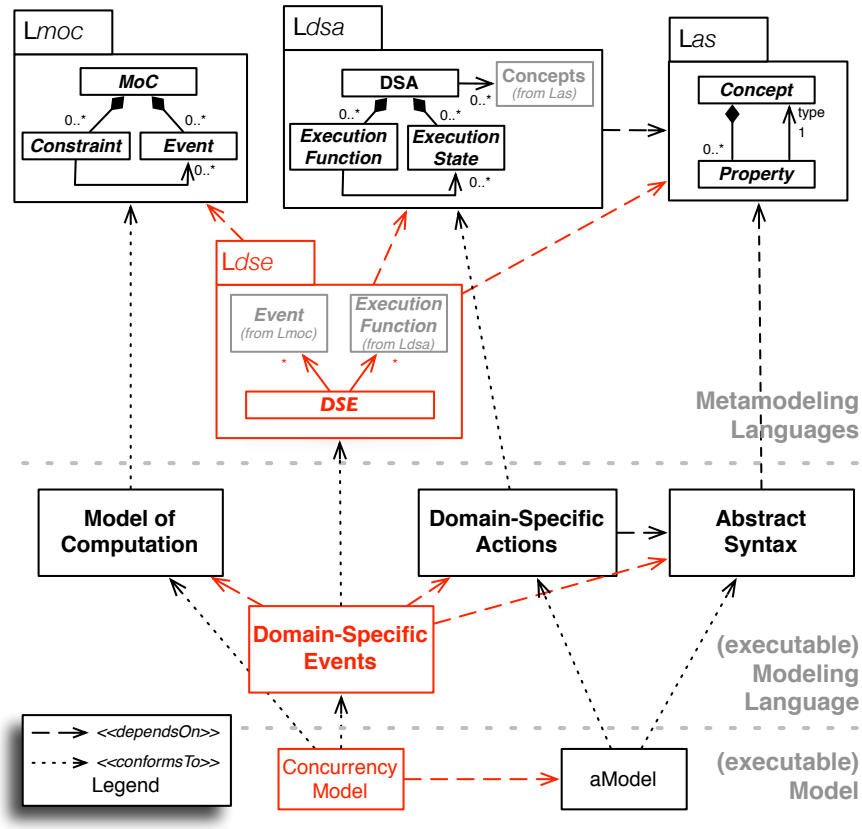

Figure 3.1: Modular Design of a Concurrency-Aware Executable Modeling Language (taken from [44])

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 the syntactic domain with language elements that represent the execution state of the model.

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. 3.1) 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* (see Fig. 3.1), adds new properties that represent the *execution state* of a model and a set of *execution functions* that operate on these properties during the execution of a model.

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 *represents 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. 3.1). However, the specification of the DSA depend 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) [27, 64, 94]. 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 Concurrency and Communication* (see Fig. 3.1) and explicitly specifies the concurrency.

Definition 3 *The Model of Concurrency and Communication (*MoCC*) represents the concurrency aspects in a language, including the synchronizations and the, possibly timed, causality relationships between the execution functions. An instance of a* MoCC *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 MoCC ($Lmoc$ in Fig. 3.1) 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.

3.2.3. Reifying Language Units Coordination

In our approach, all language units previously presented are specified separately (see middle level of Fig. 3.1). 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. 3.1), 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 MoCC with the DSA. This coordination must keep the MoCC and DSA independent to enable

the (re)use of a MoCC on different AS/DSA or changing the MoCCs on a single AS/DSA. Hence, the coordination specification can be put neither directly in the MoCC nor in the DSA. For this reason, we reify the binding as a proper language unit that bridges the gap between the MoCC 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. 3.1) specify the coordination between the events from the MoCC and the execution function calls from the DSA. The DSE depend on both the MoCC 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. 3.1) 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).

MoCC → DSE The MoCC events can be specified at an abstraction level that is different from the execution functions (defined in the DSA). For this reason, the DSE specify how they are obtained from the events constrained by the MoCC. 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 MoCC. In this case the DSE are used to specify the relevant observations on the dense event from the MoCC 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 MoCC 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 MoCC.

DSA → DSE The MoCC and the DSE represent the specification, at the language level of the concurrency model (dedicated to a specific model conforming to the DSML), see Figure 3.1. This concurrency model specifies the acceptable partial orderings of both the events constrained by the MoCC 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 MoCC 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.

MoCC ← DSE → AS Finally , the DSE must specify how the MoCC is applied on a specific model that conforms to the DSML (*i.e.*, how to create the concurrency model according to the MoCC constraints and the AS concepts). Depending on the language used for the MoCC 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 MoCC assumptions (*e.g.*, by using proven compilers or a language supporting clear and simple composition of constraints from the MoCC).

Definition 4 *The Domain Specific Events (DSE) represent a coordination between the* MoCC *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* MoCC *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* MoCC *application on a specific model that conforms to the DSML.*

As highlighted by the previous description, the coordination between the MoCC 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 3.1. Consequently, we consider in this paper the following definition for a concurrency-aware executable DSML:

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* MoCC *and the DSE to coordinate them). In the context of this paper, a concurrency-aware executable DSML (xDSML) is defined as a tuple:*

$$xDSML \triangleq \langle AS, DSA, \text{MoCC}, DSE \rangle.$$

3.3. A Language Workbench to Design and Implement Concurrency-Aware Executable DSMLs

The reification of concurrency for executable metamodeling has been presented in its general form and several implementations of it can be realized. In this section we present the actual implementation of our language workbench that was used to validate the proposition. We have tried to take the most adequate language/technology for each language unit so that the model expressed in the resulting language can actually be executed. Our implementation solution is illustrated by the definition of a concurrent Timed Finite State Machine (TFSM) language; a language where different state machines augmented with timed transitions can be concurrently executed. Here timed transitions possibly refer to different (independent) clocks.

This section is organized according to the implementation choices presented in Figure 3.1. It starts with the description of the AS, then the DSA, followed by the MoCC and to finish, the DSE reification is specified.

3.3.1. Abstract Syntax Design

In model-driven engineering, the abstract syntax is usually expressed in an object-oriented manner. For example the *de facto* standard meta-language EMOF (Essential Meta Object Facility) [137] specified by the Object Management Group (OMG) can be used. EMOF provides the following language constructs for specifying an abstract syntax: package, class, property, multiple inheritance (specialization) and different kinds of associations among classes. The semantics of these core object-oriented constructs is close to a standard object model (*e.g.*, Java, C#, Eiffel).

In practice, we have chosen Ecore to design abstract syntax, a meta-language part of the *Eclipse Modeling Framework* (EMF) [158] and aligned with EMOF. This choice is motivated by the wide acceptance of Ecore and its correspondence to the MOF standard. Additionally, EMF is well tooled and many other tools are based on it (*e.g.*, XText, OCL, GMF, Obeo Designer), so that a language developed in our workbench can benefit from such tools. Note that any meta-language aligned with EMOF can be used in our approach.

Briefly, the AS of TFSM starts with a $System$ composed of a set of $TFSM$s, a set of global $FSMEvent$s and a set of global $FSMClock$s (see Fig. 3.2). Each TFSM is composed of $State$s among which an initial state is identified. Each state can be the source of outgoing guarded $Transition$s. A guard can be specified by the reception of a $FSMEvent$ ($EventGuard$), by a duration relative to the entry time in the incoming state of the transition ($TemporalGuard$) or by a boolean condition ($BooleanGuard$). The duration of a temporal guard is measured on an explicit reference clock. An action is associated with a transition and is represented in the abstract syntax as a *String*. The condition of the boolean guard is also specified as a *String*. These strings represent model level code defined by the designer

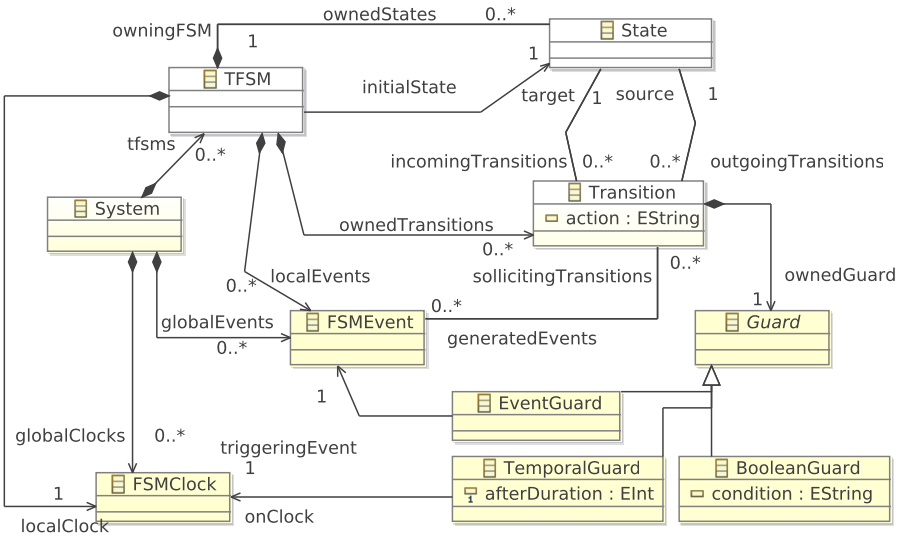

Figure 3.2: Abstract Syntax of TFSM (using Ecore)

(*i.e.*, written using an opaque action language). In our experiments such model level code is written in the Groovy language[5]. Groovy was chosen for its capacity to be dynamically invoked. However any other action language could be used. Finally, a transition can also generate a set of event occurrences when fired.

Note that in the abstract syntax, we refrain from adding concepts about the execution state or functions. These concepts are specified in the DSA.

3.3.2. Domain Specific Actions Design

The domain-specific actions (DSA) enriches the abstract syntax with data representing the execution state and with functions representing the execution functions. Since EMOF (and Ecore) does not include concepts for the definition of the behavioral semantics and OCL is a side-effect free language, we have used the Kermeta language to define the DSA of a DSML. Kermeta is an extension of Ecore that provides an action language used to express the behavioral semantics of a DSL [134]. Using the Kermeta language, an execution function is expressed as methods of the classes of the abstract syntax [134]. The body of the method imperatively describes what is the effect of executing an instance of the concept. The Kermeta language is imperative, statically typed, and includes classical control structures such as blocks, conditionals, loops and exceptions. The Kermeta language also implements traditional object-oriented mechanisms for handling multiple inheritance and generics. Kermeta does not provide any solution to specify the concurrency model. Indeed, the concurrency semantic model could be provided through the Java implicit concurrency model embedded in the underlying execution environment. As a consequence, the designer could use a foreign function interface mechanism to call the Java Thread API but there is no specific support to describe the concurrency model explicitly, in a platform independent manner.

In the approach and the language workbench proposed, the AS and the DSA are conceptually and physically (at the file level) defined in two different modules. The `aspect` keyword enables DSML engineers to bind the AS and the DSA together. It allows DSML engineers to reopen a previously created class in the abstract syntax to add some new pieces of information such as new methods (execution functions) or new properties (execution state representing the semantic domain). It is inspired by open-classes (aka. static introduction) [42].

In the case of TFSM (cf. Listing 3.1), we have added the $currentState$ as an attribute of $TFSM$. We

[5]http://groovy.codehaus.org/

have also added $numberOfTicks$ as an integer attribute of $FSMClock$. All the instances of $TFSM$ in a system possess a current state. Also, all instances of $FSMclock$ have an integer representing their actual time. The execution state of the system is then a set of current states and a set of Integers. The choice of what should be added as attribute depends on the information we want to capture in the execution state of the models. Such information can usually be specified in various ways. For instance, we could have specified the execution state by a set of sensitive transitions instead of a set of current states. Kermeta aspects are also used to specify operations on metaclasses. They provide an operational specification of the execution functions as described in the DSA language unit. The advantage is then the executability of such operations. In TFSM, we have added six operations:

- *initialize()* on $TSFM$: Operation $initialize()$ is used to initialize the execution state of the $TFSM$ (*i.e.*, the current state in our case, lines 6 to 8).
- *fire()* on $Transition$: Operation $fire()$ is in charge of changing the current state from the source state to the target state of the transition. It is also in charge of executing the groovy code specified in the action attribute (lines 14 to 19).
- *init()* on $FSMClock$: Operation $init()$ is used to initialize the *numberOfTicks* (not shown in the listing).
- *ticks()* on $FSMClock$: Operation $ticks()$ is used to increment the *numberOfTicks* of $FSMClock$ (line 26 to 28).

Listing 3.1: part of the Kermeta aspects specifying the DSA

```
1
2 @Aspect(className=TFSM)
3 class TFSMAspect extends NamedElementAspect {
4   public State currentState;
5
6   def public String initialize() {
7     _self.currentState = _self.initialState;
8   }
9 }
10
11 @Aspect(className=Transition)
12 class TransitionAspect extends NamedElementAspect {
13
14   def public void fire() {
15    if(_self.action!= null && !_self.action.empty){
16     GroovyRunner.executeScript(_self.action, _self);
17      }
18    _self.source.owningFSM.currentState = _self.target
19   }
20 }
21
22 @Aspect(className=FSMClock)
23 class FSMClockAspect extends NamedElementAspect {
24   public int numberOfTicks
25
26   def public void ticks() {
27    _self.numberOfTicks = _self.numberOfTicks + 1
28   }
29 }
```

Note that while the DSAs are described by Kermeta aspects over the concepts of the AS, none of them specifies the execution workflow (like a $main()$ operation). The schedule of the different operation calls is made by the concurrency model according to the MoCC used in the DSML.

3.3.3. Model of Concurrency and Communication + DSE Design

The MoCC defines constraints between events while the DSE defines the coordination between the different language units defined in the previous sections. While these two units are conceptually separated, we used a single language (namely MoCCML) to express both. MoCCML is actually split in two parts, a part that specify constraints between events (namely MoCC on Figure 3.3, corresponding to Model of Computation on Figure 3.1) and a part that specify a mapping with other units (namely Mapping on Figure 3.3, corresponding to the DSE of Figure 3.1). After a brief focus on this part of the approach, we focus first on the DSE design and then on the MoCC design.

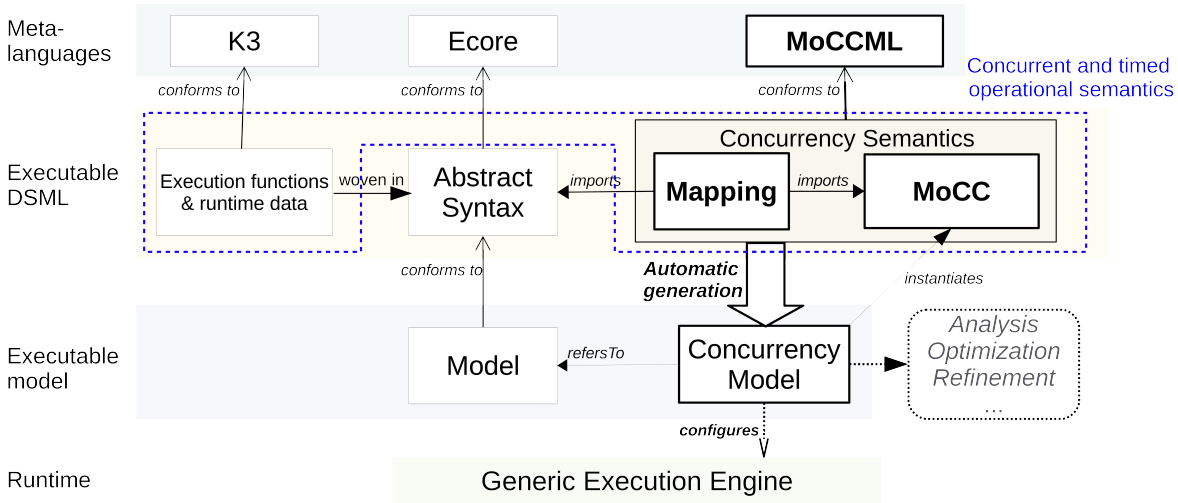

Figure 3.3: Usage of MoCCML for concurrent and timed operational semantics definition

3.3.3.1. Approach Overview

MoCCML is a formal declarative meta-language specifying the concurrency semantics of an executable language. A MoCCML specification is made up with two different models. The first model, named M*o*CC on Figure 3.3, defines a set of constraints between some events. A constraint determines how its event parameters can be scheduled (*i.e.*, ordered) during a specific run of a model. At any moment during a run, an event that does not violate the constraints imposed by a relation can occur. The M*o*CC model is then a library that specifies MoCC specific constraints. These constraints can be both functional and extra functional as long as they result in constraining the (co)evolution of the events in a model. For instance, considering the TFSM language previously introduced, the constraints can define when the guard of a transition must be evaluated or how the clocks from different TFSM evolves (*e.g.*, there are all synchronized, not at all, with a specific drift). They can also define non functional properties, like the worst case execution time between of an action due to the effect of an hardware deployment. The second model, named mapping on Figure 3.3, defines some event types (*i.e.*, DSE), which are directly woven in the abstract syntax of the language. It also defines how the constraints defined in the M*o*CC model must be instantiated for a specific model conforming the language and what are the rewriting rules (*i.e.*, execution functions from the DSA) that must be called. For instance, considering TFSM, it defines the event type *Enter* in the context of a *State* and the event type *Fire* in the context of a *Transition*. Then, it also specifies that *Enter* can only occurs synchronously with the occurrence of *Fire* of one of its *incomingTransitions*.

They are eventually instantiated to define the execution model of a specific model (see Figure 3.3). 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 woven into the context of specific concepts from the abstract syntax of a DSL. 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 3.3).

The mapping defined in MoCCML is based on the notion of event, inspired by ECL [52], an extension of the Object Constraint Language [136]. The separation of the mapping from the MoCC makes the MoCC independent of the DSL so that it can be reused. From such description, for any instance of the abstract syntax it is possible to automatically generate a dedicated concurrency model (see Figure 3.3).

In our approach, this execution model is acting as the configuration of a generic execution engine (see "generic execution engine" in Figure 3.3), which can be used for simulation or analysis of any model conforming to the abstract syntax of the DSL.

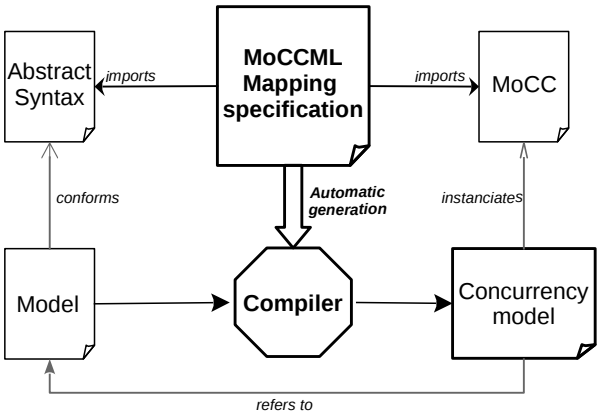

Figure 3.4: Role of the mapping in the generation of the execution model

3.3.3.2. Domain Specific Event Design

In this section, we define the *mapping* part of MoCCML, *i.e.*, the design of the DSE. One goal of the mapping is to specify how the constraints from the MoCC are instantiated for a specific model. Instead of directly writing a transformation, that can be complex and error prone, the mapping is defined between a MoCC and a language. Then, a high order transformation automatically generates a compiler dedicated to the language and the MoCC. The resulting compiler takes a model in input and produces the *concurrency model* in output (see Figure 3.4).

The MoCCML mapping is based on the Event Constraint Language (ECL [52]); itself an extension of the Object Constraint Language (OCL [136]). In short, the MoCCML mapping extends OCL by adding the *Event* Type. It allows defining Event properties in the context of Metaclass. The MoCCML mapping also extends the OCL invariant into behavioral invariant, which specifies how the events previously defined are constrained by the constraints defined in the MoCC.

Background on OCL

OCL is a formal language used to describe expressions on UML or DSL models. Most often, these expressions specify invariant conditions that must hold for the system being modeled or queries over objects described in a model. OCL was introduced by the OMG to formally define constraints that were usually written in natural language. In our context, it contains three important constructions: the metamodel import, the property definitions and the invariant definitions. These notions are quickly explained in the remainder of this subsection. More details can be found in the OMG specification [136].

Since OCL can express constraints on different languages (either UML or DSLs), the first section of an OCL file consists in importing the metamodel(s) (*i.e.*, the AS) on which the constraints are expressed. Based on this import, it is possible to reference packages and metaclasses defined by the metamodel. Listing 3.2 shows an excerpt of an OCL file importing the TFSM metamodel defined in Figure 3.2, allowing to navigate in the package named *tfsm* to the metaclass name *System*.

```
import 'http://org.gemoc.tfsm.model/1.0'
package tfsm
  context System
[...]
```

Listing 3.2: Import of the TFSM metamodel in an OCL specification and navigation in concepts of this metamodel

In OCL, it is possible to define some properties in the context of a metaclass. These properties are constants, they are evaluated after the load of the model and before any constraint or query. They are used to define new properties by using (possibly complex) queries. These new properties are then seen as new properties of the type in which it is defined and can be used in constraints or queries.

```
context TFSM
  def : allEventGuards : Set<EventGuard> = self.ownedTransitions.ownedGuard->select(g|g.oclIsTypeOf(
      EventGuard))
```

Listing 3.3: Definition of a new property defined in the context of the TFSM concept

A property definition contains three main parts.

- The first important part is the *context* of the definition, which defines in which metaclass the property is woven. For instance in Listing 3.3, the metaclass is *TFSM*, meaning that the property will be defined for all the instances of this metaclass.
- The second relevant part is the *type* of the property. OCL being a strongly typed language, the type of the new property must be given. It can be, in OCL, one of the primitive types (Integer, Real, Boolean, String or UnlimitedNatural), one of the type provided by the associated metamodel or a collection. In the example provided in Listing 3.3, the property is a collection of unique *EventGuard*.
- the last important information is the *specification* of the definition. It specifies the constant value of the property. It can be defined by any query whose type is compliant with the one of the property. In the example given in Listing 3.3, the specification returns the set of event guard defined in the associated TFSM.

In OCL an invariant is a Boolean expression that must be true at any time during design or runtime. It is specified in the context of a Metaclass, meaning that any instance of this Metaclass must verify the invariant. It can be arbitrarily complex and, to simplify its writing, it is possible to use sub expressions defining local variables to be used in the Boolean Expression.

```
context System
  inv atLeastOneGeneratedEventForAllEventGuardsV1:
    self.tfsms.allEventGuards.triggeringEvent->forall(
      guardEvent |
      self.tfsms.ownedTransitions.generatedEvents->exists(
      genEvent    |
        guardEvent = genEvent )
      )

  inv atLeastOneGeneratedEventForAllEventGuardsV2:
    let allGuardEvents : set<FSMEvent> = self.tfsms.allEventGuards.triggeringEvent in
    let allGeneratedEvents : set<FSMEvent> = self.tfsms.ownedTransitions.generatedEvents in
    allGuardEvents->forall(guardEvent |
      allGeneratedEvents->exists(genEvent | genEvent = guardEvent))
```

Listing 3.4: Two definitions of a same invariant defined in the context of a SigPML Block

In listing 3.4, two equivalent invariants are defined in the context of a TFSM System. They both specify that for each event guard, there must exist at least one generated event susceptible to trigger the transition. The first one (*atLeastOneGeneratedEventForAllEventGuardsV1*) takes advantages of the property previously defined in listing 3.3. The second invariant uses a sub expression, defined with the *let ... in* keywords. Note that the newly defined variables are local to the invariant in which it is defined. Both styles are functionnaly equivalent. An invariant can be read as "*always, for all* instances of the context Metaclass, the valuation of the Boolean expression must return true".

OCL extensions to allow MoCCML mapping

We reused the previously presented mechanisms from OCL to specify the MoCCML mapping. What was missing in our context was: 1) a conditional way to weave the events of interest in the context of a metaclass and 2) a way to specify behavioral invariants, *i.e.*, invariants specifying liveness properties on the events and whose definition is given in MoCCML.

To extend OCL, we extended its metamodel (see Figure 3.5). First, we defined a *MoCCMLmapping* as a *CompleteOCLDocument* to which we add the possibility to import MoCCML files (by using a *MoCCImportStatement*). By using such imports we have access to the declarations and definitions given in the MoCCML specification. Then we added the *EventType* as an OCL Type (*i.e.*, a *TypeRef*) to be able

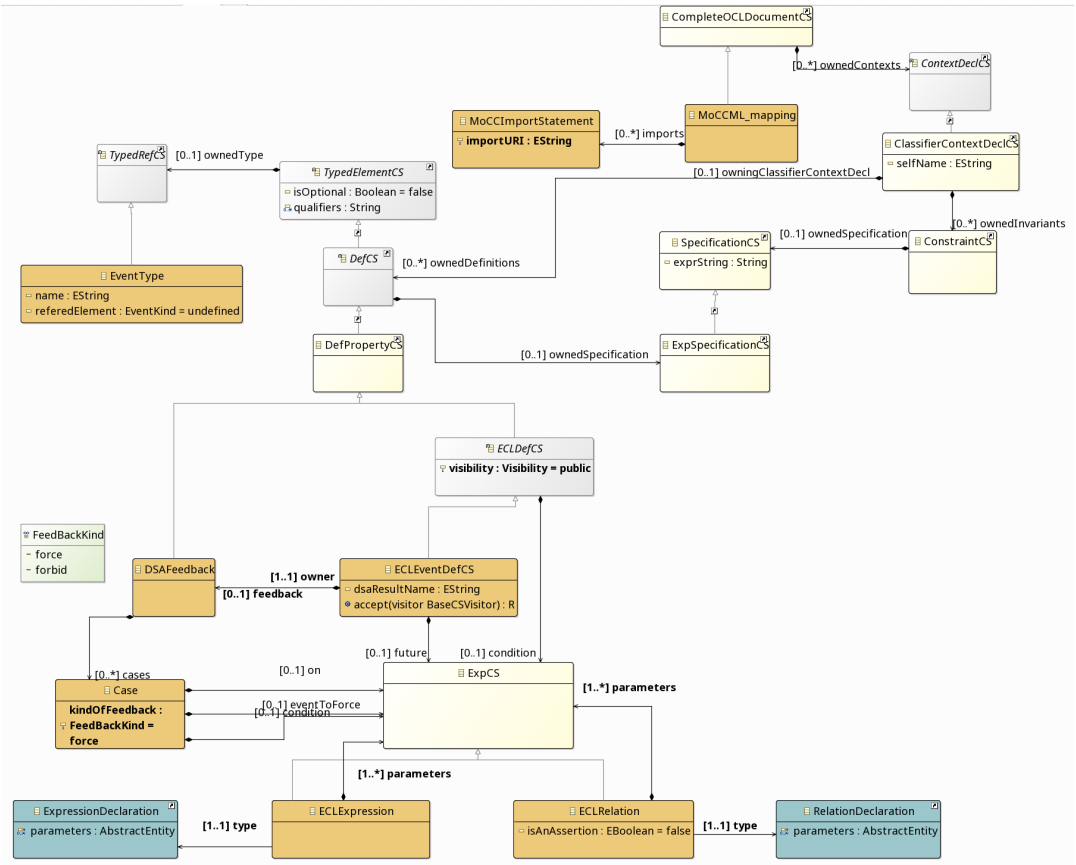

Figure 3.5: Excerpt of the MoCCML mapping metamodel

to type correspondingly the properties to weave in the metaclasses or the properties used in the *let..in* clauses of invariants. It is required to condition the definition of events in specific metaclasses[6], consequently we extended the *DefCS* concept from OCL as a *ECLDefCS* where a condition is added. The condition is specified by a Boolean expression (*ExpCS*), providing the possibility to query the model in order to define the condition. To specify the behavioral invariants, we introduced two new kinds of OCL Expressions: *Relation* and *Expression*. Both *Relation*s and *Expression*s are typed by a MoCCML declaration and have a sequence of parameters, each of them being given by an OCL expression. *Expression*s can be used in any Event property definition and *Relation*s can be used in invariants to restrict the coevolution of the specific events given as parameter.

Finally, it is required to implement the protocol resulting from the DSA to DSE communication (see language unit #4 in Section 3.2). we defined a feedback mechanism (*DSAFeedBack*). The feedback mechanism is defined by a set of *Case*s, themselves defined by a condition on the result of the DSA to specify when the feedback must be applied, an event and the kind of feedback (*FeedBackKind)* amongst *force* and *forbid*. It specify if the event must be forced to occur or forbid to occur. The reader can notice that, to avoid relying on the simulator step to force or forbid an event, we added the *on* expression to specify when the event must be forced or until when it must be forbid. More details about the DSE are provided in [114].

All together, these mechanisms are used to define the DSE, that link the MoCC and the DSA together to constitute the behavioral semantics of the DSML. They contain the events relevant to the DSML perspective and how they are linked to the execution functions of the DSA; and on the other hand they specify queries on the AS to specify the actual parameters that have to be used by the concurrency model on a specific model. The DSE also specify, if needed, the execution function that must be called when specific events occur as well as the feedback mechanism.

[6]This is for instance the case when using a profile on top of UML, where the definition of an event in the context of a metaclass can be conditioned by the use of a specific stereotype.

As an illustration of the use of these concepts, we equip a part of the AS depicted in Figure 3.6 with DSE and we show the instantiation of these DSE on a specific TFSM model. 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 3.6). These DSE are defined by using the MoCCML mapping. A partial MoCCML mapping specification of TFSM is shown in Listing 3.5 where the DSE *entering* and *leaving* are defined in the context of State (Listing 3.5: line 3 to 5); *occurs* is defined in the context of FSMEvent (Listing 3.5: line 6 and 7), *fire* is defined in the context of a Transition (Listing 3.5: line 8 and 9) and *ticks* is defined in the context of an FSMClock (Listing 3.5: line 10 and 11). 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 (Model Specific Event). 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.

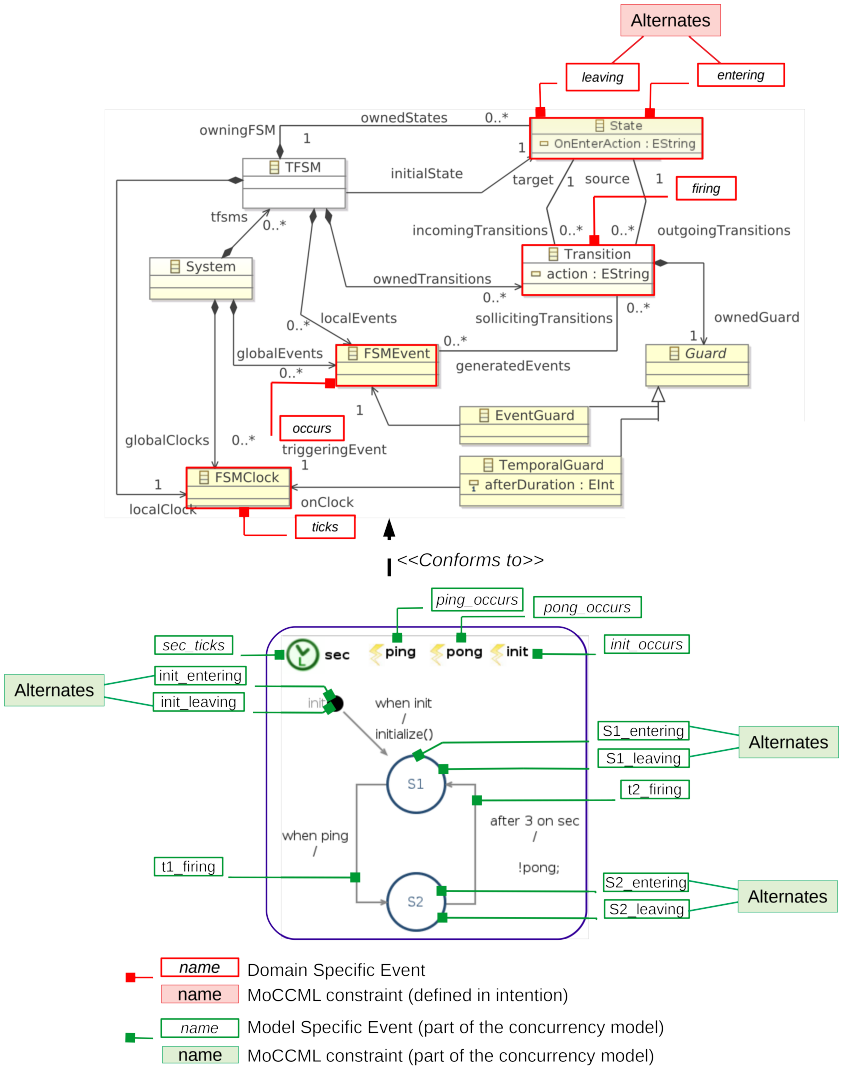

Figure 3.6: (At the top) The TFSM metamodel with its language behavioral interface. (At the bottom) a TFSM model with its model behavioral interface.

```
import 'http://fr.inria.aoste.gemoc.example.tfsm'
ECLImport "TFSMMoC.ccslLib"

package tfsm

 context State
   def : entering : Event = self
   def : leaving : Event = self
```

```
context FSMEvent
  def: occurs : Event = self
context Transition
  def: fire : Event = self.fire()
context FSMClock
  def: ticks : Event = self.ticks()
```

Listing 3.5: Partial DSE definition of TFSM (part 1: declaration of the DSE)

The DSE are optionally associated to rewriting rules defined in the DSA. For instance line 12 of listing 3.5, the *fire* DSE is associated to the *fire* function, defined in the DSA (see listing 3.1). Each time an instance of the corresponding MSE will occur, it will call the fire method on its associated EObject. Note that the MoCCML mapping listing imports a MoCC library (line 2 in Listing 3.5). It is used to define some invariants (in green in listing 3.6), which specify in which context and with which parameter(s) a constraint from the MoCC is used. The specification of the actual parameters are specified by querying the AS. For instance, line 3 of listing 3.6 the *Alternates* constraint is applied on the enter and leave DSE. To specify the mapping between MoCC and DSA, it is also possible to create intermediate events by using expressions over existing DSE. For instance, lines 5 to 17, the invariant specifies that for each transition of the AS whose guard is of type *TemporalGuard* (line 7), and whose source state of this transition has more than one other outgoing transitions (line 8), then a constraint of type *TemporalTransition* must be instanciated in the concurrency model (line 13). The parameters of the constraints are query from the AS like for instance line 14. It can also be specified by an expression over existing domain specific events like specified in line 11, which specifies a new event defined by the *Union* of all the fire events from other outgoing transitions from the same source state. It is used here to specify when the event transition must be disabled (see the formal parameters line 1 in Listing 3.7). These queries define how the structure of the AS is used to retrieve the actual parameters. For instance, the actual duration of the temporal transition is defined by the *afterDuration* attribute defined in the AS (line 10). Note that in listings 3.5 and 3.6, only the keywords in red are extensions introduced by MoCCML mapping. All other constructs are pure OCL, that are using constraint defined in MoCCML (imported line 2 of listing 3.5 and shown in green in listing 3.6).

```
context State
  inv EnterThenLeave:
    Relation Alternates(self.enter, self.leave)

context Transition
  inv fireWhenTemporalGuardHoldsVariousTransition:
    (self.ownedGuard.oclIsKindOf(TemporalGuard)
    and self.source.outgoingTransitions-> select(t|t <> self)->size() > 0)
    implies
      let guardDelay : Integer = self.ownedGuard.oclAsType(TemporalGuard).afterDuration in
      let otherFireFromTheSameState: Event = Expression Union (self.source.outgoingTransitions
                                                -> select(t|t <> self).fire) in
      Relation TemporalTransition(self.source.entering,
                                  self.ownedGuard.oclAsType(TemporalGuard).onClock.ticks,
                                  otherFireFromTheSameState,
                                  guardDelay,
                                  self.fire )

context FSMEvent
  inv occursWhenSolicitate:
    (self.sollicitingTransitions->size() >0)
    implies
      let AllTriggeringOccurrences : Event = Expression Union(self.sollicitingTransitions.fire) in
      Relation FSMEventRendezVous(AllTriggeringOccurrences, self.occurs)
```

Listing 3.6: Partial DSE definition of TFSM (part 2: use of constraints)

Listing 3.6 shows another invariant, which defines the *FSMEventRendezVous* constraint on the MoCC. This constraint is changed in section 3.5 to highlight the impact of a MoCC variation. From such a specification, it is possible to generate a CCSL specification that represents the concurrency model for any model that conforms to the AS; *i.e.*, a model that contains the actual constraints and their parameters according to a specific model[7].

[7] the concurrency model is actually the equivalent of the concurrent and timed aspects of the behavior model from Chapter 2

3.3.3.3. Definition of MoCCML Constraints

MoCCML mapping defines how the constraints are mapped on the AS of a DSML and the DSA. However, the meta-language used for the specification of the MoCC constraints must specify the constraints on events independently of the AS and the DSA on which it is applied.

We exploited the library mechanism of the Clock Constraint Specification Language (CCSL) [127] for specifying the MoCC (at the DSML level). We consistently chose CCSL specifications to represent the instances of the MoCC as concurrency models (at the model level). In CCSL, a concurrency model is a set of constraints whose definitions and formal parameters are given in libraries. In this context, a CCSL library specifies MoCC specific constraints. These constraints specify the correct evolution of the events given as formal parameters of the constraints. More precisely it is a reusable set of constraints considered as consistent with regards to a specific MoCC; it defines the possibly timed synchronizations and causality relationships between some events and such constraints have already shown to be a good candidate for the specification of the concurrent and temporal aspects of a language (see section 2.5). It is not possible to specify any data manipulation aspects in CCSL so that it fits with the separation of concern between the concurrent and temporal aspects in the MoCC and the computational aspects in the DSA.

For the TFSM example, we defined all the constraints dedicated to the TFSM MoCC in a specific library. From the external point of view, only the declarations of the constraints are exhibited. For instance we have defined *TemporalTransition* and *EventTransition* constraints whose declarations are presented in Listing 3.7. Each declaration exposes a set of formal parameters, which are needed to specify the constraint between the events (named *clock*s in CCSL). For instance, for the temporal transition relationship, four events are important, the event that starts the "timer", the event used to measure the time, the event that disables the transition (*i.e.*, makes it non fireable until the next timer starts), and the clock that actually fires the transition. Additionally, the integer representing the delay after which the transition should be fired is also a parameter. Such parameters represent the information that should be provided by the MoCCML mapping so that the MoCC can be used. Such declarations do not make any assumptions about AS and DSA.

Listing 3.7: Excerpt of a MoC library used for TFSM (using CCSL)

```
1 RelationDeclaration TemporalTransition(TemporalTransition_MakeFireable:clock,
      TemporalTransition_RefClock:clock, TemporalTransition_Reset:clock,
      TemporalTransition_delay:int, TemporalTransition_Fire:clock)
2 RelationDeclaration EventTransition(EventTransition_MakeFireable:clock,
      EventTransition_Trigger:clock, EventTransition_Reset:clock, EventTransition_Fire:
      clock)
```

During the various experiments we realized, CCSL was expressive enough to specify the MoCC constraints. However, it was difficult to use it to specify state based constraints, *i.e.*, constraints that naturally behave differently according to their history. In order to ease this task, we extended CCSL with a state based representation of constraints. This extension is fully described both syntactically and semantically in a research report [57]. In this section, we quickly describe a specific constraint and overview the formal operational semantics of MoCCML. The reader should note that the state based definition of a constraint (a MoCCML one) is based on a CCSL declaration. There is consequently no difference in the use of a constraint defined in CCSL or in MoCCML.

Figure 3.7 represents the definition of the *temporalTransition* MoCCML constraint used in Listing 3.6. The transition between states can be guarded by a set of events and a predicate based on internal variables and parameters. In a specific state, the events that are part of the parameters but not referenced by an outgoing transition are forbidden. For instance in Figure 3.7, there are two transitions from and to the *waitingStateEntry* state. This is a mean to signify that the associated events are not constrained in this state, *i.e.*, they can then occur freely, but not synchronously since there is no transition with both events. Also, a transition can have an effect, like for instance incrementing an internal value (see transition on the right of Figure 3.7 for an example).

The operational semantics of MoCCML constraints is defined in the next paragraphs.

Operational Semantics of MoCCML Constraints The MoCCML language is extending the operational semantics of CCSL from [6] with the notion of death and birth and is consistently defined in [57].

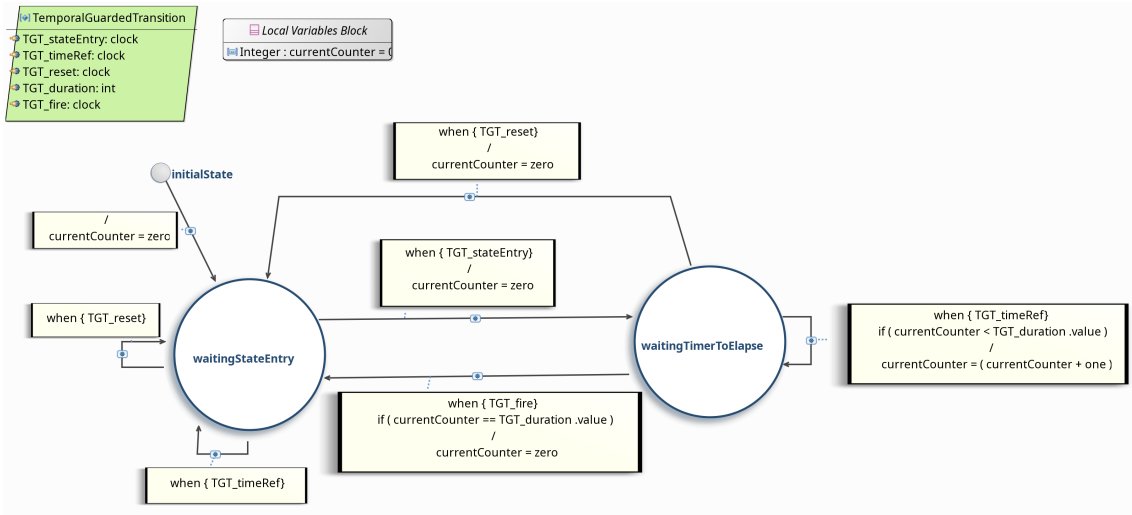

Figure 3.7: The temporalTransition MoCCML constraint in the actual MoCCML editor

In the following paragraph, we only overview the operational semantics of the state based constraints. Roughly speaking, the semantics of a constraint automaton is a set of boolean equation over the clocks. In this sense, a constraint automaton is a specific kind of relation and integrates naturally with the other constraints written in CCSL.

MoCCML automaton relations Before to specify the Structural Operational Semantics of the constraint automaton, we provide the syntactic definitions used in the remainder of the section. A MoCCML automaton R is a tuple $< S, i, S_t, C, Var, Pre, Eff, R >$, where:

- S is a set of states;
- $s_0 \in S$ is the initial state;
- $S_t \in S$ is a set of terminal states;
- C is a set of clocks, known as the alphabet of the relation;
- Var is a set of variables assuming values from a finite domain (denoted D);
- Pre is a set of preconditions on the valuation of the variables;
- Eff is a set of effects updating the value of one or more variables;
- $R \subset S \times Pre \times 2^C \times Eff \times S$, defines an automaton as a subset of possible transition relations, where a transition relation is guarded by a precondition, constrained by a set of clocks and applies a set of effects when fired.

Let $\mathcal{V}^{var} \subset D_1 \times \ldots \times D_n$ be the set of all possible valuations of the variables in the model. In the following we denote one of such valuations with $v = \langle v_1, \ldots, v_n \rangle$, where $v_i \in D_i$. The set of all possible execution configurations of a relation is $\mathscr{S}^{exe} \subset S \times D_1 \times \ldots \times D_n$. Simply put, the configuration of one relation R during execution is given by extending the variable valuation with the current state. In the following such a configuration will be denoted as $\langle s, \langle v_1, \ldots, v_n \rangle \rangle_R$ or simply $\langle s, v \rangle_R$. The initial configuration of a relation $\langle s_0, v_0 \rangle_R$ associates to the initial state of the relation the initial valuation of the variables.

Let $\mathcal{V}^{\mathscr{C}}$ be the set of all possible clock valuations, where a clock valuation is defined as $\alpha^C = \{b | b = true$ iff clock c ticks and $false$ otherwise, $\forall c \in C\}$. We say that a valuation $\alpha^C \in \mathcal{V}^{\mathscr{C}}$ satisfies a formula ϕ, written $\alpha^C \models \phi$, iff ϕ evaluates to $true$ after replacing every clock x occuring in ϕ by $\alpha^C[x]$.

$$\phi_t^R = \left[\left(\bigwedge_{c \in C^t} c \right) \vee \left(\bigwedge_{c \in C^t} \neg c \right) \right] \wedge \left(\bigwedge_{c \in (C^R \setminus C^t)} \neg c \right) \tag{3.1}$$

The Equation 3.1 represents the boolean clock contraint contributed by a transition t of a relation R. The clocks guarding the transition can tick (the first term) or not (the second term). The clocks in the vocabulary of the relation that are not present on the transition $(C^R \setminus C^t)$ are blocked (the last term).

$$enabled(\langle s, v\rangle_R) = \left\{ s \xrightarrow{[p]c\{e\}} t \middle| s \xrightarrow{[p]c\{e\}} t \in R \wedge v \models p \right\} \tag{3.2}$$

Given a relation R in the configuration $\langle s, v\rangle$, the Equation 3.2 defines the enabled set, which is the set of transitions from the state s with the precondition satisfied by the variable valuation ($v \models p$).

$$\phi^R_{\langle s,v\rangle} = \left(\bigvee_{t \in enabled(\langle s,v\rangle_R)} \phi^R_t \right) \tag{3.3}$$

The Equation 3.3 defines the clock contraint contributed by the relation R in the configuration $\langle s, v\rangle$.

$$fireable(\langle s, v\rangle_R, \alpha^C) = \left\{ s \xrightarrow{[p]c\{e\}} t \middle| s \xrightarrow{[p]c\{e\}} t \in R \wedge v \models p \wedge \alpha^C \models \bigwedge_{c \in c} c \right\} \tag{3.4}$$

The set of fireable transitions of a relation R in the configuration $\langle s, v\rangle$, allowed by the clock valuation α^C, is given in Equation 3.4 that selects amongst the enabled transitions the ones whose set of clock satisfies the clock valuation.

$$\frac{\langle s, v\rangle_R \wedge s \xrightarrow{[p]c\{e\}} t \in fireable(\langle s, v\rangle_R, \alpha^C)}{\langle s, v\rangle \wedge C \xrightarrow{[\![e]\!]} \langle t, v'\rangle \wedge C'} \qquad (transitionevaluation) \tag{3.5}$$

Finally, each of the fireable transition implies a possible change in the valuation of the variables and in the set of clocks according to the semantics of the effect of the transition (see the rewriting rule 3.5).

the remainder of the section defines the semantics of precondions and effects. The abstract syntax:

$$\begin{array}{rll}
k \in \text{Con} & & //Constants \\
x \in \text{Ide} & & //Variableidentifiers \\
e \in \text{Exp} & & //Expressions \\
a \in \text{Eff} & & //Effects \\
e ::= & k & \\
 & \mid x & \\
 & \mid \text{unop } e & \\
 & \mid e_1 \text{ binop } e_2 & \\
a ::= & x := e & \\
 & \mid \text{birth}(c) & \\
 & \mid \text{kill}(c) & \\
 & \mid a_1 \,;\, a_2 &
\end{array}$$

The semantics:

$$\frac{(\mathcal{V}, C)}{\mathcal{V}[x \mapsto [\![e]\!](\mathcal{V}, C)]} \qquad ([\![x := e]\!]) \tag{3.6}$$

$$\frac{(\mathcal{V}, C)}{\mathcal{I}_c = c^{\uparrow}} \qquad ([\![\text{birth}(c)]\!]) \tag{3.7}$$

$$\frac{(\mathscr{V}, C)}{\mathscr{I}_c = \mathscr{I}_c + c^{\downarrow}} \qquad ([\![\text{kill}(c)]\!]) \tag{3.8}$$

$$\frac{(\mathscr{V}, C)}{([\![a_2]\!] \circ [\![a_1]\!])(\mathscr{V}, C)} \qquad ([\![a_1; a_2]\!]) \tag{3.9}$$

We used such constraints to define several different languages (see Section 3.5). The declarative nature of the constraint helped a lot to exhibit refinement of the applicative concurrency when deployed on a specific execution platform model. But before to enumerate some examples on which we applied the constraints, we describe in the next section, the integration of the approach in the GEMOC Studio.

3.4. Integration of the approach in the GEMOC Studio

3.4.1. Overview of the GEMOC Studio

The design of the GEMOC studio has been thought so that several different execution engine (making explicit the concurrency and temporal aspects or not) can be integrated. The main goal was to provide a generic set of execution and debugging services to be used by different execution engines. In addition, the GEMOC studio has been used to implement the proposition detailed on section 4.4 about the coordination of heterogeneous languages. In the next section, we focus on the execution and debugging of models conforming concurrent and timed languages.

The GEMOC Studio is an Eclipse package atop the Eclipse Modeling Framework (EMF), which includes:

- The *GEMOC Language Workbench*: to be used by language designers to build and compose new xDSMLs,
- The *GEMOC Modeling Workbench*: to be used by domain designers to create and execute models conforming to xDSMLs.

The different ingredient of a DSML, as defined in the previous sections with the tools of the language workbench, are automatically deployed into the modeling workbench. They parametrize a generic execution framework that provides various generic services, such as graphical animation, debugging tools, trace and event managers, timeline visualizations, etc.

3.4.2. Overview of the Execution Framework

Figure 3.8 shows an overview of the execution framework offered by the GEMOC studio, far beyond the "only" execution of models conforming xDSMLs. This was made possible by the enthusiastic contribution of different people, both technically and scientifically (see `http://eclipse.org/gemoc/committers` and `http://gemoc.org/ins/partners.html` for a list of some of them). We focus in the remainder of this section on the concurrent and timed execution. Other details are provided in the original paper [31]. Note that for this paper, all the artifacts allowing the different experimentation were reviewed, making the different features not only a wish list but a set of actually implemented artifacts. At the middle, the xDSML defined in the language workbench is depicted. It is composed of abstract and concrete syntaxes, and of operational semantics. For a given xDSML, the operational semantics are defined using a specific metaprogramming approach (in our case, the one depicted in the previous sections). Since the GEMOC Studio is based on EMF, Ecore is used to define the abstract syntax, and at runtime the executed model is a set of EMF objects. For defining the concrete syntax, the Sirius toolkit is used. For more information and examples on the language workbench, please refer to the official documentation[8].

[8] `http://gemoc.github.io/gemoc-studio/publish/guide/html_single/Guide.html`

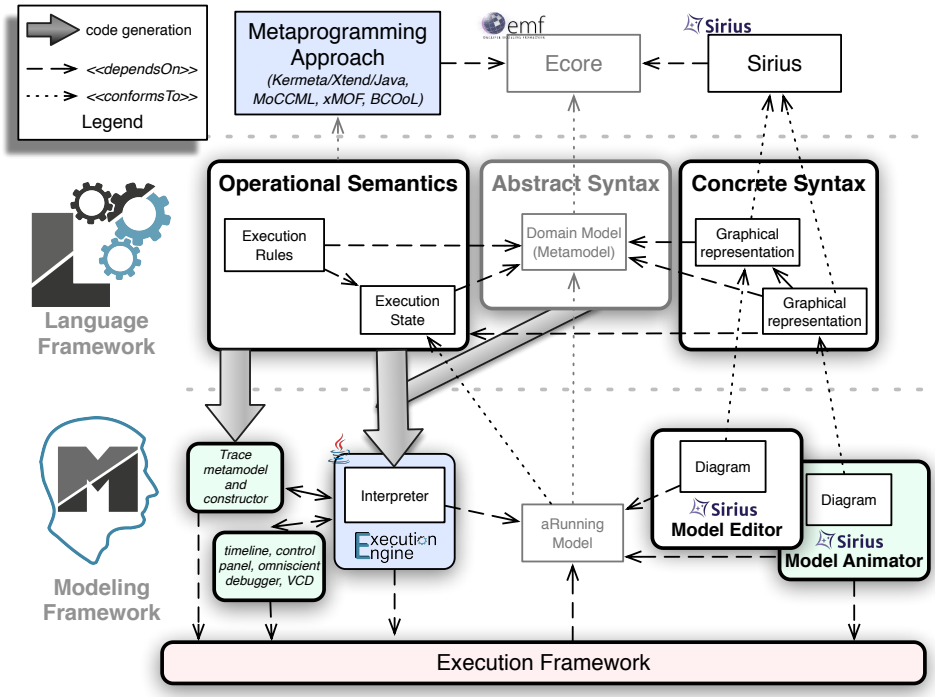

Figure 3.8: Overview of the GEMOC Execution Framework (taken from [31])

At the bottom, the modeling workbench supported by the *execution framework* is shown. This workbench allows the user to define an executable model conforming to an xDSML, and execute it using an execution engine and a selection of addons. An *addon* is developed using the execution framework and provides a set of runtime services (*e.g.*, animation or debugging). A specific *execution engine* that follows the approach presented earlier has been developed using the execution framework. It is responsible for integrating the interpreter of the considered xDSML – developped using the same metaprogramming approach as the engine – with the addons provided in the studio. This implies sending notifications to addons regarding the execution (*e.g.*, beginning of the run, start or end of a rule, etc.). By reacting to notifications, an addon may query the engine to ask for information, which can be used to provide a view that gets updated during the execution, or control the execution of the model, or even modify the model. Some addons are generated (*e.g.*, trace management), while others are generic and compatible with any engine. At runtime, the executed model contains a dynamic execution state that is modified by the interpreter and by addons.

3.4.3. Interface of the Execution Framework

The integration of an engine in the framework requires that notifications are sent to the attached addons during the execution (*i.e.*, during the *execute* operation). For this purpose, the framework defines the common notion of *execution step*, a step being the application of one or several concurrent execution rule(s) of the operational semantics. When an engine is about to apply such rules, it must create a step object containing information about the executed rule (identifiers of the rule, parameters given, etc.). Then this step object must be used to notify addons of both the start and the end of a step. Step objects are also used in the framework for other purposes, such as the storage of a stack of the steps currently being executed, or to be directly stored within an execution trace.

Since a significant part of the logic is common to all execution engines, the framework provides a basic *abstract execution engine* that can be extended into a concrete engine. This abstract engine implements part of the API described above, such as the services to manage the status, to add or remove addons, and to start or stop. In addition, this abstract engine provides internal services to notify all addons of the progress of the execution, although the task of calling these services at the right instants is left to the concrete engines. In particular, a service called *beforeStep* is provided to be called at the beginning

of a step, and second called *afterStep* must be called at the end. Depending on the technique used to define the execution semantics, the integration of these operations in the operational semantics can be done in different ways (instrumentation, event listeners, etc.).

Addons are components that can be defined to provide runtime services. Such services need to be connected to the ongoing execution of a model in order to follow the execution and to extract information (*e.g.*, the content of the execution state) or to control the execution (*e.g.*, pause or provide input). To that effect, the GEMOC execution framework provides an API that defines an addon as a component with at least four operations that are synchronously called by the engine during the execution of a model: *engineStarted* when the execution starts, *engineStopped* when the execution ends, *aboutToExecuteStep* when the engine is about to start an execution step, *stepExecuted* when an execution step finished. Within the implementation of one of these operations, an addon can access the engine and its status, the executed model, or even the graphical interface of the studio. Therewith, an addon can accomplish a large diversity of tasks, such as changing the executed model, pausing or stopping the execution, displaying information, or sending some input data to the engine.

These artifacts and methods have initially been influenced by the work realized in TimeSquare (mainly on the trace and backend aspects), making the first implementation quite straight forwards. However, it evolved a lot during the live of the GEMOC studio to introduce more possibility and flexibility in the studio, like for instance the use of multi-dimensional traces as defined in [30] or different way to control/choose the executions steps.

3.4.4. The concurrent and timed Execution Engines

The concurrent and timed execution engine relies is dedicated to operational semantics where the (possibly concurrent and timed) control is described in the MoCCML formal language (see section 3.3.3, [58]), while the execution rules are written in any Java based language (e.g., Kermeta [134] or ALE [9]. In this case, the *initialize* operation creates a so called concurrency model according to the operational semantics, where some relevant events are defined, constrained together and linked to execution rules defined in Java (or any Java-based language). Then the *execute* operation consists in a loop that asks for the next possible steps to the Timesquare solver [51]. There are potentially several possible steps since it considers all acceptable interleavings of the events. An addon allows the user to manually select the next desired step or to ask the solver to do it according to a specific policy (see section 2.4) Once a step is selected, it calls the corresponding Java methods as required. Notifications to and from addons are received and sent directly by the engine during this execution loop. Note that the execution can be a bit more complex due to the possible feedback, that prune some of the path from the execution model according to results from the execution rules (see section3.2.3).

3.4.5. Runtime Services linked to the concurrent engine

Using the API for addons that we described, we implemented a set of generic and specific runtime services that can be either shared among the different execution engines of the GEMOC Studio or used only for a specific engine.

Debug and Graphical Animator. The graphical animator is an addon that updates different views in order to display the current execution state of the executed model, hence helping to understand models under execution. Since the GEMOC Studio is based on Eclipse, the animator is connected to the Eclipse debug UI to display the stack of currently executed steps in the *Debug* view, and the values of the execution state in the *Variables* view. In addition, if a graphical concrete syntax was defined for the xDSML using Sirius, the animator updates a graphical representation of the execution state of the model during the execution.

Multi-Branches Execution Trace Addon. The *concurrency specific multi-branches trace addon* captures traces that contain different branches, each linked to a choice made during a non-deterministic

[9] `http://gemoc.org/ale-lang/`

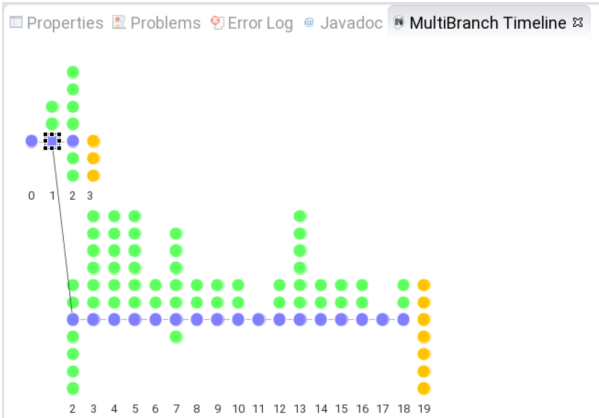

Figure 3.9: Screenshot of the omniscient debugger with two branches.

situation of the execution. This addon is used by the multibranch omniscient debugger to perform back stepping and therefore to create new branches in a trace as defined in the next paragraph.

Omniscient Debuggers. The multi branch trace addon is also a support for omniscient debugging. They can provide services expected by any debugger (*e.g.*, step into/over, breakpoints) thanks to an integration with the Eclipse Debug UI. In addition, they provide services to explore the execution *backward* in time (back), by relying on the execution traces constructed by the trace addon. It relies a view called a *MultiBranch Timeline*, which is an interactive graphical representation of the trace. It enables the exploration of different execution traces. It allows the creation of new branches by going back in time and making different choices. For instance in Figure 3.9, we have two execution traces. The one on top was followed to do 4 execution steps, then the user went back in time at step 1 and made a different choice (exploring potential interleaving of actions due to concurrency). Blue circles are steps that have been executed, green circles are potential execution step that have not been explored and yello steps are future execution steps to explore.

VCD. The VCD addon provides a representation of the execution in the form of a timing diagram represented in the Value Change Dump format (VCD) defined by the IEEE Standard 1364-1995 and extended in the IEEE Standard 1364-2001. It represents the potentially parallel evolution of the calls of the execution steps. This illustrates the possibility to adapt the GEMOC studio addons to specific domains In this case it is adapted to the Electronic Design Automation (EDA) domain, which is used to this kind of format.

Here we reused directly the TimeSquare back-end and adapted it to the GEMOC addon mechanism.

Stimuli Manager. The stimuli manager is an addon provided to send *stimuli* to an ongoing execution. This addon provides a view showing all the possible stimuli (in our specific case events) that can be sent to the execution, This addon is interesting for concurrent executions that may depends on external stimuli, *e.g.*, to simulate/record/replay stimuli from the external environment.

Step Decider. The step decider is an addon provided to make choices in concurrent situations during an ongoing execution. It provides a view that shows all the steps possible at a given point in time, and gives the possibility to rely either on an existing decider (*e.g.*, the random decider to make a choice at random), or to manually choose the next step from the displayed possibilities. Like the stimuli manager, this addon is only interesting for concurrent semantics with different possible steps at some points in the execution.

Papyrus Animator. Since GEMOC relies on EMF, it can be used to develop executable models based on UML like models (like for instance in [22]). To ease debugging and provide feedback directly on the papyrus model, we adapted the papyrus animator from TimeSquare to the GEMOC addon mechanism.

This is close to the Sirius Animator described above but without the interesting possibility to customize the application.

3.5. Validation and Discussion

In order to validate our approach we built several languages in the last years. All the languages have not always been developed in their totality since some of them are very big and our goal was only to ensure the applicability of the approach.

here is a non exhaustive list of the languages that have been developed in the last 5 years in the GEMOC Studio according to the approach presented below. Note that we reused some of the Abstract and Concrete Syntaxes (*e.g.*, in Capella and UML).

ArduinoML is a small DSL to develop both the software and the hardware configuration (with sensors and actuators) of Arduino boards. It has been used also in education at the polytech'Nice engineer school. (here is a video of the model of three boards communicating through bluetooth: `https://youtu.be/dtJZyK1RM2A`)

UML as developed in the EMF framework

- Sequence Diagram
- State Charts
- Activity diagram (model execution challenge winner [47], a partial video explaining the concurrent aspects is available here: `https://youtu.be/ko53lL2fkUA`)

Capella (partially) is an open source System Engineering language developed initially by Thales and now supported by polarsys (`http://polarsys.org/capella`). In our experiments, beyond execution and debugging, it was the support for the injection and verification of execution traces provided by an external tool. A video explaining this process is available here: `https://youtu.be/ESIX2PFGiDU`)

- Temporal scenario, functional architecture and deployment
- Mode automata and functional architecture

temporal and modal Message Sequence Chart is a specification language based on model MSC where time and deployment on an hardware architecture has been conducted to understand the impact of the execution platform on the timing analysis (this work was realized in collaboration with Jörg Holtmann and Ruslan Bernijazov from Fraunhofer IEM, Paderborn, Germany. See [22] for details).

MuVArch (developed during the PhD of Ameni Khecharem) is a multiview architecture in which each view is structured into different concerns. The approach was mainly used to make executable the controller view, used to schedule the different manager (*e.g.*, a power manager or a thermal manager)

SMCube is a formal state charts language used in embedded systems (it is more or less equivalent to state flow from Mathworks in the XCos tool suite, `http://www.evidence.eu.com/products/smcube.html`). This is a pretty complex language, entirely developped with support of model level code written in the Groovy language.

Communicating FSM is a simple language used for the tutorial at models 2017: `https://github.com/gemoc/MODELS2017Tutorial`.

TFSM is an illustrating language based on state machine with the addition of polychronous time. It has been introduced in [44] and is now one of the GEMOC official sample for the concurrent engine. The first working integration of the approach presented below can be viewed on a complementary video: `https://youtu.be/gT1QUlmFkLM`. This allows to appreciate the improvements done in the tooling over the last 5 years.

ComponentModel and TFSM is the used of the previous TFSM in a component architecture where the state machine communicate either ni a synchronous or asynchronous way through port and connectors. It has been presented at the ECNU summer school in Shanghai.

Bus Analyzer is a prototype language developed to analyze the bus occupancy according to communicating slaves and masters abstracted according to their production/consumption of data. Simulation results are plot in graph during the simulation.

MarkedGraph is a conflict free homogeneous PetriNet used in the documentation of the GEMOC Studio

SigPML is an extension of SDF with the hardware platform and allocation of agents on it. It also embed the code of agents in the form of Groovy code. It is the second official example for the concurrent engine.

AADL (partially), a modeling language for safety critical systems (`http://www.aadl.info`). We covered here only the deployment of communicating Threads on Processors.

Communicating real time tasks is a small test to illustrate how we can express classical real time problems and analyze them.

EAST-ADL (partially) an architecture language from automotive industry. We covered here only the "functional analysis architecture" package

TADL2 (in collaboration with Arda Goknil) is the Timed Augmented Description Language developed in the Timmo2use project. It should have help dealing with multiform time and symbolic timing expression in automotive.

Nodeus is a language for the development of non intrusive silver age health monitoring. It has been developed in the context of a (now aborted) transfer of technology between Inria and a small company.

It is important to notice that, when using the GEMOC studio to define the concurrent operational semantics of such languages, they all benefit from model animation, debug facilities, omniscient debugging and state space construction possibility.

All these examples allowed us to be confident in the applicability of the approach. However, we should admit that we tend to focus mostly on modeling, architecture languages and we did not pay enough attention to more classical an complex programming languages to ensure the correct expressiveness of the approach.

Nevertheless, we were able to express the concurrent and timed operational semantics of various different languages. When done, the concurrency model represent the maximum parallelism of the application. We also shown on examples like SigPML or AADL that we were able to take into account constraints from the execution platform, whose effects are mainly to restrict the application parallelism and add/fix delays between communications. However, we did not try to specify very low effects of the architecture deployment, yet.

3.6. Related Work

Much work has been done on the design and implementation of both DSML and models of computation. In this section, we propose a conceptual and technical framework to take benefits from both underlying theories. This section presents some of the related work in the field of language design and implementation, and then in the field of models of computation.

The problem of the modular design of languages has been explored by several authors (*e.g.*, [65, 172]). For example, JastAdd [65] combines traditional use of higher order attribute grammars with object-orientation and simple aspect-orientation (static introductions) to get a better modularity mechanism. With a similar support for object-orientation and static introductions, Kermeta and its aspect paradigm can be seen as an analogue of JastAdd in the DSML world. The major drawback of such approach is that none of them provides a native support for concurrency.

A language workbench is a software package for designing software languages [169]. For instance, it may encompass parser generators, specialized editors, DSLs for expressing the semantics and others. Early language workbenches include Centaur [26], ASF+SDF [107], and TXL [49]. Among more recent proposals, we can cite Generic Model Environment (GME) [159], Metacase's MetaEdit+ [162], Microsoft's DSL Tools [48], Krahn et al's Monticore [111], Kats and Visser's Spoofax [100], Jetbrain's MPS

[168] or de Lara et al. Atom^3 [50, 128]. The important difference of our approach is that we explicitly reify the concurrency concern in the design of an executable language, providing a dedicated tooling for its implementation and reuse. Our approach is also 100% compatible with all EMF-based tools (at the code level, not only at the abstract syntax level provided by Ecore), hence designing a DSL with our approach easily allows reusing the rich ecosystem of Eclipse/EMF.

Models of computation, and in particular the concurrency concern, have been mainly tooled in three different workbench: Ptolemy [64], ModHel'X [87] and ForSyDe [153]. Each of them have their own pros and cons but they are all based on a specific abstract syntax and API. On one hand the unique abstract syntax avoids their use in the context of specific DSMLs and on the other hand the use of an API to apply a specific MoCC creates a gap between the MoCC theory and the corresponding framework. In our approach we use the notion of DSE to link a MoCC to the DSA of a specific DSML and we use CCSL to specify the MoCC in a formal way, closer to theory like event structures or tagged signals. A similar approach has been used in BIP [15], where a specific algebra is used to describe the interactions through connectors between behaviors expressed in timed automata. From the properties of the connectors, it is possible to predict global properties of the models. This approach is interesting in its analysis capacity but is tailored to the composition of timed automata. Finally another approach based on CCSL has been used in [28] to describe two MoCC and the interactions between heterogeneous models of computation. This approach improved ModHel'X workbench but is still dedicated to apply a MoCC to a specific abstract syntax. However, it gives good hint for the use of the approach proposed in this paper for the composition of heterogeneous executable modeling languages.

3.7. Conclusion about the Modeling of Concurrent and Timed Operational Semantics.

This work proposes an approach that reifies the key concerns to design and implement a concurrency-aware executable DSML (AS, DSA, MoCC and DSE). The approach is supported by the GEMOC Studio eclipse project (`http://eclipse.org/gemoc`), a language workbench based on EMF, including a meta-language dedicated to each concern to design concurrency-aware executable DSMLs in a modular way. Then, the implementation of a DSML automatically results in a dedicated environment for concurrent execution, debugging and exploration of the conforming models. The explicit modeling of concurrency as first-class concern paves the way to a full understanding and configuration ability of the behavioral semantics. Additionally, the modular design enables the reuse of existing MoCC library that come with specific analysis capabilities and tool support. We illustrated our approach and language workbench on the design, the implementation and the use of various different languages, either already existing like UML or Capella or developed from scratch like Nodeus or ArduinoML.

We have many perspective on this work. They are detailed in section 6.

Chapter 4

Coordination Patterns between Heterogeneous Languages

Contents

4.1 Introduction 73
4.2 Background on Existing Approaches 74
4.3 Language Behavioral Interface 83
4.4 B-COoL 85
4.5 Validation of the Approach 90
4.6 Implementation 93
4.7 Conclusion 95

This chapter is an up to date excerpt of the following papers:

> Matias Ezequiel Vara Larsen, Julien Deantoni, Benoit Combemale, and Frédéric Mallet. A Behavioral Coordination Operator Language (BCOoL). In *International Conference on Model Driven Engineering Languages and Systems (MODELS)*, number 18, page 462. ACM, September 2015. URL `https://hal.inria.fr/hal-01182773`

> Matias Vara Larsen. *BCOol : the Behavioral Coordination Operator Language*. Theses, Université Nice Sophia Antipolis, April 2016. URL `https://tel.archives-ouvertes.fr/tel-01302875`

> Matias Ezequiel Vara Larsen, Julien Deantoni, Benoit Combemale, and Frédéric Mallet. A Model-Driven Based Environment for Automatic Model Coordination. In CEUR, editor, *Models 2015 demo and posters*, Models 2015 demo and posters, Ottawa, Canada, October 2015. URL `https://hal.inria.fr/hal-01198744`

4.1. Introduction

The development of complex software intensive systems involves interactions between different subsystems. For instance, embedded and cyber-physical systems require the interaction of multiple computing resources (general-purpose processors, DSP, GPU), and various digital or analog devices (sensors, actuators) connected through a wide range of heterogeneous communication resources (buses, networks, meshes). The design of complex systems often relies on several Domain Specific Modeling

Languages (DSMLs) that may pertain to different theoretical domains with different expected expressiveness and properties. As a result, several models conforming to different DSMLs are developed and the specification of the overall system becomes *heterogeneous*.

To understand the system and its emerging behavior globally, it is necessary to specify how models and languages are related to each other, in both a structural and a behavioral way. This problem is becoming more and more important with the globalization of modeling languages [45]. Whereas the MDE community provides some extensive support for the structural composition of models and languages (*e.g.*, [72, 110]), in this work, we rather focus on the coordination [78] of behavioral languages to provide simulation and/or verification capabilities for the whole system specification. In current coordination approaches [68, 78, 156, 164], the coordination is manually defined between particular models. This is usually done by integrator experts[1] that apply some coordination patterns according to their own skills and know-how.

In this chapter, we leverage the integrator expert's skills into a dedicated language named B-COoL that allows for capturing coordination patterns for a given set of DSMLs. These patterns are captured at the language level, and then used to derive a coordination specification automatically for models conforming to the targeted DSMLs. The coordination at the language level relies on a so-called *language behavioral interface*. This interface exposes an abstraction of the language behavioral semantics in terms of Events. B-COoL helps understand and reason about the relationships between different languages used in the design of a system.

The chapter is organized as follows. Section 4.2 provides an extensive view of the background, Section 4.2.3 sum-up the main issues in the coordination of behavioral models, and shows how they can be tackled by explicitly capturing coordination patterns at the language level. Section 4.3 defines the notion of language behavioral interface by using an example language named Timed Finite State Machine (TFSM). This language is used later in Section 4.4 to illustrate B-COoL. In Section 4.5, we validate the approach by using B-COoL to capture three coordination patterns between two languages: TFSM and fUML Activities. Section 4.6 gives an overview of the implementation of B-COoL in the GEMOC Studio Section 4.7 concludes with a brief summary and a discussion of ongoing and future actions.

4.2. Background on Existing Approaches

Based on a DSML, a domain expert builds a model to describe the structure and the behavior of a domain. However, the development of complex applications is often tackled by several domain experts. Each domain expert uses its own DSML to describe a part of the system. Thus, the use of several DSMLs results in a heterogeneous specification, *i.e.*, made of models that conform to different DSMLs.

At some point of the development, a global representation of the system is needed to reason about the system as a whole. For instance, a system designer must be able to perform verification and validation activities of the overall system. Thus, it is necessary to specify how models and languages are related in a structural and behavioral way.

This chapter presents the state-of-art approaches that give support for the heterogeneous development of systems by providing composition and/or coordination of models/languages. We begin by presenting *Composition Approaches* that propose to compose models/languages to obtain a new model/language. We continue by presenting *Coordination Approaches* that propose to specify the interaction between model/languages into an additional model so-called *Coordination Model*.

4.2.1. Composition Approaches

Composition approaches propose to compose models/languages into a new model/language. We categorize these approaches into approaches that compose models and approaches that compose languages, in both structural and behavioral way. In the following, we first present *Model Composition Approaches*, and then, we continue with *Language Composition Approaches*.

[1] also called system designers

4.2.1.1. Model Composition Approaches

Model composition has as goal to get a resulting model that is built by composing one or more models of the same language or from different languages. The resulting model can conform to the input model languages, or to a different language.

Some approaches [33, 72, 110] have automated the composition between models by using two operators: *matching* and *merging*. The matching operator is used to look for syntactic similarities between models. This results in a set of correspondences between model elements that defines *what* elements must be composed (definition in intention). From a set of correspondences, the merging operator generates a new model in which the matched elements are composed into new model elements. In [33], authors identified that the composition between models always relies on a merging of structure. They propose a matching and merging operator to compare different approaches of model composition. However, they do not propose any implementation.

In [72], the composition of models is also automated by relying on two generic operators:

- A matching operator that selects model elements by comparing the signature of the elements;
- A merging operator that composes the selected elements by using a generic algorithm [151].

These operators have been implemented in a tool named Kompose which is based on the Ecore meta language. Thus, the operators can be used to compose models that conform to different languages, as long as they conform to the same meta language (*i.e.*, Ecore).

While in Kompose these operators are generic, Epsilon [110][2] provides dedicated languages to define both the matching and merging. The matching is specified in the *Epsilon Comparison Language* (ECL) and the merging is specified in the *Epsilon Merging Language* (EML). ECL is used to define *matching rules* to specify correspondences between concepts of two languages. Matching rules apply between models and select elements that must be composed. Then, the EML is used to define *merging rules* that specify how the matched elements must be composed. This results in a new model. The metamodel of both the input models and the output model must conform to Ecore.

The approaches previously studied rely on structural similarities of the models for the composition. In other words, the matching operator only focus on the syntax of languages. While this works well with structural models such as class diagrams, it becomes a limitation when working with model whose structure represent a specific behavior like for instance Sequences Diagrams (SD). Thus, to produce a meaningful composition operator for SD, the order in which events and messages have to be composed is based on the semantics of the language. In the following, we present some approaches that have addressed this problem by proposing an asymmetric composition of models.

Aspect Oriented Modeling approaches (AOM) [104–106] propose an asymmetric composition of models in which one model plays the role of *base* and other the role of *aspects*, both models conform to the same language. The composition of aspects into the base model is named *weaving*. An aspect is made of a *pointcut* and an *advice*. The pointcut is a predicate over a model that is used to select relevant model elements called *join points*. The join points are correspondences between the aspects and the base model. During the weaving, the join points are matched in the base model, and then, they are replaced by the advices, *i.e.*, the elements of a model (aspects) are injected (woven) to another model that conforms to the same metamodel. The weaving acts as merging operator that replaces the join points by the advices.

In some approaches, the weaving of aspects considers the behavioral semantics of languages. For example, in [104], authors propose the weaving of aspects in which the base model and the aspects are represented by SD. An aspect is defined as a pair of SD: one SD serves as a pointcut (specification of the behavior to detect), and one serves as an advice (representing the expected behavior at the join point). When a behavior in the base model is detected, the join point is replaced by the SD that represents the advice. However, the detection of behaviors cannot be performed by only considering the syntax of the SD [103]. For example, consider a loop over a basic scenario where we have a message 'a' and then a message 'b'. We want to weave some extra-behavior into our system each time a message 'a' directly

[2] `http://www.eclipse.org/epsilon/`

follows a message 'b'. The only way to detect such a behavior is to unroll the loop thus using knowledge about the semantics of the loop construct. Therefore, these approaches use the knowledge of the behavioral semantics of languages for the weaving. The weaving algorithm varies depending on the approach. Thus, the resulting SD varies from one approach to another.

In this subsection, we have presented approaches that automated the composition between models by relying either on a matching and a merging operator or a weaving of aspects. Most of these approaches focus on the syntax of languages. Some of them identified that, in some cases, it is necessary to retrieve information about the behavioral semantics of the languages to compose the models. These approaches support the heterogeneous modeling of a system by providing composition capabilities. In this sense, they are suitable for development of a single system by using different DSMLs. In the next subsection, we present approaches that propose to compose languages into a new language.

4.2.1.2. Language Composition Approaches

Language composition approaches provide techniques to compose different languages into a new language We start by presenting approaches that compose the syntax of languages into a new syntax [67]. Then, we present an approach [40] that proposes to define the behavioral semantics of a language as the composition of different language behavioral semantics.

In the literature, Emerson et al. [67] propose three techniques that compose the syntax of different languages into a new language syntax:

- **Merge:** The merging composes two languages that share a concept. These concepts are used as "join points" to stitch the two languages together into a unified whole;
- **Interfacing:** When languages do not present join points, the composition requires an interface. Thus interfacing composes languages that capture distinct but related domains by relying on an interface;
- **Refinement:** One language captures in detail a modeling concept that exists only as a "blackbox" in a second DSML, *i.e.*, the concept defined in one language refines the concepts from another language.

These techniques have been implemented in the GME framework [67], which is based on the MetaGME[3] meta language. The refinement and interfacing have also been implemented in Monticore [112]. In this approach, these techniques are named respectively: *inheritance* and *language embedding*. A different approach is Neverlang [37] that relies on interfacing to build a custom language from features coming from different General Purpose Languages.

Semantic Anchoring [40] proposes to define the behavioral semantics of a language by relying on the concept of *Semantic Unit* (SU). A SU is itself a language identified as "basic", *e.g.*, Finite State Machine (FSM), Timed Automaton (TA) and Hybrid Automaton (HA). A SU is defined in the Abstract State Machine Language (AsmL[4]) in terms of (a) an AsmL Abstract Data Model (which corresponds to the abstract syntax), (b) the behavioral semantics (which is defined by the Abstract State Machine mathematical framework). SUs can be composed into a new SU. Roughly speaking, the composition is expressed manually by using AsmL.

The semantic anchoring approach proposes to define a DSML by:

- Defining the syntax by its metamodel;
- Defining the behavioral semantics by specifying the model transformation rules between the metamodel of the DSML and the abstract data model of a SU.

Such a SU could be the result of the composition of other SUs. For example, in [41], a SU named FSM (Finite State Machine) and a SU named SDF (Synchronous Data Flow) are composed to get a new SU called SU-EFSM. Then, this SU can be used to define the behavioral semantics of a heterogeneous DSMLs.

[3] `http://w3.isis.vanderbilt.edu/projects/gme/meta.html`
[4] `http://research.microsoft.com/en-us/projects/asml/`

In this subsection, we have presented approaches that compose the syntax and the behavioral semantics of languages into a new language syntax and behavioral semantics. These approaches proposed to model a heterogeneous system by using a single language which results from the composition of different languages. In this next subsection, we discuss about the benefits and drawbacks of the reviewed approaches.

4.2.1.3. Discussion About Model/Language Composition

In the previous sections, we presented approaches that addressed the use of heterogeneous DSMLs by providing composition capabilities between model/languages.

Model composition approaches automate the composition between heterogeneous models by relying on a matching and a merging operator [33, 72, 102, 110]. In particular, Epsilon [110] eases the customization of operators by providing dedicated languages. Thus, the specification of the composition can be adapted as needed. In these approaches, input and output models can conform to different metamodels, but their language must be defined in a single meta language. Most of these approaches consider only the syntax of languages thus ignoring their behavioral semantics. Only few approaches consider the semantics of languages for the composition [104–106]. Such kind of behavioral composition has been identified as being equivalent to a *scheduling* between the different model behaviors [102]. However, these approaches only compose homogeneous models (*e.g.*, sequence diagrams [104]). Thus, there use in heterogeneous systems remains very limited. Furthermore, in these approaches, the composition is encoded inside a tool. Then, to modify the specification of the composition, it is necessary to modify the implementation itself thus limiting the customization.

Only semantic anchoring [40] enables the definition of the behavioral semantics of a language by composing other behavioral semantics through the notion of Semantics Units. The authors of this work stated that their solution is to define semantics for heterogeneous DSMLs as the composition of semantic units. Their final goal is to provide a composed language. However, developing heterogeneous systems may involve different domain experts that use different languages (and consequently different tools). In this context, language composition approaches do not seem the most suitable for separation of preoccupation and development of a single system by various domain experts.

We present in the next section a different kind of approaches that propose to *coordinate* heterogeneous models/languages.

4.2.2. Coordination Approaches

Coordination approaches focus on how behavioral models interact one each other. They propose to specify the interaction between (heterogeneous) behavioral models in an additional model named *model of coordination*. In this section, like in most of the state of the art, we use the term *coordination* as being the explicit modeling of the interactions amongst behavioral models suitable to obtain the emerging system behavior. The coordination must be executable to enable the evaluation of the emerging behavior of the whole system.

We categorize the coordination approaches into *Model Coordination Approaches* and *Language Coordination Approaches*. The former proposes Coordination Languages [78] and Architecture Description Languages (ADLs) [131] to specify the coordination between behavioral models. The latter are Coordination Frameworks [27, 34] and ad-hocs solutions [25, 60] that enable the automation of the coordination between models. To do so, they have captured the specification of a coordination pattern between languages into a tool or framework, *e.g.*, Ptolemy, ModHel'X. In the following, we first present model coordination approaches, and then, we continue with language coordination approaches.

4.2.2.1. Model Coordination Approaches

Model coordination approaches provide dedicated languages to specify the coordination between (heterogeneous) behavioral models. We begin this subsection by presenting Coordination Languages, and then we continue with ADLs.

Coordination Languages [78] propose a dedicated language to model the coordination between heterogeneous behavioral models. By relying on a coordination language, a system designer builds a coordination model to specify how behavioral models interact. Depending on the entities coordinated, approaches can be categorized into *data-driven* or *control-driven*. The former coordinates data among models whereas the latter coordinates events among models. Arbab at al. [10] proposed another classification into *endogenous* and *exogenous* languages. Endogenous languages provide coordination primitives that must be incorporated within a model for its coordination. For instance, Linda [78] provides a set of primitives like *in()* or *out()* to exchange data between models. These primitives must be added to a host language by using libraries.

Exogenous coordination languages dealt 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. The coordination is thus specified between elements of the interface. The notion of interface varies depending on the approach. For instance, in *Opus* [39], 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 [170] (also named signals in [119]). 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 [9, 14, 68]. While in Esper [68], 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. This is the case of [14] where authors 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. The benefits of the use of events to coordinate the behavior of models are twofold:

- It gives support for control and timed coordination while remaining independent of the internal model implementation;
- It enables the coordination of models without any change to their implementation, thus ensuring a complete separation between the coordination and the computational concerns.

During the same period than the work presented previously, the *software architecture* community has developed so-called ADLs to gain abstraction, structuring and reasoning capabilities in the development of complex systems [3, 76, 122, 131, 156]. An ADL usually specifies a system in terms of *components* and interactions among those components. They enable a system designer to:

- Clarify structural and semantics difference between a component and its interaction;
- Reuse and compose architectural elements;
- Identify/enforce commonly used patterns (*e.g.*, architectural styles).

Depending on the ADL, a *Component* can be an encapsulation of some procedure, an encapsulation of an object file or a (formal) abstraction of its behavior. To externally characterize the components, ADLs rely on well identified *Component Interfaces*. The interfaces are used by *Connectors* whose behavior is specified by a *glue*. Connectors can represent a large variety of interactions (*e.g.*, procedure call, event broadcast or database queries) and the glue can range from a simple function to complex protocols.

Both coordination languages and ADLs make a separation between the specification of the component (*i.e.*, the computational aspects) and the assembly of these components (*i.e.*, the communication/coordination aspects). The latter is usually done by a system designer that has to deal with the architecture-level communication, which is expressed with different protocols. To abstract away these protocols and make them reusable, ADLs proposed connectors as types [131] that can be used on the shelf to specify domain specific interactions.

For example, Clara [62] is an ADL dedicated to real time systems. It proposed built-in connector types like *Rendez-vous*, *Mutex* or *Mailboxes*. A system designer can express the interactions by relying on these specific connectors that are relevant in his domain. This eases the task of a system designer, but also limits what can be used in the interaction.

Other approaches introduced the notion of *User Defined Type* [3, 11, 156] that enables a system designer to build connector types for a specific domain. A connector type is defined by a set of *roles* and a *glue*

```
System SimpleExample
    component Server =
        port provide [provide protocol]
        spec [Server specification]
    component Client =
        port request [request protocol]
        spec [Client specification]
    connector C-S-connector =
        role client [client protocol]
        role server [server protocol]
        glue [glue protocol]
Instances
    s: Server
    c: Client
    cs: C-S-connector
Attachments
    s.provide as cs.server;
    c.request as cs.client
end SimpleExample.
```

Component type

Connector type

Component instance

Connector instance

Binding

Manually specified by the System Designer

Figure 4.1: Specification of a client-server system in Wright [3]

specification. Roughly speaking, a role represents a formal parameter that is used to specify the glue which specifies how roles are coordinated. Some approaches express the glue in a formal language. For instance, in Wright [3], the glue is specified in a variant of CSP [90], differently in Reo, it is specified by the composition of dedicated primitives [11]. The connector types are later on instantiated and the roles are bound to the actual interfaces of the instances of components. By expressing the glue in a formal language, it is possible to provide reasoning about the global system behavior.

To illustrate the use of connector types, Figure 4.1 shows a simple client-server system described in Wright [3]. The specification defines two component types named *Client* and *Server*, and one connector type named *C-S-connector*. The C-S-connector has two roles (*client* and *server*) and a glue that describes how the activities of a client and a server roles are usually coordinated. The section *Instances* describes a particular configuration by instantiating the corresponding component and connector types. The example describes a system where there is a single server (*s*), a single client (*c*) and a single connector (*cs*). Then, the section *Attachments* defines which component ports are attached to the connector roles.

In this subsection, we have presented approaches that make a clear separation between the specification of the component and the assembly of these components. While the first activity is usually done by a system engineer, the second one is usually done by a system designer. To ease the task of a system designer, ADLs' community has successfully identified the need of connector types.

In [3], authors claimed that connector types help to "*understand a general pattern of interaction that can occur many times in any given system*".

Thus, a system designer has only to instantiate and bind connector types as needed by its architecture. However, the major drawback of coordination languages and ADLs is that the coordination is specified between particular models. For example, in the case of ADLs, a system designer has to instantiate and bind connector types manually. Returning to the example of Figure 4.1, for each new client in the system, the system designer has to instantiate one component and one connector; then, he has to bind the component ports with the connector roles. With the increasing number and heterogeneity of the components, this task can quickly become difficult and error prone. We present in the next subsection approaches that automate the coordination between behavioral models by specifying coordination patterns between languages.

4.2.2.2. Language Coordination Approaches

Language Coordination Approaches have identified that the instantiation and binding of connector types can be a systematic activity the system designer repeats many times and may consequently be defined as a *coordination pattern*. Such a pattern is based on the *know-how* of the system designer

and sometimes on naming or organizational conventions adopted by the models. In the following, we present approaches that have captured the specification of a *behavioral coordination pattern* inside a tool/framework to automate the instantiation and binding of connector types. In these approaches, the coordination is specified between heterogeneous languages rather than between particular models. We begin this subsection by presenting ad-hoc solutions [25, 60] which use a predefined set of languages. We continue with more systematic approaches named Coordination Frameworks [27, 34].

Mascot [25] is an approach focused on the integration of Matlab [1] and SDL [66]. Whereas SDL is a language suitable for control systems modeling, Matlab is better for modeling dataflow aspects of a system. These languages are rather different: while SDL processes operate on events, represented by simple signals, Matlab processes operate on vectors, represented by vectorized signals. Authors proposed to automate the synchronization of control signals from SDL with data signals from Matlab. To do so, the approach deals with the integration of the timing and synchronization concepts from both languages by proposing two synchronization modes: *head synchronization* and *tail synchronization*. In the head synchronization, when a model in matlab receives a frame *a1*, it immediately transforms *a1* into *b1* (dataflow network model). Any control signal from SDL that occurs during the transformation of *a1* to *b1* cannot influence its value. Then, the head synchronization mode ensures that the occurrence of the control signal is taken into account when the next frame is processed. In the tail synchronization, when a model in Matlab receives a control signal from SDL, the signal is collected until it cease to occur, then, it is translated to a vector and passed to the Matlab model. The modes of synchronization ensure the communication between Matlab and SDL by relying on the knowledge about the semantics of the languages. Once the synchronization policy is defined, the approach enables the co-simulation of a SDL model and a Matlab model. A process in the SDL specification that is specified in Matlab contains a wrapper that interfaces between the SDL simulator and the Matlab engine. A SDL wrapper is made of a set methods that enable the SDL engine to control the behavior of the data signals in Matlab. To do so, the approach relies on the name of signals in SDL specification to communicate with the signals in Matlab.

Thus, the approach forces a naming convention between signals in both domains and specified two important coordination patterns between Matlab and SDL.

Di Natale et al [60] (see Figure 4.2), dealt with the integration between a language to describe the functional aspects of a system and a language to describe the deployment platform of the system. They proposed to integrate these languages by relying on a dedicated mapping language. The mapping language syntax references syntactic elements from both the functional and the platform language to map the functions on specific computational resources from the platform. For instance, the $SWdeployment$ concept from the mapping language, references the $Task$ concept from the functional language and the CPU concept from the platform language. Based on the mapping model, the approach generates the code of the communication between the code of the functional and the code of the platform models. The semantics of both the functional and the platform languages are defined by a translational approach into C++ code. The translational semantics of the mapping language takes advantages of some knowledges about the translational semantics of the other languages. For instance, for each subsystem described using the functional language, the approach takes advantages of the knowledge that a class is generated with a name like: *SubsystemName*`ModelClass`. It also takes advantages of the knowledge that the generated class has a `step` operation used for the runtime evaluation of the block outputs given its inputs and its internal state.

To express a coordination pattern between a functional and a platform language, the approach proposed a set of connectors to specify the mapping of a functional model to a platform model. From the mapping model, the approach generates the communication code between a functional and a platform model. In this sense, the approach is similar to others ADLs that propose a set of built-in connectors. It partially automates the coordination between models since a system designer has to instantiate the connectors.

Ptolemy [34] and ModHel'X [27] went beyond previous approaches, which were ad-hoc solutions for two particular languages. Ptolemy [34] and ModHel'X [27] are systematic approaches to coordinate

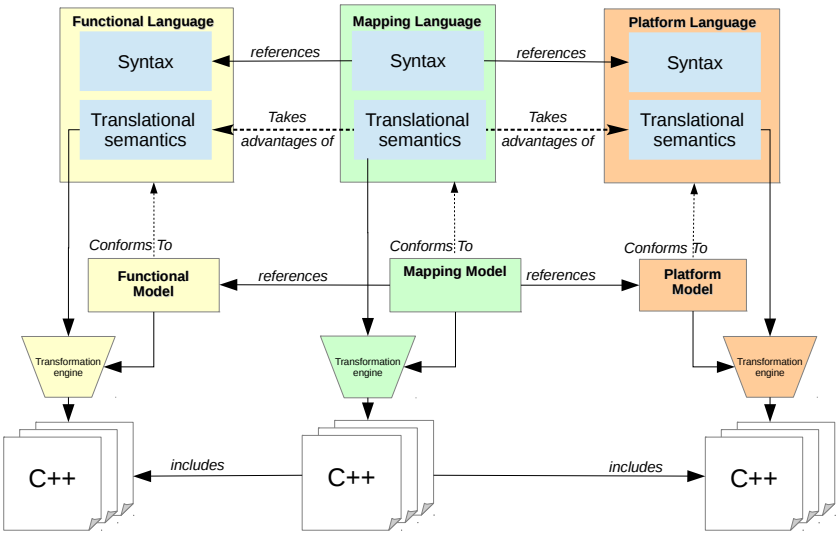

Figure 4.2: High level view of the approach proposed by Di Natale et al. in [60]

models that conform to heterogeneous languages. These approaches rely on a framework in which the syntax of models is described by actors and the semantics is given by a Model of Computation (MoC). Actors can be atomic (*e.g.*, *Actor 1* in Figure 4.3) or composite (*e.g.*, *Actor 0* and *Actor 2* in Figure 4.3), *i.e.*, made of internal, connected, actors. Each composite actor is associated with a model of computation that defines a *Domain*. A domain specifies both the communication semantics and the execution order among internal actors. A domain is implemented by a *Director*. For instance, in Figure 4.3, *Actor 0* has the director *D0*, which implements a SDF (Synchronous Dataflow) domain, and the *Actor 2* has the director *D1*, which implements a FSM (Finite State Machine) domain. In this approach, actors, both atomic and composite, are executable. In a composite actor, the execution order of internal actors is controlled by a director. In the example of Figure 4.3, the director *D0* controls the execution of the *Actor 1* and *Actor 2* and director *D1* controls the execution of *A3* and *A4* whenever *Actor 2* is executed. In this sense, the execution of composite actors is strictly hierarchical. The behavior of actors is represented by a generic interface that contains a set of methods, *e.g.*, fire(). The MoC implemented by the director of a composite actor specifies when the methods in the interface of internal actors are invoked. For instance, when an actor is fired, the director associated with a composite actor fires the internal actors.

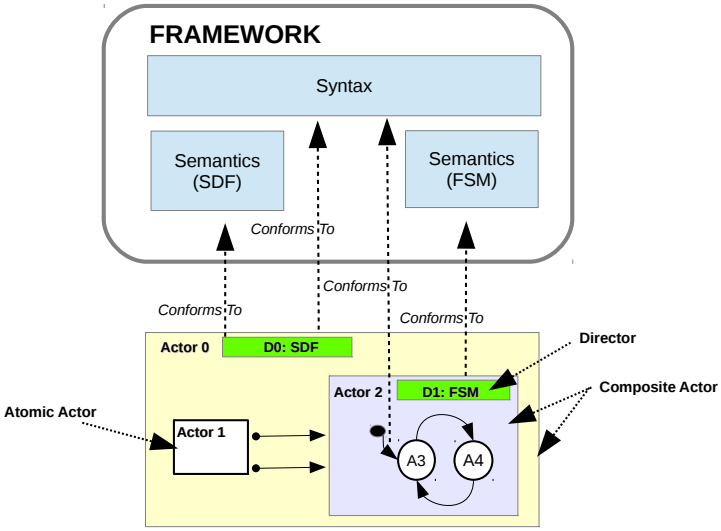

Figure 4.3: High level view of Ptolemy [80]

In conclusion, based on a fixed syntax and a generic language behavioral interface, these approaches achieved to capture a hierarchical coordination pattern into a framework. For the syntactic aspects of the pattern, the frameworks provide composite actors. Then, for the semantic aspects, they encode the glue between interfaces of composite actors and internal actors to coordinate their execution. This results in a heterogeneous model in which the execution of actors is strictly hierarchical.

In this subsection, we have presented approaches that captured the specification of a coordination pattern between languages. Such specification is defined at language level and captures a systematic way to coordinate behavioral models. By specifying the coordination at language level, these approaches have achieved to automate the coordination between models. In the next subsection, we discuss about the reviewed approaches.

4.2.2.3. Discussion About Coordination Approaches

In this section, we studied approaches to coordinate the behavior of models. While model coordination approaches proposed dedicated languages to build a model of coordination, Language coordination approaches proposed frameworks/tools that automate the coordination between models by specifying a coordination pattern between languages.

Model Coordination approaches provide dedicated languages, *i.e.*, Coordination Languages and ADLs, to specify the coordination between particular models. Such a model of coordination specifies how the behavioral models interact. The main benefit of these approaches is that the global behavior is made explicit and amenable for reasoning (for instance for Verification and Validation activities). Furthermore, they propose languages dedicated to a specific system designer domain. For example, ADLs provide connector types in order to define domain-specific connectors. However, a system designer still has to instantiate the required connector types when needed; he has to manually instantiate them by relying on his know-how. In a complex system, such a task can quickly become tedious and error prone. Furthermore, if one of the model changes, the model of coordination must also be changed. By relying on Coordination Languages and ADLs, a system designer only captures the solution for one single problem but he does not specify a systematic way to coordinate models.

Coordination frameworks achieved to capture the know-how of a system designer by specifying coordination patterns. Such specification, defined at the language level, allows for the synthesis of the coordination between heterogeneous behavioral models.

By embedding the coordination pattern inside a tool, these approaches have two major drawbacks:

1. Validation and verification activities are limited since the coordination is encoded by using a General Purpose Language (*e.g.*, Java in Ptolemy and ModHel'X);
2. The system designer cannot change the proposed coordination without altering the core of the tool.

Regarding to first point, some coordination languages and ADLs already tackled this problem by using a formal language to express the coordination (*e.g.*, CSP in Wright). This provides for verification and validation support for the coordinated system.

Concerning the second point, for complex systems, the system designer may need to capture several coordination patterns and potentially combine them. However, current coordination frameworks can only support such a variation by modifying the framework itself. The coordination model is mixed with the functional model, which makes it very tricky to modify one without risking altering the others.

4.2.3. Leason Learned From Existing Approaches

Large and complex systems are often made up with smaller "coordinated" behavioral models. There are several definitions of the notion of coordination in the literature [144]. Carriero et al. [78] define coordination as the process of building programs by gluing together active pieces. Differently, Eker et al. [64] use the word *composition* to refer to the interactions and communications between models while preserving the properties of each individual model. By relying on these definitions, in this paper, we adopt

the wording of *coordination specification* as being the explicit modeling of the interactions amongst behavioral models to obtain the emerging system behavior. In this sense, the coordination specification must be executable to enable the evaluation of the emerging behavior of the whole system.

The coordination between models can be explicitly modeled by using a *coordination language* (*e.g.*, Linda [78], Esper [68]). An integrator can define one or more coordination specifications to specify how models interact. This results in a global behavior that is explicit and amenable for reasoning (for instance for Verification and Validation activities). However, if a similar problem occurs on different models, the integrator has to define another coordination specification. When applied to models, a coordination specification only captures the solution for one single problem and does not propose a commonly applicable solution for other coordination problems.

To capture once and for all the know-how of the integrator, some approaches capture coordination patterns by specifying the coordination at the language level. This is the case for coordination frameworks like Ptolemy and ModHel'X, which rely on hierarchical coordination patterns between heterogeneous languages. This is also the case for solutions like MASCOT [25], which provide ad-hoc coordination patterns (*i.e.*, between Matlab and SDL). Once the coordination pattern between a set of languages is captured, the models conforming to such languages can be coordinated automatically. Such approaches successfully capture the know-how of the integrator, however, they do so by embedding the coordination pattern inside a tool. As a result, the integrator cannot change the coordination specification without altering the core of the tool. However, for complex systems, the integrator may need to capture several coordination patterns and potentially combine them. This is highlighted in [79] where authors use Ptolemy to capture three hierarchical coordination patterns between a finite state machine with Data flows, a Discrete event model and a Synchronous/Reactive model. Since each coordination pattern is applicable in different situations, authors argue that each one is useful. Thus, the integrator must be able to capture different coordination patterns since there is not necessarily a single *valid* one. However, current coordination frameworks can only support such a variation by modifying the framework itself. Additionally, the coordination model is mixed with the functional model, which makes it very tricky to modify one without risking altering the others.

During the integration activity, the integrator must be able to capture different coordination patterns and their semantics must be made explicit rather than being hidden inside a tool[5]. Moreover, validation and verification activities are limited in current coordination frameworks since the coordination is encoded by using a general purpose language.

The knowledge about system integration is currently either implicitly held by the integrator or encoded within a framework. To capture explicitly this knowledge and thus leverage integrator know-how, we propose B-COOL (Behavioral Coordination Operator Language), a dedicated language to capture coordination patterns between languages, thus reifying the coordination specification at the language level (see Figure 4.4).

A B-COOL specification captures a coordination pattern that defines what and how elements from different models are coordinated. Once specified in B-COOL, integration experts can share this specification thus allowing the reuse and tuning of coordination patterns. Also, such a specification can be exploited by generative techniques to generate an explicit coordination specification when specific models are used.

To be able to specify the coordination between languages, a partial representation of the language behavioral semantics is mandatory. In our approach, the semantics is abstracted by using a behavioral language interface. This notion of behavioral language interface is further discussed in the next section and is illustrated with a language named TFSM (Timed Finite State Machine). This language is also used in Section 4.4 to introduce B-COOL.

4.3. Language Behavioral Interface

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

[5]for coordination, the problem is the same than for semantic variations hidden in the encoding of different tools, where the resulting semantics is difficult to apprehend like illustrated in [13]

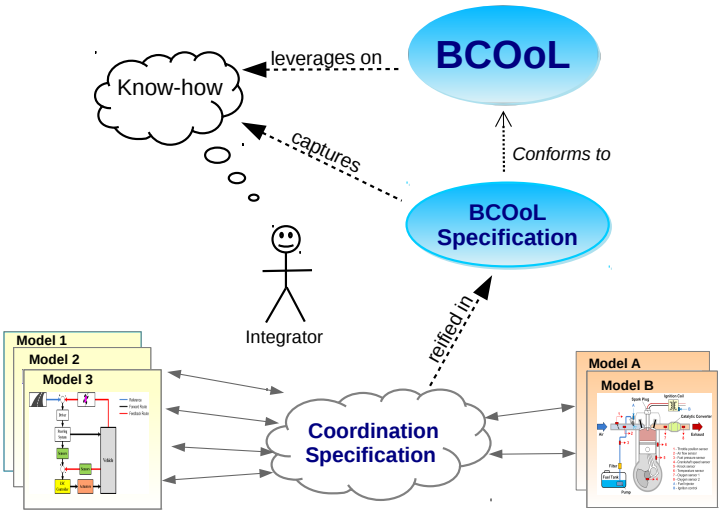

Figure 4.4: Coordination at the language level with BCOoL (taken from [165])

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* [39], 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 [170] (also named signals in [119]). 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 [14, 68, 156]. While in Rapide [156], 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. This is the case of [14] where authors 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 in 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 [170] 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.*, [6] or [20]).

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 [44], as presented in Chapter 3, elements of event structures are reified at the language level to propose a behavioral interface based on sets of *event types* and *contraints*. 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 DSE are attached to a metaclass, MSE 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 the DSE (and the corresponding MSE) as handles or control points in three complementary ways: 1) to specify coordination patterns at the language level, 2) to observe what happens inside the model under execution, and 3) to control what is allowed to happen or not when two (or more) model executions need to be coordinated. 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 execution paths are allowed (*e.g.*, due to different possible interleavings), 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 reused the simple state-based language named Timed Finite State Machine (TFSM) and its behavioral interface introduced in the previous chapter (See Figure 3.6).

In the following section, the Behavioral Coordination Operator Language is detailed.

4.4. B-COoL

4.4.1. 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. A specific set of *operators* encodes a coordination pattern and specifies 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 instances of DSE in a specific set of models. Therefore, the generated coordination model implements the coordination patterns defined at the language level and dedicated to the actual models under coordination.

The design of B-COoL is inspired by existing structural composition languages (*e.g.*, [72, 110]). 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 (*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 [12][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 "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 level. It is then possible to apply it on specific state machines, *i.e.*, at the model level. Note that this first behavioral coordination pattern is homogeneous (*i.e.*, it

[6]Note that, because this is an homogeneous coordination, it may be realized by a specific merge of the automata structure. Here we consider only the resulting behavioral semantics and we want to express the synchronized product as a coordination pattern

involves a single language). Examples of heterogeneous coordination patterns (*i.e.*, that involve several languages) are provided in Section 4.5.

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 used the studio on an heterogeneous version of the running example, and we generated the coordination model between a TFSM model and an activity diagram model. We then shown how the generated coordination model can be exploited for execution and exhaustive simulation.

4.4.2. Abstract Syntax of B-COoL

The root element of B-COoL (see Figure 4.5) is a *BCoolSpecification* that contains imports of language behavioral interfaces (*importsInterfaceStatements*) and *Operators*. The specification must import at least two language behavioral interfaces (line 2 and 3 on Listing 4.1). Interfaces provide the *DSE* needed for the coordination. The imported DSE serve as parameters for the operators. Then, an operator specifies what instances of these DSE are selected and how they are coordinated (the dse reference). For instance, to build the synchronized product of TFSM, we need to synchronize the instances of the *occurs* DSE from different models. This is done by coordinating taking two *occurs* DSE as parameters of the operator (line 7 on Listing 4.1)).

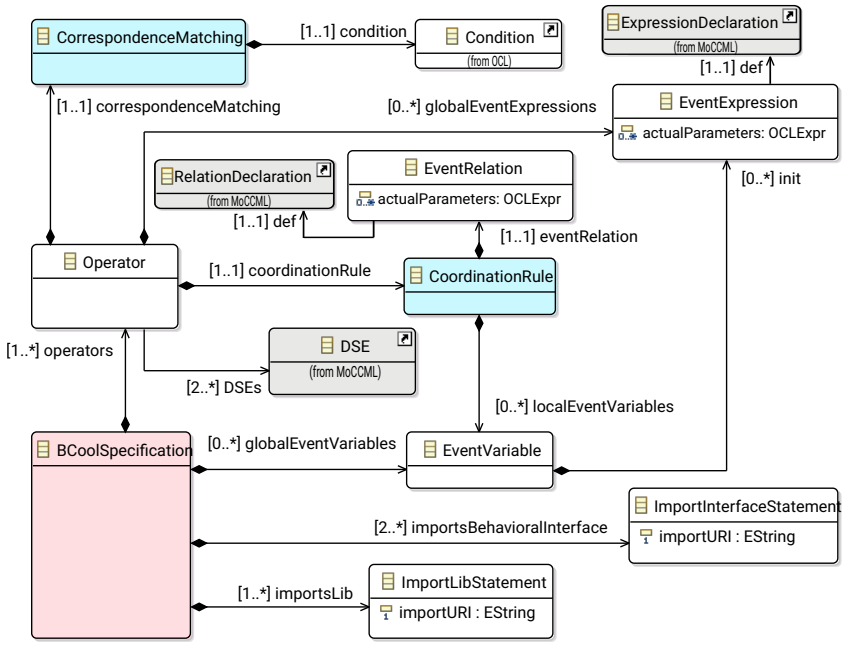

Figure 4.5: Simplified view of B-COoL abstract syntax

Listing 4.1: synchronized product operator between two TFSM models

```
1 BCOoLSpec SynchronizedProduct
2   ImportInterface "TFSM.ecl" as lbi1
3   ImportInterface "TFSM.ecl" as lbi2
4
5   ImportLib "facilities.moccml"
6
7 Operator SyncProduct(dse1 : lbi1::occurs, dse2 : lbi2::occurs)
8   CorrespondenceMatching: when(dse1.name = dse2.name)
9   CoordinationRule: RendezVous(dse1, dse2)
10 end operator
```

Each operator contains both a *correspondenceMatching* (line 8 on Listing 4.1) and a *coordinationRule* (line 9 on Listing 4.1). 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 the *name* attribute of an FSMEvent (FSMEvent being the context of the occurs DSE).

The *coordinationRule* specifies how the selected instances of DSE must be coordinated. To do so, the user can define some *EventVariables* (*localEventVariables*) and must use 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. Both event relations and event expressions are defined by respectively using *ExpressionDeclaration* and *RelationDeclaration* from MoCCML, inheriting this way its expressiveness and formal definition. In B-COoL, coordination librairies can be explicitly imported (line 5 on Listing 4.1 and *ImportLibStatement* on Figure 4.5).

How the selected events are coordinated is determined by event relations that enforce a (partial) order between 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.

Now that we presented a very simple example, let us adapt it so that it becomes heterogeneous. This first heterogeneous operator differs from the one on Listing 4.1 because it applies to heterogeneous languages (namely fUML and TFSM).

Before going into the example, we briefly present the language behavioral interface of the fUML language partially shown in Listing 4.2. For each *Activity* two DSE are defined: *startActivity* and *finishActivity*, to identify respectively the starting and finishing instants of the activity. Similarly, Two DSE are defined for each *Action*: *startAction* and *finishAction*, to identify the starting and the finishing of an Action. These DSE, part of the fUML language behavioral interface, are used throughout this section to specify the coordination operators.

Whereas in the running example the occurrences of FSMEvents were synchronized with the occurrences of other FSMEvents, here they are synchronized with the start of fUML *Actions*, *i.e.*, by coordinating instances of DSE *occurs* and *startAction*.

Then, when the name of an `Action` is equal to the name of an `FSMEvent`, the respective *startAction* and *occurs* DSE are coordinated with a rendez-vous relation.

Listing 4.2: Partial MoCCML mapping specification of Activity Diagram

```
1 package uml
2  context Activity
3   def: startActivity : Event = self
4   def: finishActivity: Event = self
5  context Action
6   def : startAction : Event = self
7   def : finishAction : Event = self
```

Like the previous one, this operator imports a facility lib written in MoCCML where the Rendez-Vous relation is defined. It also imports the language interfaces of the activity and tfsm languages (line 3 and 4 on Listing 4.3). The operator is based on two DSE, one of type *startAction* defined on the `Action` concept from the activity language and one of type *occurs* defined on the `FSMEvent` concept from the tfsm language ((line 6 on Listing 4.3). The DSE matches when the name of the `Event` instance and the name of `Action` instance are the same (line 7 on Listing 4.3). Fincally, the coordination rule to apply is the Rendez-Vous relation (line 8 on Listing 4.3).

Listing 4.3: Heterogeneous synchronized product operator between the TFSM and fUML languages

```
1 BCOoLSpec TFSM-fUMLOperators
2   ImportLib "facilities.moccml"
3   ImportInterface "activitySemantics.ecl" as activity
```

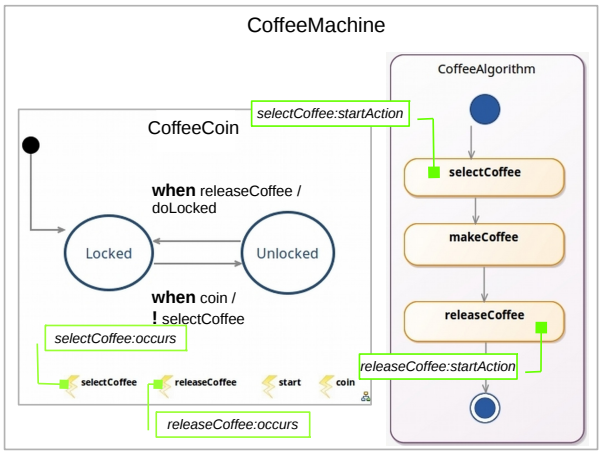

Figure 4.6: Simple heterogeneous specification of a coffee machine (taken from [163])

```
4   ImportInterface "TFSM.ecl" as tfsm
5
6 Operator SyncProduct(dse1 : activity::startAction , dse2 : tfsm::occurs)
7   CorrespondenceMatching: when(dse1.name = dse2.name)
8   CoordinationRule: RendezVous(dse1, dse2)
9 end operator
```

Based on Listing 4.3, the execution semantics of B-COOL is specified in the next section.

4.4.3. Execution Semantics

In this subsection, we describe the execution semantics of B-COOL, *i.e.*, how a B-COOL specification is used to generate a coordination model. To illustrate the different steps in the generation on Figure 4.7, we rely on the application of the operator presented in the previous section (see Listing 4.3). In order to explain the whole semantics, we need to present two models on which to apply the previously defined operator. We use for this purpose a very simple coffee machine defined two models, one TFSM model to handle the payment and one activity to handle the coffee making (see Figure 4.6). Of course, the coffee making is only possible after having payed.

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, with $N \geq 2$. Also, a B-COOL specification contains a set of operators $\mathcal{O}p$. 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. 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. For instance, when the operator from Listing 4.3 is applied between the two models of the coffee machine, the first step consists in extracting the MSE of the *CoffeeAlgorithm* activity diagram and the *CoffeeCoin* state machine. It results in two sets of MSE named respectively $\mathcal{I}_{\mathcal{M}}1$ and $\mathcal{I}_{\mathcal{M}}2$ (step 1 in Figure 4.7).

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 interface (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 $\mathcal{I}'_{\mathcal{M}_{\mathcal{I}}}$ (step 2 in Figure 4.7). For instance, the operator of Listing 4.3 has as parameters the *Action::startAction* DSE and *FSMEvent::occurs*. Thus, the set $\mathcal{I}'_{\mathcal{M}_{\mathcal{I}}}$ is

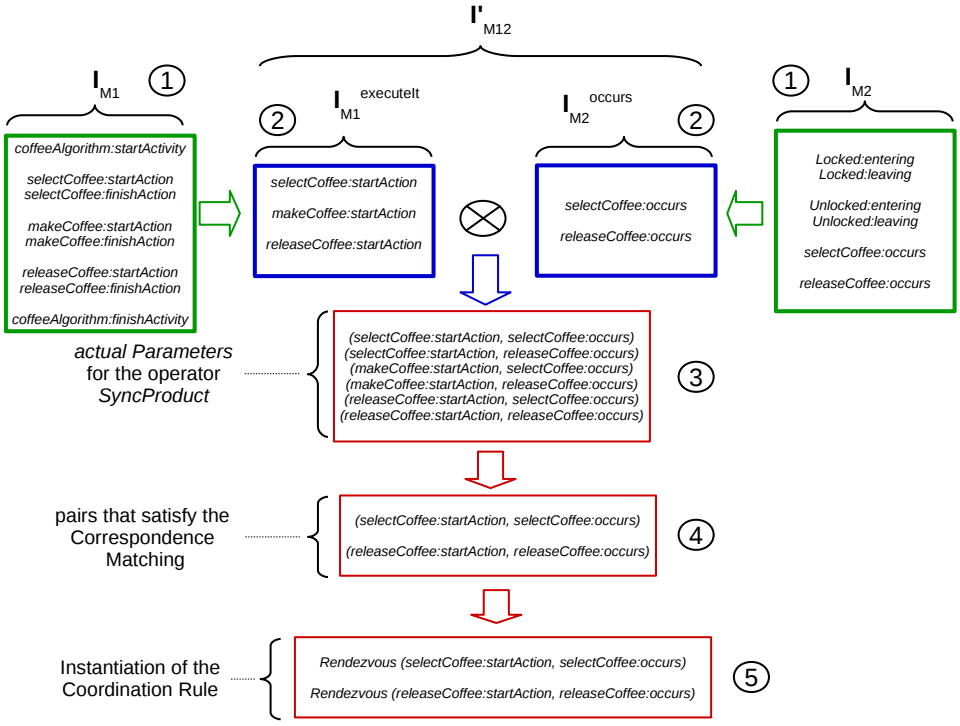

Figure 4.7: Steps in the application of the B-COoL specification between the models of the coffee machine

composed of two set named $\mathscr{I}_{\mathscr{M}_1}^{startAction}$ and $\mathscr{I}_{\mathscr{M}_1}^{occurs}$ that corresponds respectively with events of type *Action::startAction* and *FSMEvent::occurs* (step 2 in Figure 4.7).

Second, a classical Cartesian product is applied on $\mathscr{I}'_{\mathscr{M}_\mathscr{I}}$. 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 (step 3 in Figure 4.7). 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 $actualParams$. Returning to the illustrating operator, the correspondence matching condition is used to select MSE by comparing the instances names. This results in two selected sets: *selectCoffee:occurs* and *selectCoffee:startAction*, and *releaseCoffee:occurs* and *releaseCoffee:startAction* (step 4 in Figure 4.7). The coordination rule is instantiated two times. 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 step. 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. In the example depicted on Figure 4.6, the event relation rendezvous is instantiated twice; one time for each set in $actualParams$ that satisfies the CMC (step 5 in Figure 4.7).

Currently, the application of a B-COoL operator generates a CCSL specification that represents the coordination. Listing 4.4 shows the partial CCSL specification for the coffee machine. The specification begins by importing the CCSL specification of each model (Listing 4.4: line 4 and 5). Then, the main block contains the coordination specification that is made of two instances of the RendezVous event relation (Listing 4.4: line 10 and 14). Notice that individual specification of each model are not modified. So that, the behavior of individual models is not altered. Instead, the coordination adds some constrains thus restricting the behavior of models, but it does not add new behaviors. This results in a generated coordination that is not intrusive (*i.e.*, exogenous).

In this section, we have presented the abstract syntax and the semantics of B-COoL. We saw that B-COoL reuse MoCCML for the definition of event relation and expression. In the next section, in order to validate the approach, we present some examples in B-COoL.

Listing 4.4: Resulting CCSL specification for the coffee machine system

```
1 ClockConstraintSystem TFSMandActivity {
2 imports {
3   import "facilities.moccml" as lib;
4   import "coffeeCoin.extendedCCSL" as coffeeCoin;
5   import "coffeeAlgorithm.extendedCCSL" as coffeeAlgorithm;
6 }
7 entryBlock mainBlock
8   Block mainBlock {
9     Block coffeeCoincoffeeAlgorithmsublock {
10      Relation SyncProduct_selectCoffee_startAction_selectCoffee_occurs[ RendezVous ](
11        ClockA -> "coffeeAlgorithm::selectCoffee_startAction",
12        ClockB -> "coffeeCoin::selectCoffee_occurs"
13      )
14      Relation SyncProduct_releaseCoffee_startAction_releaseCoffee_occurs[ RendezVous ](
15        ClockA -> "coffeeAlgorithm::releaseCoffee_startAction",
16        ClockB -> "coffeeCoin::releaseCoffee_occurs"
17      )
18    }
19  }
20 }
```

4.5. Validation of the Approach

In this section, we validate the use of B-COoL through the specification of four coordination operators. Each operator captures a given coordination pattern between two languages: TFSM (the language presented in Chapter 3) and the fUML language [135] quickly introduced in the previous section. The operators are used to coordinate the model of a video surveillance system. By using this example, we illustrate the benefits of our approach for the definition of coordination operators (*i.e.*, understandability and adaptability), the generation of the coordination specification (*i.e.*, usability) and the studies of the coordinated system (*i.e.*, analysis capabilities). We finish this section by quickly comparing our approach with coordination languages (like the one presented in Section 4.2.3) and with ad-hoc approaches like the one of [60] (best paper of MODELS 2014).

4.5.1. Definition of Coordination Operators between the TFSM and fUML Language

In addition to the B-COoL specification started in Listing 4.3, we defined three new coordination patterns between the TFSM and fUML languages.

For the second and third operators, we specify a hierarchical coordination pattern between the TFSM and fUML languages, unlike hierarchical coordination frameworks where the semantics is hidden, these operators explicitly specify how the hierarchical coordination is implemented. In our case, we chose the semantics in which entering a specific state of a TFSM model triggers the execution of a given fUML activity. When leaving a state, several semantic variation points may be chosen. The outgoing transitions from a state can be considered, for instance, as preemptive for the activity model (*i.e.*, firing a transition from a state to another preempts the internal activity). Alternatively, the transition can be considered as non-preemptive (*i.e.*, the states cannot be left before the associated activity finishes). In this paper, we chose non-preemptive transitions, but preemptive ones and more examples are provided on the B-COoL webpage: `http://gemoc.org/BCOoL/index.html`. Listing 4.5 presents the B-COoL specification that is organized around two operators: *StateEntering* and *StateLeaving*.

The *StateEntering* operator coordinates the action of entering into a state with the start of an activity so that when a state is entered, the execution of the activity is started synchronously. Entering into a state is identified by the *entering* DSE defined in the context of State (see Figure 3.6). Instances of such DSE have to be coordinated with instances of the *startActivity* DSE. To identify pairs of such events, the correspondence matching selects events by comparing the *onEnterAction* defined in the states and the name of the activities (Listing 4.5: line 12). OnEnterAction is a method defined in the context of State (see Figure 3.6) that specifies the method invoked when a state is entered. In our case, we use this attribute to specify the name of the activity that the state represents. Then, since we have selected this semantics in which entering a state synchronously triggers the starting of an activity, the coordination rule specifies a rendez-vous relation between the selected pairs of DSE *entering* and *startActivity* (Listing 4.5: line 13).

The *StateLeaving* operator (Listing 4.5: line 16) coordinates the finishing of the activity with the leaving of the state. Leaving a state is identified by DSE *leaving* and finishing an activity is identified by DSE *finishActivity*. In this case, the coordination rule has to express that leaving the state follows the termination of the activity. To do that, we use a causal event relation (Listing 4.5: line 19).

Listing 4.5: Hierarchical coordination operators between TFSM and fUML languages

```
10 Operator StateEntering(dse1 : activity::startActivity , dse2 : tfsm::entering)
11   CorrespondenceMatching:
12    when(dse1.name = dse2.onEnterAction.name)
13   CoordinationRule: RendezVous(dse1, dse2)
14 end operator
15
16 Operator StateLeaving(dse1 : activity::finishActivity , dse2 : tfsm::leaving)
17   CorrespondenceMatching:
18    when(dse1.name = dse2.onEnterAction.name)
19   CoordinationRule: Causality(dse1, dse2)
20 end operator
```

For the fourth operator, we deal with the temporal aspects of the model coordination. The operator specifies how the time in the TFSM elapses during the execution of the activities that specify the on-entry action of a state. This coordination is also hierarchical, but in this case, only considers the timing aspects. In the TFSM language, each state machine has a *localClock* used to measure the time (see Figure 3.6) while the fUML language is untimed. The local clock is a *FSMClock*, which defines a DSE named *ticks* whose occurrences represent a physical time increment. In the fUML language, the duration of activities can be represented as the time between the DSE *startActivity* and DSE *finishActivity* (Listing 4.2). To coordinate the time, it is necessary to specify the number of *ticks* of the local clock between the occurrence of the DSE *startActivity* and *finishActivity*. We propose an operator that enforces the execution of the "internal" activity to be atomic with respect to the time in the TFSM model. As a result, there is no occurrence of the DSE ticks of the corresponding local clock during the execution of the activity.

Listing 4.6 captures the corresponding coordination pattern by defining the operator named *NoTimeinRefinedActivity*. The operator selects instances of DSE startActivity and finishActivity by using their context. As a result, the selected DSEs identify the starting and finishing of an activity. Then, we select the activities that represent a state (Listing 4.6: line 24). To do so, we use the onEnterAction defined in the context of State. Then, we use the selected instances of DSE entering to select instances of DSE ticks of the corresponding local clock (Listing 4.6: line 25). The coordination rule must specify how much time is consumed during the execution of an activity. First, we use the event expression *SampledBy*(as defined in [57] and [56]) to create a local event named *sampled* which ticks always after the startActivity instance, and coincides with the occurrences of the "next" instant of the corresponding *ticks* DSE instance (Listing 4.6: line 27). Second, we synchronize the event sampled with the finishing of the activity by using a causality relation (Listing 4.6: line 28). This forbids instances of *ticks* to occur between the start and the end of an activity.

The coordination rule presented earlier can be built by relying on a specific MoCCML library. In this case, we have to extend the *facilities.moccml* with a new event relation named *atomicActivity*. Then, we have to replace the event expressions and relations by the event relation atomicActivity with the corresponding parameters (*i.e.*, dse1, dse2, dse4). The use of the library to define domain specific relations has two major benefits. First, once defined in the library, event relations can be reused in various B-COoL specifications. Second, by defining a dedicated event relation, we improve the readability and modularity of the B-COoL specification.

Listing 4.6: Timing coordination operator between TFSM and fUML language

```
21 Operator NoTimeinRefinedActivity(dse1 : activity::startActivity , dse2 : activity::
       finishActivity , dse3 : tfsm::entering , dse4 : tfsm::ticks)
22  CorrespondenceMatching:
23   when (dse1.name = dse2.name)
24   and  (dse1.name = dse3.onEnterAction.name)
25   and  (dse3.owningFSM.localClock = dse4)
26   CoordinationRule:
27    Local Event sampled = SampledBy(dse1, dse4);
28    Causality(dse2, sampled)
29 end operator
```

4.5.2. Use of Coordination Operators in a Surveillance Camera System

In this section, we develop the heterogeneous model of a surveillance camera system (see Figure 4.8). To model different aspects of the system, we use the TFSM and the fUML languages. Then, we use the operators developed in the previous section to generate the coordination specification.

The video surveillance system is composed of a camera and a battery control. The camera takes pictures by using either the *JPEG2000* or *JPG* algorithm and is powered by a battery. When the battery is low, the battery control makes the camera use the *JPG* algorithm, thus reducing the quality of the picture but also the energy consumption [152]. When the battery is high, the JPEG2000 algorithm is used instead. In Figure 4.8, the activity diagrams named *BatteryControl* represents the simple algorithm implemented in the battery control. At the bottom of Figure 4.8, the TFSM named *CameraControl* represents a partial view of the camera. When the TFSM model is in state *BatteryHigh*, the JPEG2000 algorithm is used (specified by the activity diagram on the right of Figure 4.8 named *doJPEG2000*). When in state *BatteryLow*, the encoding algorithm is replaced by a mere JPEG algorithm represented by an activity named *doJPEG* (The activity is not shown for better readability). The transition from one state to another is done when either the *BatteryIsHigh* event or the *BatteryIsLow* event occurs, depending on the current state.

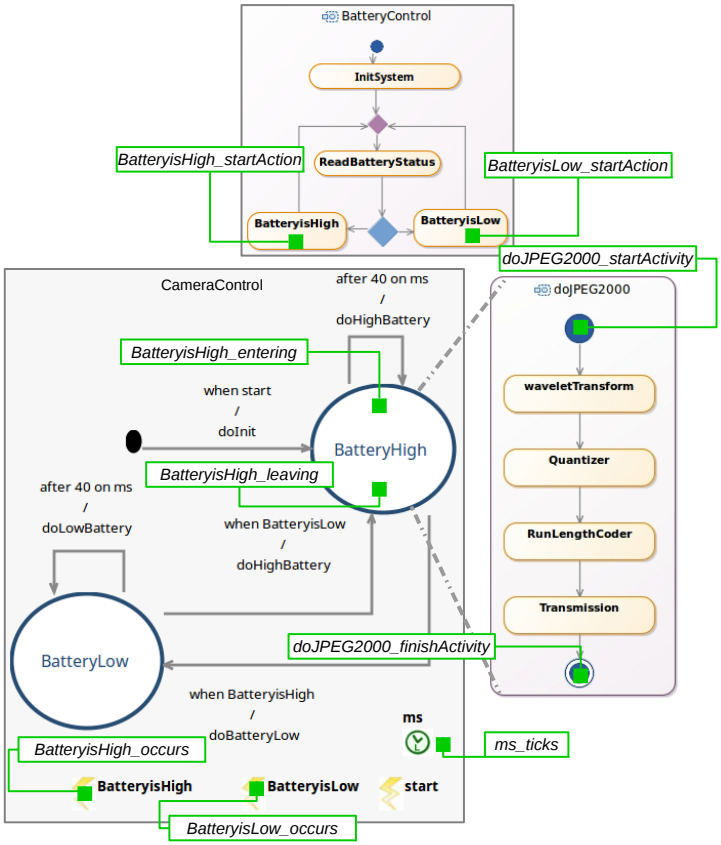

Figure 4.8: Hierarchical model of a surveillance camera system and a partial representation of the behavioral interface (taken from [165])

To coordinate the models, we have to specify a timing and hierarchical coordination between the states of the TFSM CameraControl and the activities doJPEG and doJPEG2000. In addition, we have to synchronize the activity BatteryControl and the TFSM CameraControl by coordinating the corresponding Action and FSMEvent. Applying the four operators on these simple models, we generate the expected coordination specification. The coordination generated by using our approach corresponds to eight CCSL relations.

4.5.3. Use of the Coordination Specification

In B-COOL, the generated coordination specification conforms to the CCSL language. Since we are using a formal language, the integrator can execute and verify the coordination specification of the system. By using the language workbench presented in Section 4.6, the coordination specification generated for the surveillance camera system can be executed and analysed. More precisely, we are able to execute the coordination specification by using TimeSquare, and to explore the state space.

4.5.4. Comparison with Existing Approaches

In B-COOL, the definition of the coordination between languages is based on operators. In particular, coordination rules explicitly define the semantics of the resulting coordination. The reader can notice that variations of the semantics of the resulting coordination can be done by only modifying the coordination rules of the operators. In frameworks like Ptolemy, such a variation is only supported by modifying the framework itself. For instance, in Ptolemy, this means changing the current implementation of a *director* written in Java. The same problem appears in ad-hoc translational approaches [60], where the transformation needs to be changed. Since this state of the art approach is using general-purpose transformation frameworks, this work needs a good knowledge of coordinated languages as well as a good knowledge of the transformation language itself. This is beyond the expected skills of an integrator. In our approach, we are using a language dedicated to integrator experts thus easing the understanding and adaptation of the B-COOL specification.

The definition of domain specific coordination operators enables the automation of the coordination between models. For instance, in the case of the video surveillance system, the application of the operator generates eight CCSL relations. By manually coordinating the models (as proposed in [164] or when using a coordination language), this would require to specify each relation manually. The reader can notice that the number of relations increases with the number of model elements involved in the coordination. For instance, for a system with N cameras, the integrator would need to specify 8*N relations. Our proposition is to leverage this task for the integrator at the language level and then to generate all the required relations accordingly.

Regarding system execution and verification, both coordination languages and coordination frameworks allow to execute the coordinated system, however, the verification varies from one approach to another. Some coordination languages rely on a formal language thus providing verification. Differently, in Ptolemy, the main validation method is based on the simulation of the coordinated system [79]. In our approach, by relying on CCSL, we are able to provide execution and verification of the coordinated system.

4.6. Implementation

This section presents the implementation of B-COOL into the GEMOC studio[7]; which integrates technologies based on Eclipse Modeling Framework (EMF) [8] adequate for the specification of executable domain specific modeling languages. The studio includes a *language workbench* to design and implement tool-supported DSMLs and a *modeling workbench* where the DSMLs are automatically deployed to allow designers to edit, execute and animate their models (see Chapter 3 or [47]).

B-COOL takes advantages of this collaborative environment by adding coordination facilities. In this section, we illustrate the implementation of the language workbench by developing the coffee machine example operator (see Listing 4.3). Then, in the modeling workbench, we use this operator to execute and verify the models of the coffee machine.

B-COOL is developed as a set of plugins based on the EMF (at top of Figure 4.9). The B-COOL abstract syntax has been developed using Ecore (*i.e.*, the metalanguage associated with EMF) and the textual

[7] `http://eclipse.org/gemoc`
[8] `http://eclipse.org/modeling/emf/`

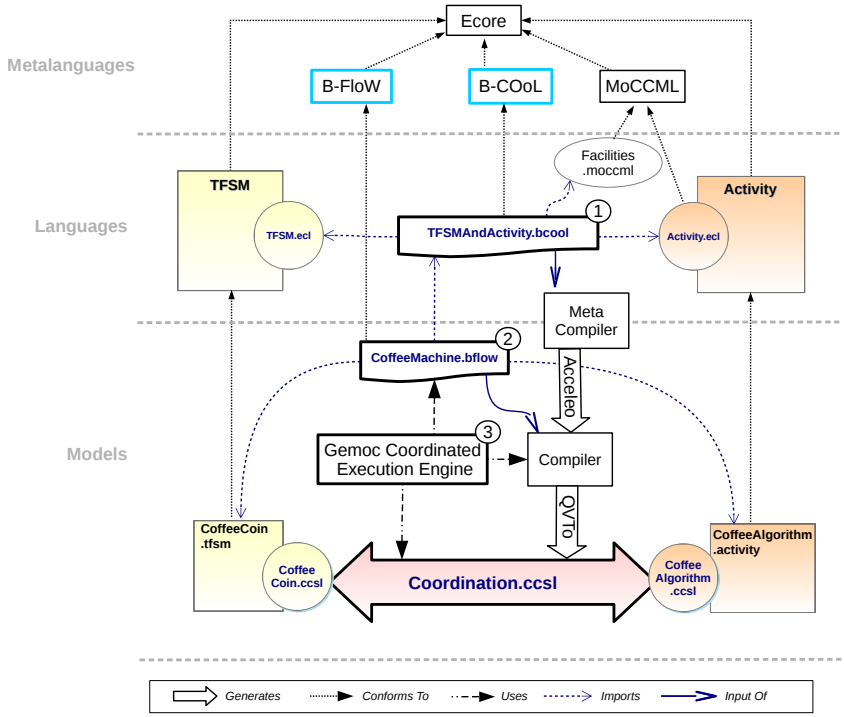

Figure 4.9: Overview of the implementation of B-COoL and its integration into the Gemoc Studio

concrete syntax has been developed in Xtext [9], thus providing advanced editing facilities. For the running example operator, we use the TFSM and Activity languages that have been developed into the studio. Then, we use B-COoL to specify the Listing 4.3 (Figure 4.9: step 1). In the B-COoL specification, we can import the language behavioral interfaces of each language deployed in the language workbench. In addition, the language workbench provides MoCCML thus helping the integrator to specify relations and expression.

In the modeling workbench, a system designer can use B-COoL operators to automate the coordination of models and to execute the coordinated system. To do so, a system designer has to specify a B-FLOW specification (Figure 4.9: step 2), and then, uses it to launch the *Gemoc Coordinated Execution Engine* (Figure 4.9: step 3). In the following, we elaborate on these two tasks.

We provide a simple language named B-FLOW (standing for B-COoL *FLoW*) that enables a system designer to specify how operators of a B-COoL specification are applied on a set of models (Figure 4.9: step 2). To introduce B-FLOW, Listing 4.7 shows the specification for the models of the coffee machine. It begins by importing the B-COoL specification that contains the operators (Listing 4.7: line 3). Then, it specifies the models that will be coordinated. For the running example, this corresponds with the TFSM named *CoffeeCoin.tfsm* and the Activity named *CoffeeAlgorithm.ad* (Listing 4.7: line 5 and 6). Then, the specification contains a *Flow* that defines which operators are used and on which models they are applied. A Flow defines a sequential order of application of operators. In other words, the first line is the first operator that will be applied and soon on. For instance, in Listing 4.7, line 9 specifies that the operator named *SyncFMEventsAndActions* must be applied between the models CoffeeCoin and CoffeeAlgorithm. This corresponds with the first and the only operator to be applied to coordinate the models of the coffee machine. However, a B-FLOW specification may use several operators depending on the number of models in the system.

Listing 4.7: B-FLOW specification for the models of the coffee machine

```
1 BCOoLFlow CoffeeMachine
2
3  ImportBCOoL  "TFSMAndActivity.bcool" ;
4
```

[9] http://eclipse.org/Xtext/

```
5   Model CoffeeCoin "coffeecoin.tfsm"
6   Model CoffeeAlgorithm "coffeeAlgorithm.ad"
7
8  Flow
9   applies SyncFSMEventsAndActions between (CoffeeAlgorithm, CoffeCoin);
10 end Flow;
```

The Gemoc Coordinated Execution Engine uses the B-FLOW specification to generate a model of coordination that is used to execute the coordinated system. The generation is implemented by using a high-order transformation in Acceleo [10] that translates the B-COOL specification into a QVTo [11] transformation (*Compiler* in Figure 4.9). Then, the Gemoc Coordinated Engine invokes the generated QVTo transformation which takes as parameter the B-FLOW specification from which the models to coordinate and the operators to apply are retrieved. The QVTo transformation is finally applied between the corresponding models thus generating a model of coordination in CCSL.

To execute the coordinated models, the Gemoc Coordinated Execution Engine first initializes the *Gemoc Execution Engine* of each individual model. These engines compute the next valid step for each model. Also, there is a *Coordination Engine* that computes the next valid step for the coordination. To compute the next *global* valid step, the Gemoc Execution Engine gets the next valid step from each individual engine and from the coordination engine. Then, it selects the possible steps that are valid for both the individual models and the coordination. This results is a set of global valid steps.

To illustrate the use of the Coordinated Execution Engine, we coordinate and execute the models of the coffee machine. First, we configure the launcher that contains information about the B-FLOW specification and the configuration launcher for each model. In Figure 4.10, by clicking on *Debug*, the execution is launched. Then, each individual engine is initialized together with the engine for the coordination (Figure 4.11: point 2). The *Concurrent Logical Steps Decider* view provides several options to drive the execution of the models. For instance, it provides a list of the next valid execution steps (Figure 4.11: point 3). Also, the workbench provides the animation of models (Figure 4.11: point 4).

The modeling workbench provides tools to analyze the resulting CCSL specification (Figure 4.11: point 1). For example, it is possible to obtain by exploration quantitative results on the scheduling state-space of the coordinated system. The exploration of all schedules can be done explicitly in a state space graph. Any cyclic path in this graph (starting from the initial configuration) represents a valid schedule of the models. Figure 4.12 shows the resulting state-space for the coordinated models of the coffee machine[12]. Each state represents a valid step in the execution. The transitions contain the events that tick simultaneously when a transition is taken. For example, in red, Figure 4.12 shows the transitions that contain the events forced to happen simultaneously by the coordination, *e.g.*, *selectCoffee:occurs* and *selectCoffee:executeIt*. For the coffee machine, the state exploration results in 39 states, 405 transitions and no dead-locks.

The project B-COOL is hosted on Github as part of the GEMOC project[13], thus making the source code publicly available. B-COOL is currently integrated into the GEMOC studio[14]. To try the coffee machine example, the reader needs to download the studio and then to follow the tutorial from the companion website[15]. In addition, the website contains more examples with full descriptions. The reader should notice that, at the time I'm writing this chapter (December 2018), B-COOL is under strong refactoring in order to be moved to the eclipse gemoc project[16].

4.7. Conclusion

In this chapter, we addressed the problem of the coordination of behavioral models by reifying coordination patterns at the language level. We present B-COOL, a dedicated (meta)language to capture coordination patterns between modeling languages and generate a formal coordination model for conform-

[10] `http://www.eclipse.org/acceleo/`
[11] `https://projects.eclipse.org/projects/modeling.mmt.qvt-oml`
[12] The graph in DOT format can be downloaded from `http://gemoc.org/BCOoL/index.html#bcoolcoffeemachine`.
[13] `http://github.com/gemoc/coordination`
[14] The reader can download studio the from `http://gemoc.org/studio-download/`
[15] `http://gemoc.org/BCOoL/index.html`
[16] `http://eclipse.org/gemoc`

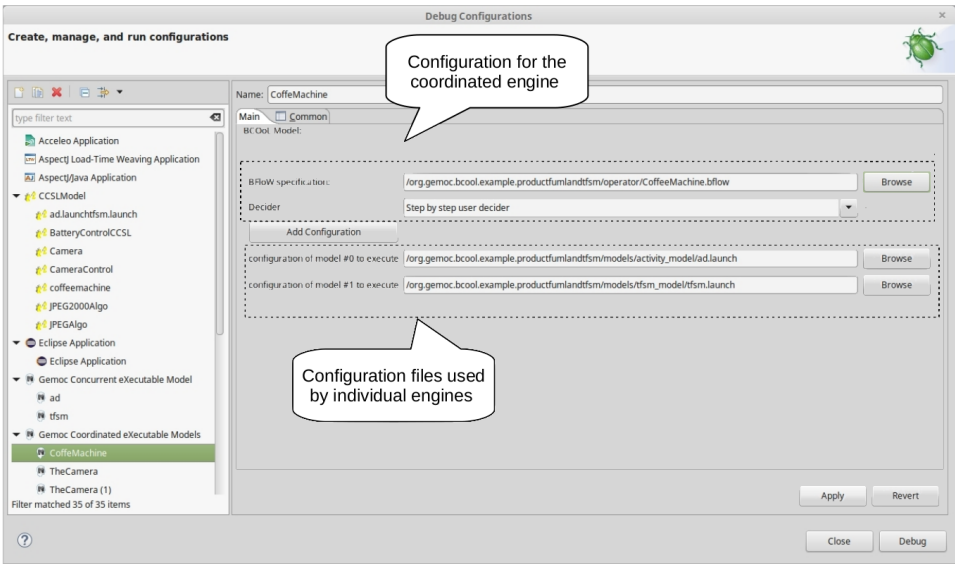

Figure 4.10: Configuration of the launcher of the Gemoc Coordinated Execution Engine

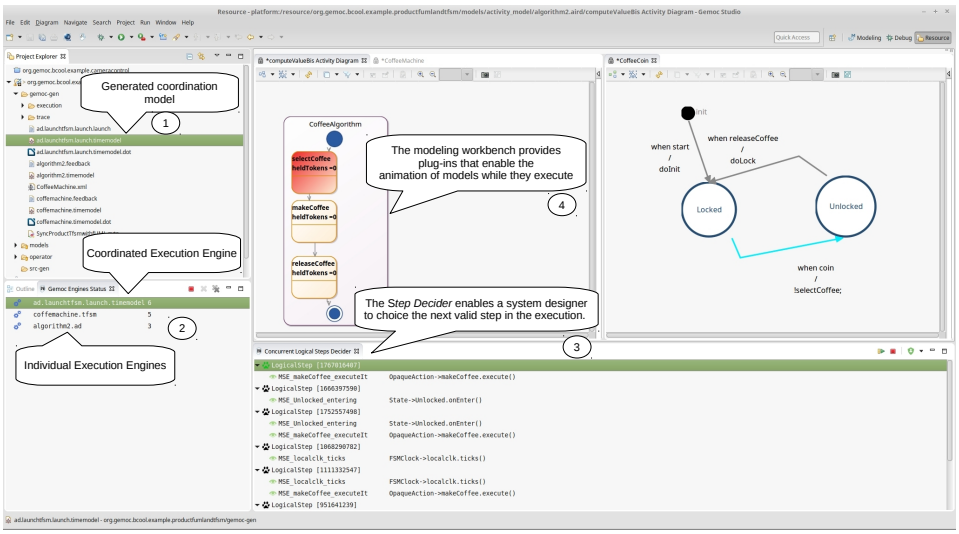

Figure 4.11: Coordinated Execution and animation of the models of the coffee machine (taken from [163])

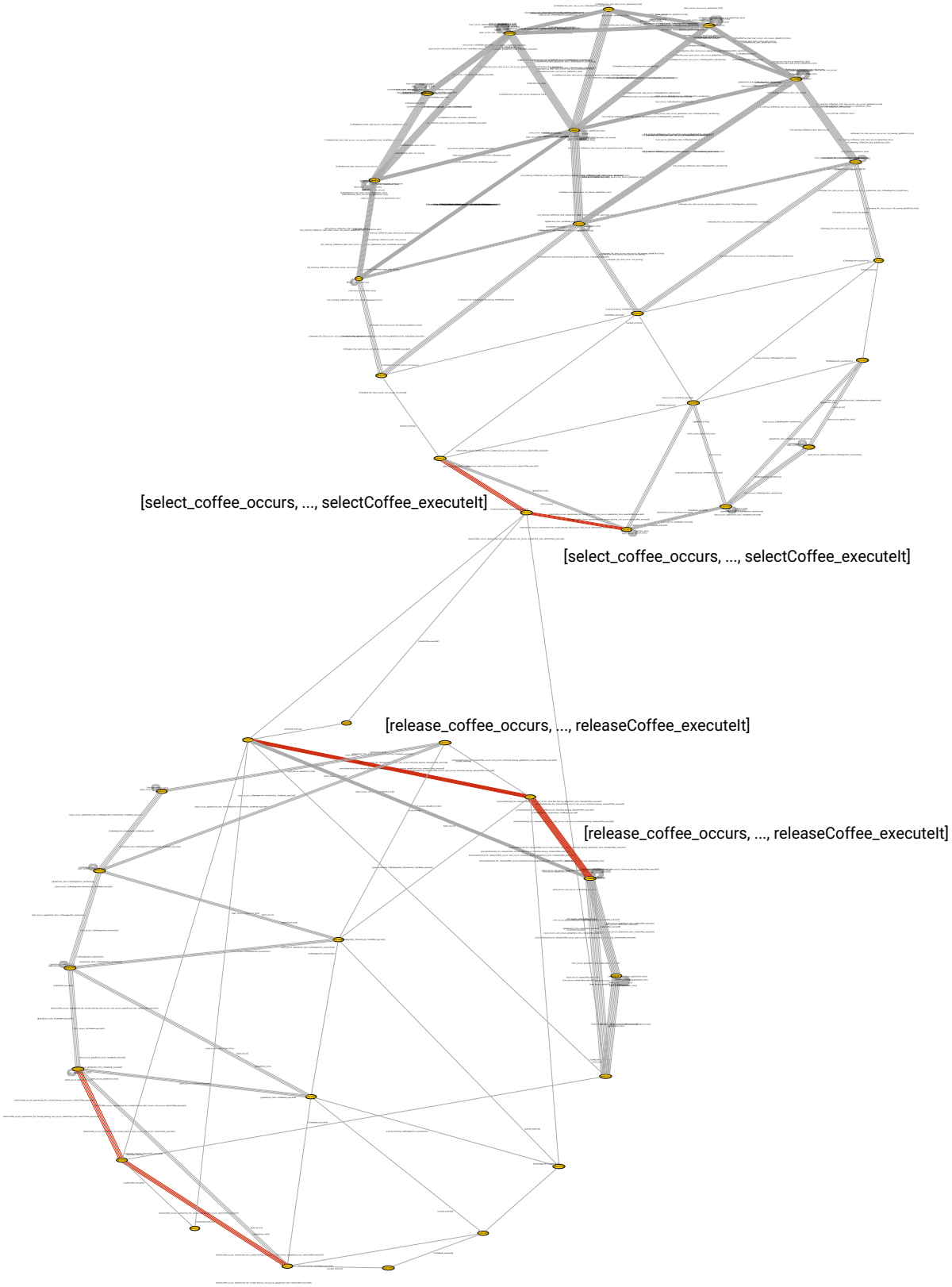

Figure 4.12: State space representation of the coordinated models of the coffee machine, encoding the set of valid schedules. The transitions in red represent the events forced to happen simultaneously by the coordination. (taken from [163])

ing models. Our workbench provides a support for simulation and analysis. Using B-COOL, the know-how of an integrator is made explicit, stored and shared in dedicated specifications and amenable to analysis.

Chapter 5

Conclusion

In the last ten years, I focused on the modeling of the formal operational semantics of languages, with explicit concurrent and timed aspects. Since the early development of CCSL when I arrived in the team, I quickly understood the benefits of logical time and clock schedules for the concurrent and timed specification of a model behavior (see for instance [53] early 2009). It makes explicit a symbolic (in intention) representation of all the acceptable simulation traces of a specific model. By augmenting the model syntax with such an explicit concurrency execution model it allows for reasoning on the concurrent aspects independently of (1) the language used for the model structure; (2) specific (non tractable) concurrency constructs offered by a specific interpreter technology and (3) the concurrency capabilities of the platform on which the model is simulated. This behavior model provides a golden set of acceptable behaviors, which any execution must conform to. It can then be used to ensure that a specific implementation of a model (obtained by automatic translation or manual encoding) provides traces that are conform to the specification. Finally, it makes explicit the synchronization points that can be used to coordinate different (possibly heterogeneous) models.

The behavior model usually depends on the structure of its companion syntactic model and on the behavioral semantics of the language used to specify the model. Consequently, if one considers a set of syntactic models written in the same language, there exist systematic rules that can be used to generate the behavior model for each syntactic model. This is the reason why we provided dedicated metalanguages to express these rules and that are amenable to the automatic generation of the behavior model for any syntactic model written in a specific language. During the development of the metalanguages, we were inspired by Structural Operational Semantics [149] but with a will to make explicit the, possibly timed and concurrent order in which the rewriting rules can be applied. This was done by developing MoCCML; which can be seen as a promotion of CCSL at the language level (we additionally made use of KerMeta, to weave the rewriting rules into the metaClasses of the abstract syntax of the language).

Based on such work, to face to multiplication of (modeling) languages, we though it was important to enable reasoning about model coordination at the language level. We took benefits of the architecture of the behavioral semantics specification to exhibit a language interface, *i.e.*, an interface exposing (partial) information about the different ingredients used in the definition of the language. Based on such interface, we proposed a language dedicated to the specification of coordination patterns. A coordination pattern is a way to capture, at the language level, the coordination rules that are usually applied by a system architect to consistently relate the execution of different models. Such coordination patterns are valid for any models in a set of languages and can be used to generate the artifacts used to coordinate the execution of heterogeneous models (*i.e.*, written in different languages). Since the behavior models of all coordinated syntactic models as well as the coordination between them are specified in CCSL, it is possible to reason about the coordinated behavior in an homogeneous way. This set of CCSL artifacts actually represents a specification in intention, of all the possible coordinations between the models, which respect (1) the behavioral semantics of the languages used to develop the models and (2) the semantics of the coordination patterns used to express contextualized correspondences between these languages.

During these ten years, we achieved great improvements in the modeling of executable behavioral languages. Following the same idea as grammar-ware approaches, which made great improvements by

providing metalanguages for syntax description amenable to automatic generation of various artifacts (*e.g.*, generation of advanced editors, parser and syntax checkers from an appropriate syntax description); we provided metalanguages for possibly concurrent and timed behavioral semantics amenable to the generation of various artifacts like, for instance, interpreters, model checkers, omniscient debuggers and model animator.

Despite our non negligible progresses on the specification and tooling of metalanguages for behavioral semantics specification of languages, we still lack some elements to cover the challenges exposed in the introduction of the this document. In the remainder of this conclusion, and before the perspectives, I list some of the limits I identified.

At the model level, we lack a way to embed the manipulation of data (*i.e.*, the rewriting rules of a specific model) in the behavior model. Instead, we captured (in a formal way) the acceptable order between the call to such rewriting rules and relied either on ad-hoc pieces of code or on the code defined in the concepts of the languages. Consequently, to execute the model we require the interpreter of the concurrency model and the definition of the rewriting rules (so possibly the definition of the language itself). This was a pragmatic choice since in our experiments we were always executing the models inside the modeling workbench. However, nowadays we would like to export our executable models as standalone modules and these last choices becomes an issue. we are however currently dealing with it.

At the language level, while MoCCML is formally defined, it is difficult to formally reason on the specification of the rewriting rules since they are specified in a non formal language. It indeed introduced limitation on the reasoning capability but we obtained lots of benefits as a counter part. For instance we were able to embed complex rewriting rules involving the call to a Functional Mock-up Unit[1] or the call to graphical data visualization API[2]. This was a good balance between formal reasoning capability, development facilities and user acceptance. Nevertheless, I think we should go further in the formalization of the rewriting rules to better understand/take advantage of the behavioral semantics properties of a language.

At the coordination level, the B-COOL metalanguage defines the formal coordination of model execution without supporting the formal coordination of data between models during the execution (data coordination can be added easily but it is difficult to ensure that the MoCC from the coordinated models are not modified in a non expected way by such coordination). We are currently actively investigating this aspect in the PhD thesis of Giovanni Liboni. Another important point related to the notion of pattern and challenge #2, is that we did not investigate yet the possibility to define patterns amenable to the automatic application of vertical correspondences. In other words, it could be interesting, based on a specific language behavioral interface, to define how different languages are related to each other with respect to a refinement process. It may for instance list the properties that are preserved from an abstraction level to another one.

Finally, From an acceptance point of view, we encountered three main problems. First it was difficult to let people understand the benefits of having metalanguages for behavioral semantics specification and how it is different from developing a specific language. Things evolved and I think it is now better understood. Second, we actually (December 2018) use 6 different metalanguages (some of them being formal) to develop a graphical executable modeling language with concurrent and timed semantics. While we (of course) have a rationale for that, it raises the entry ticket price for having your first executable language working. The third acceptance brake is the evolving technological background. We improved a lot the GEMOC studio along the years and it is difficult to keep up to date the various tutorials/examples we had. Despite all these three obstacles, we gained interest over the years and people understood the benefits of joining the approach (either as developers of executable models or to integrate their own metalanguage for behavioral semantics specification and consequently take benefit from our tooling).

Under the strong involvement of Benoit Combemale, the GEMOC became an eclipse research consortium[3] and the GEMOC studio is now an eclipse project[4]. One goal is to share the framework as an open research platform without the will to make an industrial product from it (some other previous tries

[1] `http://fmi-standard.org`
[2] `http://www.i3s.unice.fr/~hogie/software/?name=oscilloscup`
[3] `https://www.eclipse.org/org/workinggroups/gemoc_rc_charter.php`
[4] `http://eclipse.org/gemoc`

from academics have shown that it is a specific job to make software with a technology readiness level greater or equal to 6). Of course, there is no restriction for anyone to make a product of it by forking the project . The GEMOC studio is a research platform to experiment on the various challenges (formal) system/software engineering faces nowadays with the globalization of modeling languages.

In the last chapter, I elaborate on some of the various perspectives we have towards formal system modeling.

Chapter 6

Perspectives

Contents

6.1 Perspectives About CCSL evolutions/reasoning 103
6.2 Perspective About the Modeling of Structural Operational Semantics 108
6.3 Perspectives About Heterogeneous Modeling and Simulation 111
6.4 Thoughts About Formal System Engineering 114

In this chapter I list some of the perspectives to my work, which I'd like to investigate in the next years if I have sufficient PhD students and/or time and/or fundings and/or collaborations with other researchers. I tried to limit these perspectives to "short term" research questions, some of them possibly fitting a PhD thesis subject. Note that I kept only the perspectives on which we already worked/discussed a minimum but many others can be drawn since the field of system engineering is, according to me, still handcraft on many aspects (hopefully they rely on a useful and solid interface based design). After the presentation of these perspectives, I propose some thoughts of what may be a more formal system engineering ecosystem.

I kept in this chapter the same structure as in the document, starting with perspectives about CCSL as a behavioral model, followed by perspectives on the modeling of operational semantics to finish with perspectives on the coordination of heterogeneous simulations.

6.1. Perspectives About CCSL evolutions/reasoning

Many different works about CCSL are currently conducted by members of the Kairos team (but not only), on several aspects (*e.g.*, expressiveness, calculability, formal reasoning). In this section I draw some of the perspectives that are directly valuable for the use of CCSL as a behavior model, companion of the syntax model.

6.1.1. Exploitation of the CCSL clock graph

In CCSL, we are able to construct a clock graph that abstracts a CCSL specification and represents the different clocks and their relations (in simple terms like for instance SubClock, Precedes, Excludes, Coincides) according to the constraints used in the specification [51]. This clock graph is, for now, used only for manual debugging purpose, when the user tries to understand the relations between clocks in her/his specification.

This clock graph size is linear in the number of constraints and clocks in a CCSL specification (there are almost as many vertices as the number of clocks and between one to two edges by constraints). It is then cheap to construct but may be exploited for several purposes. For instance, it can be used to detect

bad patterns in the specification. A well known bad pattern is *causality loop* between some clocks[1], meaning that the clocks involved in the loop are deadlocking and can never tick during a simulation. Such deadlocks can be detected manually by simulation, or automatically by the construction of the state space (if finite) and verification of all the paths to look for the possibility for a clock to tick (based on classical model checking). However, it is computationally costly and possible only when the state space is finite. Conversely, checking causality loop on the clock graph can be done by some specific kind of *graph isomorphism matching*[2] by looking at any cycle in the graph that corresponds to the following regular expression: $< +(< (=)?)*$. The complexity of this algorithm on arbitrary graphs is known to be very costly in the worst case. However, the size of the clock graph is very small and can consequently, in practice, be checked instantaneously (could even be done in background during the edition of the CCSL specification).[3]

By following the same idea, we can detect other bad patterns, corresponding to more complex regular expressions to be detected in the clock graph. For instance, if there is no causality loop but a cycle like this one $a \boxed{\prec} b \boxed{\prec} c; a \boxed{\subset} c$ then there is also a deadlock and the corresponding regular expression can be specified (this is actually obvious since $a \boxed{\subset} c \implies c \boxed{\prec} a$).

More interestingly, such approach can also be used to detect potential deadlock, *i.e.*, specification in which there exist correct periodic schedules but where there also exist paths leading to deadlock. The simplest specification that illustrates this is given in equation 6.1.1.

$$a \boxed{\subset} \text{sup; b } \textbf{periodicOn} \text{ sup } \textbf{period} \text{ N; a } \boxed{\prec} \text{b.} \tag{6.1.1}$$

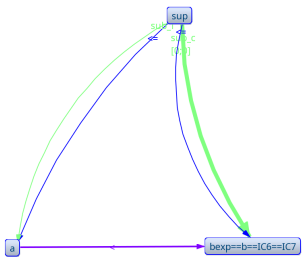

Figure 6.1: the clock graph corresponding to the CCSL specification 6.1.1

In this specification, if a does not tick before the N-1 tick of sup, then there is a deadlock (b must tick due to periodicity but cannot since a did not tick and a cannot tick without a tick of its super clock sup). Other paths lead to correct simulation. All these kinds of deadlock can also be detected by the same idea of graph isomorphism matching based on regular expressions. I already developed a prototype that successfully implements these ideas (see for instance identification of the problem obtained by the prototype on Figure 6.2: bold colored arrows represent the problem). However, I have, for now, neither formalized the problem nor proven that the clock graph is a correct abstraction with regards to the deadlock problem. This would be an interesting research project to provide advances static analysis of CCSL specifications.

Note that the clock graph could also be used to improve the simulation in TimeSquare. For instance, when two parts of the specification are loosely coupled, the clock graph can be used to detect it and create multiple Binary Decision Diagrams instead of a single one, reducing the computation of *irrelevant interleaving*[4].

[1] Note that in CCSL it is actually "precedes loop" which are problematic since in a causality loop all clock in the loop can tick simultaneously (see Section 2.2.6)

[2] see `http://www.i3s.unice.fr/~hogie/software/grph/doc/javadoc/grph/oo/ObjectGrph.html#findMatchingPaths-java.util.Set-`

[3] the reader can notice that the pattern is even more expressive than the *Precedes loop* announced since it takes into consideration any causality cycle with at least one *Precedence* in it

[4] irrelevant since they are not ordered at all and can be obtained by a post treatment if needed

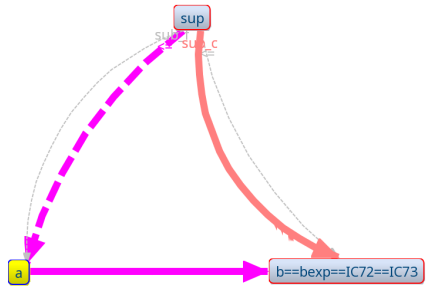

Figure 6.2: Automatic identification of the potential deadlock in the CCSL specification 6.1.1 (proof of concept)

It may also be used as a basis for generation of concurrent code, as already experienced in Signal [96].

Finally, it may also be used to simplify the CCSL specification before its use for model checking. Let consider that a user wants to construct the state space of a specification, looking for possible deadlocks. In this case, based on the clock graph, we can detect and remove clocks and constraints that are irrelevant according to the deadlock property. It may also make the state space bounded while it was originally not. For instance, let consider the following specification (a modification of the previous one):

$$a \boxed{\subset} \text{sup; b } \textbf{periodicOn} \text{ sup } \textbf{period} \text{ N; a } \boxed{\prec} \text{b; newClock } \boxed{\prec} \text{b} \tag{6.1.2}$$

In this case, since $newclock$ is a *source*, than it can be safely removed without interfering on the result of the property checking (informally, it is not constrained, so that, even if it can disable b, it can always be added before b and will never introduce any deadlock). More than simplifying the specification, it also makes it bounded so that the state space can be constructed for it. Once again, formalizing all these properties must be done in order to ensure the correctness of the graph abstraction/manipulation with respect to some property. Note that, related to the part of this perspective, the prototype we developed identified the $newClock$ as non relevant in the potential deadlock (see Figure 6.3)

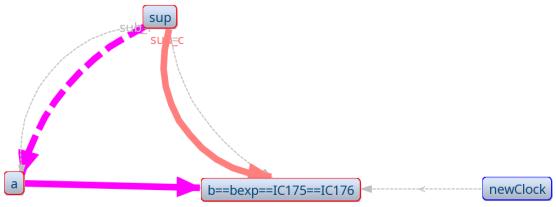

Figure 6.3: $newClock$ is considered as non relevant in the identification of the potential deadlock by the prototype

6.1.2. Extension transition system and/or temporal logic for CCSL model checking

In CCSL, we can construct the state space (if finite), which represents the set of all the possible clock schedules. We can export it as a Labeled Transition System (LTS), for instance in the Aldebaran format [5] for model checking purpose. However, since a CCSL specification can allow several clocks to tick synchronously, the *Labels* in the resulting LTS does not always represent a single clock name (that ticks), but can represent a set of clock names (that tick synchronously). For instance, lets consider the following specification: $a \boxed{\sim} b; c \boxed{\subset} b$; The corresponding "synchronous" LTS is then the following:

In this case, if we want to model check the CCSL specification against the following LTL formula: $\Box a \Rightarrow \Diamond b$ (Always, a implies eventually b) then the answer will be $True$ only under a fairness hypothesis while it is actually always $True$ (since the $b.c$ transition is not understood as being b together with c).

[5] `https://www.mcrl2.org/web/user_manual/language_reference/lts.html#aldebaran-format`

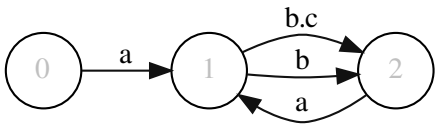

Figure 6.4: The "synchronous" LTS computed in TimeSquare from $a\ b; csubOf\ b;$

In order to allow model checking synchronous specifications, there is the need to extend existing tools so that they interpret correctly "Synchronous" LTS, *i.e.*, LTS where labels can specify a conjunction of possible labels during a single transition. In collaboration with Ciprian Teodorov and Joel Champeau, we proposed an "unfolding" of the LTS so that synchronous labels are split in all their possible interleavings (resulting in the LTS Figure 6.5). However, this is costly as soon as the labels contain many synchronous clock ticks. Moreover, it is impossible to explicitly speak about clock synchrony in the property language.

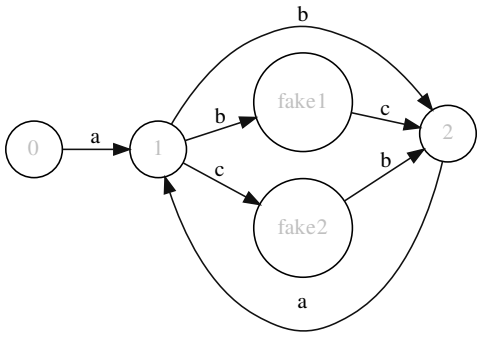

Figure 6.5: The unfolded "synchronous" LTS

A potential solution to this problem is to redefine the synchronous product of automata so that the *match* between transitions take the synchronous aspect into account (*e.g.*, label.split('.').contains(b)). This kind of extension has been proposed for instance for model checking LTS where labels are a complex expression that must be interpreted [129]. By using this approach we were able to use the MCL language to model check some properties over a CCSL specification. However, to make it easier to handle, the semantics of the temporal logic should be adapted consequently. In our case, it means that $\Box a \Rightarrow \Diamond b$ actually signifies that b can occurs synchronously with any of the clocks from the specification. What is missing is a way to speak explicitly about specific synchrony/exclusion directly in the temporal logic. For instance it should be possible to define the two following properties. 1) $\Box a \Rightarrow \Diamond b.c$, which enforce b and c to be synchronous (being wrong in the unfolded LTS) at least once after a a but that does not restrict them to occur synchronously with any other clock from the alphabet or to occur separately at some (other) steps of the execution. 2) $\Box a \Rightarrow \Diamond b.\bar{c}$, which enforces b to be present and c to be absent at least once after an a but that does not restrict them to occur synchronously with any other clock from the alphabet or to occur synchronously at some (other) steps of the execution.

6.1.3. Multi Physical Dimensions Associated to Simulation Speedup

In system engineering, it is common to refer to variables representing evolution along different physical dimensions like *e.g.*, some angles, forces, pressures, distances and of course time. Such variables

usually evolves continuously and using clocks (or signals) associated to a tag is a well founded way to access to some of the values of these variables. These variables are usually constrained by physical laws, the plant, or the way the control is realized. During modeling, specific values of these variables, accessed through some clocks can be used as a referential for other events in the system. Such mechanism is for instance supported by the MARTE profile, which enables the possibility to define various physical dimensions and to define clocks, that represent variables associated to such dimensions. These clocks can on the one hand be constrained to ensure the consistency between the different dimensions, and on the other hand be used to specify some other clocks in the model. For instance, it is possible to specify that the ignition in a cylinder occurs at most 10µs after the crankshaft reaches angle 2.8 radians. Here, two variables referring to physical dimensions are used (the physical time and the crankshaft angle). To make the specification consistent, it is necessary to add a relation between these variables, obtained from a knowledge of the plant. For instance, if the angular velocity is constant[6]: ($crkAngle = \frac{2.\pi.RPM}{60}.t$). For a behavioral model to be CPS ready, it is important to support such mechanisms.

In CCSL, only a part of the MARTE profile is supported. It is possible to specify various physical dimensions, to define clocks discretized from a physical dimension and to constrain the discretized clocks. It is however not possible to define relation directly on different dimensions so that tags of the corresponding discretized clocks are kept consistent during a simulation of the model. The goal here is certainly not to equip any clock with a tag on which arbitrary arithmetic can be done (like in Signal [19]) but rather to characterize some clocks, defined from the discretization of a physical dimension, on which some rules on the evolution of their tag apply. For instance the evolution of the physical time tag is monotonous and increasing, the evolution of tags representing the angle of a crankshaft is monotonous and increasing but its values are provided $modulo\ 2\pi$. Such information should also be handled in CCSL since they can be used as reference for control.

By adding such notion of physical dimension relations directly in CCSL and the associated tooling, the associated tags could be used for different purpose. First, in CCSL, we nowadays discretize physical dimensions periodically, and associated tags are used only for user feedback. However, the variables representing the evolution of physical dimensions may be accessed at any time (not necessarily periodically) and providing relations directly based on tags is a way to define in an intuitive way an order (possibly partial) between specific tags of these clocks during the simulation (this is for instance what was done in the camshaft example: “the ignition in a cylinder occurs at most 10µs after the crankshaft reaches angle 2.8 radians”). Second, when heterogeneous models are coordinated, we currently base our coordination on the ordering of specific events by considering their instants. With an explicit specification of variables obtained from physical dimension, and a notion of tags over the clocks used to access them, it makes possible to coordinate with other (possibly heterogeneous) models as long as clocks used to access the same variables can be ordered. A classical example is to totally order tags of two clocks from two different models both based on the total order relation implied by the underlying physical dimension (*e.g.*, two related computations on two computers on earth, implicitly linked by the physical time). Third, in CCSL we reason on instants obtained from the periodic discretization of variables representing evolution of physical dimension. This introduces bad performance. For instance, consider two periodic clocks based on the same physical time (of a same referential, let’s say earth). The first one is periodic with a period of 97ms while the period of the second one is 67. In this case, the only common clock that can be used as a base clock is a clock that discretizes the physical time with a precision of 1ms. It means in CCSL that all instant of this clock will be computed only for counting when other clocks occurs. By using tags, we can obtain a speed up in the simulation by avoiding the enumeration of all instants that are not required to be computed will providing the exact same total order. This last point can be exploited only on specific conditions, like for instance when different clocks are accessing in a deterministic way to a same dimension. In this case, reasoning on tag relations is enough to speed up the simulation and avoiding the possibly numerous ticks of a clock discretizing the physical dimension with the greatest common divisor. It however requires a correct integration of such mechanism in the operational semantics of the CCSL language.

[6]For more complex relations, where the angular velocity is not considered as constant, it can be bounded, or given by an external process (*e.g.*, a black or white box coordinated model) to simulate a specific scenario.

6.2. Perspective About the Modeling of Structural Operational Semantics

In this section, I draw perspectives about the concurrent and timed modeling of structural operational semantics.

6.2.1. Formal specification or abstraction of rewriting rules

As explained in chapter 3, in the definition of the operational semantics of a language we did not use a formal language for the definition of rewriting rules (named DSA). On the one hand, it disabled a full formal reasoning over the operational semantics of the language and/or over the behavior of a model. On the other hand it allowed a better integration with external (java based) libraries, allowing for advanced examples (like ones embedding FMUs or equation solvers) and better user feedback (for example by linking with plotting libraries and providing visual feedback on the evolution of some Run Time Data (RTD)).

In order to make the operational semantics formal, we need to replace the language used for the rewriting rules by a formal language, and to understand the link with the language used for the MoCC (which is already formal). Several languages are candidates to specify the DSA but we are currently initiating a light collaboration with Akram Idani[7], who is working on a modeling framework [8] where the rewriting rules are defined in B [2]. Based on pre/post conditions, it limits the acceptable order in which the rewriting rules can be called. By mixing his approach with our approach, we could define the rewriting rules in B and drive the calls to these rewriting rules by the concurrency model. We could then take benefits from all the B tooling, for instance to ensure the compatibility between the MoCC and a set of rewriting rules. It may also be used to detect non determinism hidden inside rewriting rules; or to compare different execution states. To go beyond these first ideas, some research activities must be done to precisely understand the link between the two formal languages used to specify the MoCC and the DSA. This requires formalizing the communication used between the MoCC and the DSA (as already investigated in [114]). This would eventually lead to formal reasoning over the whole operational semantics, including refinement checking, according to some properties, between the behavior of different, possibly heterogeneous, models.

If formalizing the operational semantics is an important step, it is however important to take benefits from the calls to external, possibly complex, java-like actions (when a rewriting rule is executed). There are multiple ways to handle this depending on the role of the java-like actions.

First, if a java-like action is used to provide feedback to the user (like for instance data plotting), then this action usually does not depend on a specific semantics but on the RTD. In this case, it should probably be generalized and provided once and for all in a dedicated GEMOC add-on (see section 3.4.5 for examples of existing add-ons). This way, any language could take benefit from the user feedback.

Second, when the java-like actions are used to call some specific libraries (*e.g.*, FMI or ILP), it is then important to characterize such calls and their impact on the RTD. Then, these calls may be abstracted by using (B) pre and post conditions so that it ensures minimal reasoning capabilities. Any black box action conforming to these pre/post conditions are then considered as correct. We could for instance check at runtime (or in some cases by static analysis) that the "black box" java-like actions are not violating their abstraction.

6.2.2. Clarification of the links between various "views" over the semantics

When dealing with a model or program, written in a specific language, there usually exist different *views* of the behavioral semantics of the language. For instance, let consider that a language is specified by a syntax and the definition of a structural operational semantics: $< Syn, SoS >$. The SoS can be specified in various ways, ranging from informal documens in natural languages to formal specifications like the

[7] http://lig-membres.imag.fr/idani/
[8] `http://vasco.imag.fr/tools/meeduse/`

one proposed by Plotkin [149]. If this specification is not directly executable, then an interpreter must be implemented (based on the definition). It is important that the interpreter satisfies the definition of the SoS. However it can introduce some restrictions during the implementation. For instance, a *fork* statement can be defined in the SoS as allowing all the possible interleavings of its consecutive statements (until a *join*). Some interpreters correctly implement a subset of this semantics by choosing a specific order in the execution of the consecutive statements (for instance choosing the order of the actions in the document)[9]. Such implementation of the interpreter is correct in the sense that all the traces produced by the interpreters are valid traces, *i.e.*, acceptable by the semantics. It can however lead to undetectable bugs event if strongly tested. For instance if the system if tested in a different context. For instance when using a compiler with a more permissive semantics or when using another interpreter. In some case, a different execution trace can be obtained after simple copy/paste, which changes the order of the actions in the document.

In order to better apprehend such phenomenon, we advocate the use of a golden executable semantics (as formal as possible). However, even in this case it seems important to better understand the relation(s) between different software with respect to this semantics. A better (formal ?) characterization of these relations may be of great help to pin point potential problems/point of attention when choosing a specific tool/technology.

Interestingly, the same kind of phenomenon appears when controlling the simulation of a model behind a specific API, common to various languages (for instance when using HLA or FMI for co-simulation or older *software bus*). Indeed, we have shown in [38, 121] that using a time triggered API to control the simulation of cyber models can lead to a *blurred* resulting semantics, introducing inaccuracy and bad simulation performances. This is usually due to the injection of temporal inaccuracy due to the homogenization of the control API. Once again it seems important to be more systematic in the characterization of the effect(s) of a specific API on the relation(s) between a program and its semantics. This should help to better understand (co-)simulation results.

Such phenomenon is well known in the domain of differential equation solving. Since there is no exact SoS for differential equations, there exist only approximations of it. However, based on advanced structural analysis [21] and/or mathematical techniques, it is (for some classes of differential equations) possible to precisely bound the error introduced by the approximation. The proposition is then to explore the possibility to create the formal context and the technique appropriate to do the same kind of analysis in the context of cyber models. The relations between a golden semantics and the interpretation done by some implementations could then be precisely characterized. More ambitious, based on such relations it may be possible to automatically infer the set of properties that are preserved/checkable after the choice of a specific tool.

This perspective is highly exploratory but may have an interesting impact in 1) the understanding of implementation choices when encoding an interpreter/compiler and 2) the acceptance/validity of co-simulation results.

6.2.3. Integration of the various metalanguages used for SOS modeling

From a practical point of view, we are using three different metalanguages for the specification of the operational semantics of a language (MoCCML mapping, MoCCML constraints, KerMeta). This is because we used the most appropriate meta language for each of the "ingredients" of the operational semantics. We obtained some benefits by doing so since the metalanguages came with the facilities appropriate to their usage. For example, the MoCCML constraint language, by its declarative nature, provide a natural and elegant way to refine the MoCC by addition of constraints. This has been used for instance to specify the deployment of an application on a hardware architecture or to ensure that the coordination of a model does not introduce new behavior in the coordinated models. Additionally, it allowed us to reuse the existing tooling for each of the different metalanguages. However, because the different meta languages follow different formalisms, it makes it difficult for newcomers to define/understand the resulting semantics.

[9]This is a classical behavior (even if dangerous), found for instance in the Moka fUML interpreter (`https://wiki.eclipse.org/Papyrus/UserGuide/fUML_ALF_Moka`) or in the SCXML reinterpretation of State Charts (`http://www.w3.org/TR/scxml/`)

To help in the definition of the operational semantics, these different ingredients should be specified in an seamless way, *i.e.*, with no (or few) context switch overhead due to the formalism change. This can be done in (at least) two different manners.

A first one consists in providing one single meta language, able to specify both the rewriting rules, the concurrent and time way these rules can be scheduled, and their mapping to the abstract syntax. While this may be possible to define such a meta language, one has to take care to keep the benefits introduced by each of the metalanguages; both at the specification level (an appropriate formalism) and at the runtime level (with appropriate tooling/reasoning facilities). A side benefit of such approach is that it requires to enumerate the properties of each meta language that makes it appropriate to the specification of a particular ingredient. Once everything is clarified, redundant roles can be removed and it may be easier to figure out a language suitable to the definition of a (formal) concurrent and timed operational semantics. This is however (1) not sure that a suitable formalism exist, and (2) it requires a lot of work to provide the tooling facilities equivalent to the one provided by each of the existing metalanguages.

A second way to help the user defining its operational semantics is to provide editing facilities that decrease the distances between the different meta languages, while keeping them and their tooling under the hood. This would avoid switching from one technical environment to another. This is for instance what the ALE language [10] does by allowing the definition of the DSA in the same environment then the abstract syntax. Behind the scene, it creates a dedicated file storing the required information and the operation definition. More generally, the idea is to provide a consistent view of the operational semantics definition (or even of the whole language definition) despite the use of several metalanguages. It may be done for instance by using the notion of *viewpoint* provided by Sirius or the facility to mix different concrete syntaxes provided by tools like JetBrains MPS[11].

If this perspective is not directly linked with scientific questions about operational semantics, not dealing with it may be a blocking point for communication, user acceptance and education; consequently limiting the scientific impact.

6.2.4. Helping Writing/Debugging The Operational Semantics

Defining a concurrent and timed operational semantics can be complex. In the GEMOC studio we provided facilities so that the language developers can have a quick feedback about the operational semantics they are developing (*e.g.*, by running example models). However, the declarative nature of the MoCC can make it difficult for a user to understand the semantics resulting from the conjunction of all the constraints. There are multiple ways to help the language developers in the definition of the operational semantics.

First, when he is executing a model to check the correctness of its semantics, it may happen that some actions are forbidden while they should be allowed by the semantics. In this case, it may be possible for the user to enforce such action and to have a feedback on which constraints are violated in this case. This would be of great help for correcting the MoCC but requires non trivial manipulation of the underlying Binary Decision Diagram used in TimeSquare. This manipulation should be reversible so that the "initial" execution can continue.

Second, during the execution of a model it may happen that some actions are allowed while forbidden by the semantics. In this case, it should be possible for the language developer to manually disable the corresponding actions and to have a feedback on this impact on the semantics. Note that a crude version of this possibility has already been implemented in the GEMOC Studio where disabling a specific clock modifies the BDD and recomputes the new possible futures. However we are missing two important features. The first one should support the language developer that wants to remove a logical step rather than a clock itself from the solution space. For instance he may want to remove a logical step where two clocks are synchronous, but keep the logical steps where clocks occur exclusively. The second missing behavior is a help on how this logical step removal can be done by the addition of a

[10] `https://github.com/gemoc/ale-lang`

[11] `https://www.jetbrains.com/mps/` or more specifically `https://www.jetbrains.com/mps/img/screenshots/2017.2/language-support.png`

constraint in the MoCC. In some specific cases, a simple constraint can be automatically added to disable some logical steps (for instance by adding an exclusion between two clocks). More generally, this can not be deduced automatically but all the constraints that relate to the undesired logical step could be emphasized to help the language developer figuring out the missing constraint(s). This emphasize could be based on an adaptation of the clock graph; where only relevant clocks are presented and where new constraint could be draw to provide a test and error way to reason on the constraints.

Third, we can also imagine helping the language developer to write (a first shot of) its semantics. For example, we may ask a user to define some representative case studies and propose a way to manually "play" the acceptable traces for these examples. Then, it may be possible to infer (at least) a base definition of the MoCC. For instance it may be possible to infer an untimed version of the MoCC by checking when the causalities between two actions are always true for a specific structural pattern in the model. This may also be done by trying some well known patterns extracted from existing MoCC definition (*e.g.*, fork concept, join concept, exclusive (or not) predicate based branching concept).

Finally, if we go even further, it may be possible to infer some information about the MoCC aspect of an operational semantics by learning on the acceptable traces of various different models. This is actually what most developers are doing, instead of learning the operational semantics of a language by reading the specification (when available), they learn from testing and practicing with different programs. To formulate it more appealingly, if we consider that each constraint of the MoCC is a property of the semantics then, extracting properties of the operational semantics of a language from the deep learning of traces obtained from models, may be a very interesting topic when it is required to make explicit an implicit behavioral semantics (for instance encoded into a tool). Of course extracting the concurrent and timed aspects may be challenging and strongly rely on the test set. However, automatically providing a partial semantics may be better in many cases than a purely black box one.

6.3. Perspectives About Heterogeneous Modeling and Simulation

6.3.1. Support for Data Coordination

As presented in chapter 4, we proposed, based on a language behavioral interface to define coordination patterns from which the orchestration between different, possibly heterogeneous, models can be automatically generated. It is then possible to interpret the specification of the orchestration to co-simulate the different models.

The patterns allow to generate the synchronization between the different model executions, however they do not define how the data are exchanged between the different models. We did not support data exchange since we had the will to define correct by construction orchestrations in the sense that, it is not possible to introduce a new behavior in a model due to its coordination. In other words, any behavior of a coordinated model belongs to the set of behaviors defined by its execution model. Also, in BCOoL, we wanted the coordination to be non intrusive, so that a model did not need to predefine some events as being inputs or outputs.

In order to allow data coordination, it is important to make choice on the properties we want about the coordination. It is for instance impossible in the general case to ensure that no new behavior will be introduced since allowing a data to be set from the outside could modify the internal control of the model, driving the model into an execution path that was not accessible before.

After a clear definition of the properties that are expected by the coordination, it is important to be aware of the reading/writing time of such data. According to our definition of the operational semantics, it means that the MoCC defines specific read/write DSE for each data. This means that the manipulation of data by using rewriting rules can not be left hidden in the rewriting rules and must be exposed through the use of the corresponding DSE. On the one hand it is safe since in this case the MoCC defines the exact points in time when data can be set/get but on the other hand it increases in a significant way the complexity of the MoCC. Additionally, from a technical point of view, adding all these DSE introduces a lot of new DSE that may be only loosely constrained. For instance if some data are only *set*, then it means that the DSE associated to the *get* of the data is free to occur at any time (except perhaps synchronously with the set). Since the concurrency model represent at each step of the

simulation all the possible execution paths, it means that any model having N non constrained MSE have, by definition 2^N possible execution paths, without even considering any of the "relevant" paths. This unfortunately introduces (artificial) computational explosion.

An important enabler to allow a safe coordination of data between heterogeneous models is to correctly handle the link between the MoCC and the data manipulation so that the impact of the coordination can be clearly understood while avoiding artificial complexity for the designer and during the co-simulation of heterogeneous models.

One potential idea in order to handle non intrusive coordination without computational explosion is to (automatically) extract (*e.g.*, by static analysis) the inputs/outputs of a model. This would allow the coordination of models whose language is not equipped with the notion of input/output. These notions are then properties of the models rather than properties of the language. Of course, if the language already defines these notions, we can exploit then naturally. With such information, we can for instance forbid coordinations that require the modification of output data, so that it avoids potential race conditions between the internal semantics of the models and the coordination itself.

I am currently advising a PhD thesis (started less than one year ago) on this subject and we are confident to introduce the appropriate mechanism to provide both a sound data coordination and no significant performance overhead due to data coordination.

Finally, it is interesting to note that data coordination is actually strongly linked to the notion of master algorithm in the co-simulation domain. We already published preliminary work that highlights the need to better understand the internal data manipulation of a model to drive an accurate and efficient co-simulation [38, 121]. We believe that we can have an interesting insight in co-simulation by taking advantage of 1) language specific semantics particularities (like for instance delays due to computation of cyber models) and 2) the definition of an explicit coordination model to allows the synthesis of a dedicated master algorithm, based on the behavioral characteristics of models and their coordination (once again, see [121] for preliminary results on this subject).

6.3.2. Understand the models "Greyification" for Correct and Formal Coordination

In all of our previous works, we were advocating a complete knowledge about the model syntax and semantics in order to propose definition of coordination patterns and eventually the co-simulation of heterogeneous (cyber) models. It was important to first understand the mechanisms and information required for a correct coordination of model executions.

When considering industrial context, there is a need to protect the Intellectual Property (IP) of models so that they can be exchanged (*e.g.*, between suppliers) without revealing industrial skills implied in the realization of the models. This is the reason why *de facto* standards like FMI hide the models and their simulator behind a common interface from which the model can successively *do steps* for some duration and their state can be queried or set at specific points in time between two steps. This is a pragmatic solution but it does not ensure accurate simulation results [38, 121].

In order to make a correct coordination of the different black box models, the algorithm in charge of coordinating the models under execution must be aware of the semantic specificities of each models. For instance, coordinating a model based on differential equations is different than coordinating a model representing a periodic computation on a computer; which is also different than coordinating a model representing a reactive application. Exposing such differences is not violating the IP property, however, it is not clear, yet, how much we can/need to expose from the different models.

In order to ensure the correctness of co-simulation, we may explore two different ways to greify the models under co-simulation.

6.3.2.1. Greyification strategy #1

The first way to greyify the models for co-simulation consists in exposing some information on the entities, which is relevant to the synthesis of a master algorithm dedicated to the topology of models (*i.e.*, a st of executable models and connectors between their input/ouputs) and whose behavior depends on the properties exposed by each of the models.

In first work on the subject initiated with Claudio Gomes, we identified three important kinds of information: the usage made by a model of its input data, the capability of the executable model with respect to the co-simulation and the responsibilities, which are assigned based on the usage, capabilities and the model topology either to the master algorithm or to the models.

For instance, a black box model should expose the input usage. An *input usage* provides few but enough information of what is expected by the model, like, for instance the following ones:

ZCD this usage means that the input is used to perform a Zero Crossing Detection, and then that the entity is interested in a precise temporal localization of the crossing.

trigger This usage means that a new valuation of the input leads to the triggering of the internal entity behavior, usually including reading of its inputs and further production of its outputs. This is interesting since without new valuation, the entity does not need to be simulated.

sampling This usage means that the input will be internally sampled at a given rate; there is consequently no need to feed this input except on reading.

A black box model should also expose its capabilities. A *capability* denotes a specific behavior of a model, associated or not to an input or output, which is related to co-simulation.

Examples of capabilities associated to an input are:

reject if unexpected change this capability means that the model is capable to reject an input valuation if the difference with the previous valuation does not allow him to guarantee a correct behavior. This is for instance the case when an entity is using an input to do a Zero Crossing Detection (ZCD). In this case, the entity is actually trying to detect the actual time at which the input value is crossing a specific value. Then, it may found out that the difference between two successive inputs is too big to precisely enough localize the zero crossing. In this case, it will reject the step.

extrapolation the capability of a model to do extrapolation on the given input. This is of first importance to correctly feed the entity.

interpolation the capability of a model to do interpolation on the given input. This is of first importance to correctly feed the entity.

understand discontinuity location This capability means that the model accepts to be notified that the input had a discontinuity. This is important since if this capability is associated to a ZCD, then no useless rollback is done.

Based on the different capabilities and a specific topology (*i.e.*, set of interconnected models), it seems possible to elaborate a dedicated master algorithm, which implements the appropriate mechanisms according to the allocation of so called responsibilities. For instance, if a connector links an output the capability to do a discontinuity location to an input with a zero crossing detection usage and with no rollback mechanism, the master algorithm should not behave the same than if the input does not use the input for zero crossing detection.

All this work still needs to be better studied and understood but it seems interesting to tame co-simulation. This is also interesting to see that the FMI de facto standard is shyly following the same direction and proposed to adopt two of the recommendation we made in [38, 121], namely conservation of the history between two communication points and explicit specification of event arrival (typically for periodically executed systems). This seems however not at all attributed to our work.

6.3.2.2. Greyification strategy #2

We are also working on another way to expose information on how to co-simulate heterogeneous models. This second methods is more linked to the work presented in chapter 4. In previous works, we proposed an extension of the FMI API with a specialized version of the doStep method dedicated to cyber models. The idea was that original time driven doStep was not appropriate to simulate cyber models where it is more accurate to simulate until a specific event occurs (*e.g.*, a read on an input and/or a write on an output).

It may be interesting to generalize this idea, which considers that different models need to communicate differently with the master algorithm depending on both their behavioral semantics and their connection(s) to other models.

A first idea consists in proposing an API with a doStep function that takes a predicate as a parameter. The predicate should be defined in a dedicated language, whose models are parameterized by a language behavioral interface. More precisely a predicate for a model written in a language L should speak about elements the language behavioral interface of L, applied on the model.

The language behavioral interface should provide information about the event and runtime data that may appear in the conforming models. The models should also expose some (input and output) ports. When someone calls a doStep(10ms) (*i.e.*, doStep(Δt)) on a model, it is because he intuitively knows that the model is executed according to the physically time. If we consider quantized state systems (QSS) methods, where the system is discretized based on quantum of the system space instead of discretization of the time, than one may call a doStep(quantum on v = 1cm) (*i.e.*, doStep(Δv)). Once again this is based on the knowledge about the semantics of the solver, and the knowledge about a v variable. By extension, with an appropriate knowledge on a language semantics, one may do doStep(sup(input1, output2)) (*i.e.*, doStep($eventoccurrence$))[12].

This idea is challenging at many levels ranging from the identification of the appropriate information to put in the language behavioral interface to the understanding of how a model wrapper can be implemented on the basis of a predicate.

6.4. Thoughts About Formal System Engineering

There is still a long road towards formal system engineering. In order to pave this road, we need to rely on a more systematic understanding of the concepts manipulated in each engineering disciplines without enumerating all of these disciplines. Of course, the first step in my honest opinion is to make explicit the semantics of a language. We saw that the behavioral semantics is a key point but it may not be the operational semantics. Generally speaking, we need to specify the behavioral semantics in a tractable way, so that behaviors of models are clearly understood. We may also think about other, additional, descriptions of the semantics like for instance ontologies, that could make it easier to align concepts from a domain to another.

In all the cases, we should try to avoid tooling directly specific languages and we should learn how to reason on meta languages. By tooling the meta languages, we can reason on their instances, *i.e.*, on the description of a language, so that any new language can benefit from the tooling. This is for instance what we did for L_{sem} in GEMOC (see top left of Figure 6.6). Using a same meta-language to define different languages also bring benefits when it is time to compare them. For instance, in our previous work we can imagine a comparison of the behavioral semantics from two languages by relying on L_{sem}. If both languages are described by using different meta languages (even sometimes the natural language), then comparing their semantics is tedious and error prone. Also, such meta languages should be well adapted to their purpose. For instance we may use Mathematics as a common meta-language for many activities, but being very expressive, it makes automatic reasoning more difficult and we fall back to the same pros/cons than when comparing the use of a DSL and a GPL.

Also, if we want to rely on a more systematic understanding of the concepts manipulated in each engineering discipline, we need to relate the meta languages themselves. For instance, if we consider that we can obtain a *simulator* tool based on the L_{sem} meta language, which can produce traces when used with a specific semantics and a model (*i.e.*, a syntax) as parameters, then this trace should be instance of a model which is itself related to the meta language used for the semantics (see top left of Figure 6.6). This is also what we did in GEMOC, were traces speak about occurrences of events, themselves typed by DSE part of the semantics and defined in L_{sem}. By doing so, it was easier to know the important point of interest in a trace and we were able to compare a trace from a tool to the original semantics (see an implementation of it based on the Capella language here: `https://youtu.be/ESIX2PFGiDU`). These relations between meta languages should be generalized to ensure a better consistency of a system

[12] where sup is the CCSL operator from chapter 2

engineering ecosystem[13].

On Figure 6.6, I put some of the meta languages and relations between them that may be important in the context of a consistent system engineering. Of course it was not possible to put all of them or to be precise enough without having a too complex Figure[14]. On this picture we can see that, ideally the tooling is never done on a specific language but always on specific meta-language. We can differentiate two kinds of tools. The first one, like the simulator already presented above, is defined independently of a specific language but its goal is to reason on models written in a specific language. Consequently, its parameters are both a part of the language definition and a model. Other examples of this type of tool is *tool5*, defined with respect to a language for horizontal patterns ($L_{h-pattern}$). It takes a pattern and two models in order to generate a set of horizontal correspondences between the two models. This correspond to a part of the tooling realized in B-COOL. The other type of tools are used for reasoning on languages. For instance *tool2* may be a tool to compare two semantics. As such it takes two semantics as parameters and may, for instance, say if a semantics is a refinement of the other or not. Also, while not presented in this document, Melange [59] is one of such tool, that allows to obtain a languages by the composition of different (parts of) Languages. In the context of system engineering, such tools may be used to compare language interfaces (*e.g.*, *tool3*) or to abstract a language interface according to specific criteria. For instance *tool4* may be used to extract from a very permissive language interface a language interface enforcing the notion of input/output.

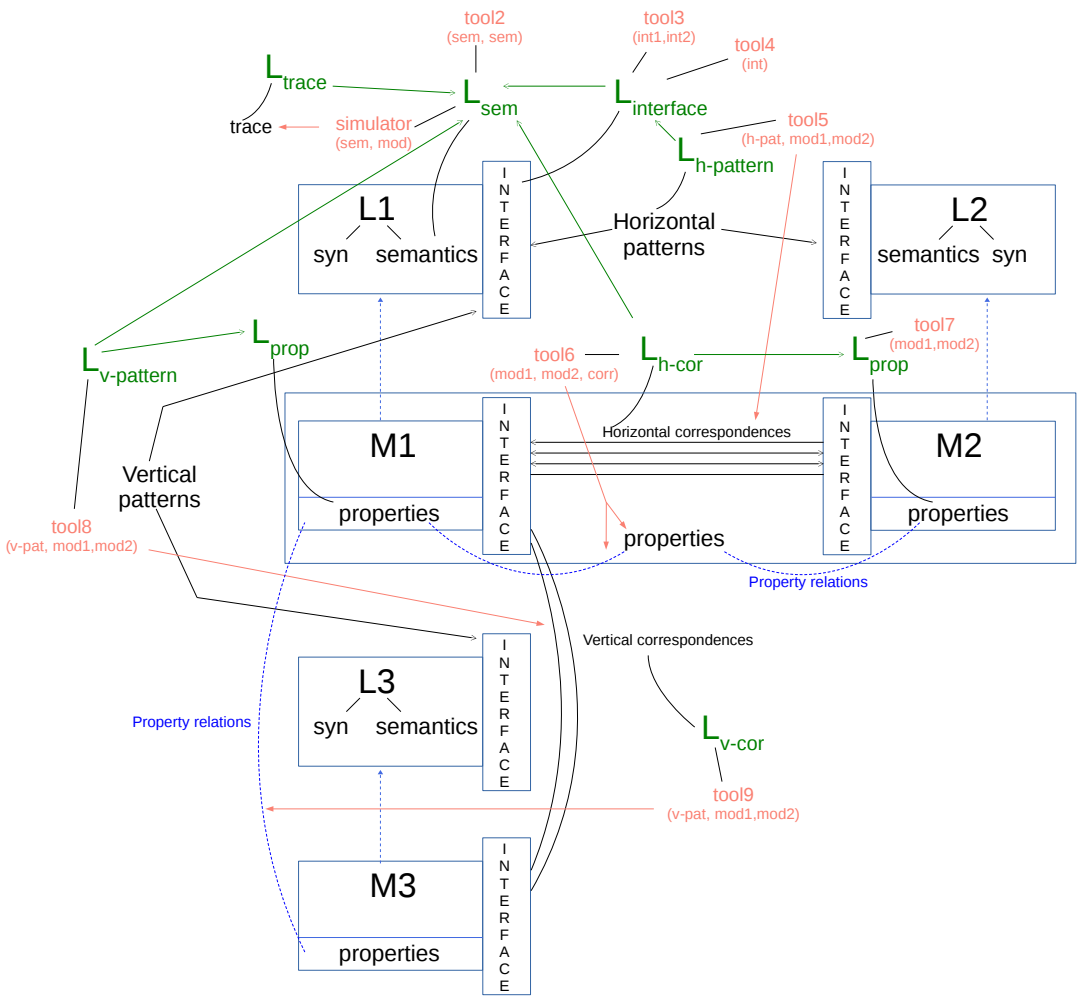

Figure 6.6: Support for Some Thoughts About an Ecosystem for System Engineering

[13] one may try to avoid direct, code level dependencies, and can use for instance megamodels; but conceptually, relations or meta-correspondences are an important support for consistency

[14] for instance there is no reference to the meta language used for the syntax while most of the other meta-languages have relations with it.

Another aspect of such ecosystems is their flexibility in the relations between meta languages and/or the way(s) artifacts are produced. For instance on Figure 6.6, we do not know how properties of models are obtained. They may be obtained by the analysis of a set of traces (*e.g.*, obtained from a simulator), or obtained by an analysis of the semantics of the language according to the structure of the models; or even tagged manually by an expert of the domain. We can imagine that, all these sources bring different kinds of properties that can be useful for different analysis.

In this framework, one interesting work would be to precisely relate models at different abstraction levels. As a start, one can think about the classical abstraction/refinement relation. This relation is usually either left informal or its formalization is hard coded, typically as a *simulation* relation between the traces obtained from the refined model and its abstraction. We could imagine going further and defining different types of abstraction by using *Vertical* patterns. Such patterns could then be applied on models to obtain vertical correspondences from which the relations between the set of properties from two models is clearly established. In other words, we could reason on the possibility to characterize some abstraction/refinement relations so that the properties that are conserved by the abstraction are known and made explicit. This may be a way to tract the relations between systems and subsystems from the requirements until the implementation/realization.

All this said, I have to admit that I understand the potential amount of work such framework represents since understanding the required concepts for a single meta language and realizing its tooling may require several years. We may also be restricted in our analysis/tooling by state of the art or by memory/computational explosion. However, in such case one can think about an appropriate abstraction, not necessarily exact, but that may already be of great help. Additionally, I realize that a lot is left out of the picture like for instance requirement engineering, reuse of legacy code, use of black box tools or socio technical alignment between domains to cite only a few. Anyway I think this is both an interesting playground for researchers and an important subject for industries.

Bibliography

[1] MATLAB. http://www.mathworks.fr/products/matlab/. [1994-2014 The MathWorks, Inc].

[2] Idani Akram. Mise en oeuvre d'une approche formelle en ingénierie des modèles. In *17eme édition d'AFADL, Approches formelles dans l'assistance au développement de logiciels*, Grenoble, 2018.

[3] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, July 1997. ISSN 1049-331X. doi: 10.1145/258077.258078. URL http://doi.acm.org/10.1145/258077.258078.

[4] E. Andrade, P. Maciel, G. Callou, and B. Nogueira. A Methodology for Mapping SysML Activity Diagram to Time Petri Net for Requirement Validation of Embedded Real-Time Systems with Energy Constraints. In *Digital Society, 2009. ICDS '09. Third International Conference on*, pages 266–271, Feb 2009.

[5] C. André, F. Mallet, and M.-A. Peraldi-Frati. A multiform time approach to real-time system modeling: Application to an automotive system. In Universidade Nova de Lisboa, editor, *Int. Symp. on Industrial Embedded Systems*, pages 234–241, Lisboa, Portugal, July 2007. IEEE. ISBN 1-424-40840-7.

[6] Charles André. Syntax and Semantics of the Clock Constraint Specification Language (CCSL). Research Report 6925, INRIA and University of Nice, May 2009.

[7] Charles André and Frédéric Mallet. Clock constraint specification language in UML/MARTE CCSL. Research Report 6540, INRIA, 05 2008. URL https://hal.inria.fr/inria-00280941.

[8] Charles André, Julien DeAntoni, Frédéric Mallet, and Robert de Simone. *The Time Model of Logical Clocks available in the OMG MARTE profile*, chapter 7, pages 201–227. Springer Science+Business Media, LLC 2010, July 2010.

[9] F. Arbab, I. Herman, and P. Spilling. An overview of manifold and its implementation. *Concurrency: Pract. Exper.*, 5(1):23–70, February 1993. ISSN 1040-3108. doi: 10.1002/cpe.4330050103. URL http://dx.doi.org/10.1002/cpe.4330050103.

[10] Farhad Arbab. What do you mean, coordination? In *Bulletin of the Dutch Association for Theoretical Computer Science (NVTI*, pages 11–22, 1998.

[11] Farhad Arbab. A channel-based coordination model for component composition. In *Mathematical Structures in Computer Science*, pages 329–366. University Press, 2002.

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

[13] Daniel Balasubramanian, Corina S. Păsăreanu, Michael W. Whalen, Gábor Karsai, and Michael Lowry. Polyglot: Modeling and analysis for multiple statechart formalisms. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 45–55, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0562-4. doi: 10.1145/2001420.2001427. URL http://doi.acm.org/10.1145/2001420.2001427.

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

[15] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time systems in BIP. In *4th IEEE SEFM*, pages 3–12, September 2006.

[16] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in bip. In *SEFM*, pages 3–12, 2006.

[17] 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*, LNCS, pages 148–160. Springer, 2009.

[18] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceeding of the IEEE*, 79(9):1270–1282, September 1991.

[19] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Sci. Comput. Program.*, 16(2):103–149, 1991.

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

[21] Albert Benveniste, Benoît Caillaud, Hilding Elmqvist, Khalil Ghorbal, Martin Otter, and Marc Pouzet. Structural analysis of multi-mode DAE systems. In Goran Frehse and Sayan Mitra, editors, *Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control, HSCC 2017, Pittsburgh, PA, USA, April 18-20, 2017*, pages 253–263. ACM, 2017. ISBN 978-1-4503-4590-3. doi: 10.1145/3049797.3049806. URL `https://doi.org/10.1145/3049797.3049806`.

[22] Ruslan Bernijazov. Early timing analysis of scenario-based software requirements. Master's thesis, Paderborn University, Paderborn, Germany, June 2017.

[23] G. Berry. The foundations of Esterel. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 425–454, 2000.

[24] Gérard Berry. *The Esterel Language Primer, version v5_91*. Ecole des Mines de Paris, CMA, INRIA, July 2000.

[25] P. Bjureus and A. Jantsch. Modeling of mixed control and dataflow systems in MASCOT. *VLSI Systems, IEEE Transactions on*, 2001.

[26] P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *3rd ACM software engineering symposium on Practical software development environments*, pages 14–24. ACM, 1988.

[27] F. Boulanger and C. Hardebolle. Simulation of multi-formalism models with modhelx. In *ICST*, pages 318–327. IEEE Computer Society, 2008.

[28] Frédéric Boulanger, Ayman Dogui, Cécile Hardebolle, Christophe Jacquet, Dominique Marcadet, and Iuliana Prodan. Semantic Adaptation Using CCSL Clock Constraints. In *Workshops and Symposia at MODELS 2011*, LNCS, pages 104–118. Springer, 2012.

[29] Timothy Bourke and Marc Pouzet. Zelus: A Synchronous Language with ODEs. In *16th International Conference on Hybrid Systems: Computation and Control (HSCC'13)*, pages 113–118, Philadelphia, USA, March 2013. URL `http://www.di.ens.fr/~pouzet/bib/hscc13.pdf`.

[30] Erwan Bousse, Tanja Mayerhofer, Benoit Combemale, and Benoit Baudry. A generative approach to define rich domain-specific trace metamodels. In Gabriele Taentzer and Francis Bordeleau, editors, *Modelling Foundations and Applications*, pages 45–61, Cham, 2015. Springer International Publishing. ISBN 978-3-319-21151-0.

[31] Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale. Execution Framework of the GEMOC Studio (Tool Demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, page 8, Amsterdam, Netherlands, October 2016. URL `https://hal.inria.fr/hal-01355391`.

[32] F. Boussinot and R. De Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293–1304, 2002. ISSN 0018-9219.

[33] Greg Brunet, Marsha Chechik, Steve Easterbrook, Shiva Nejati, Nan Niu, and Mehrdad Sabetzadeh. A manifesto for model merging. In *Proceedings of the 2006 International Workshop on Global Integrated Model Management*, GaMMa '06, pages 5–12, New York, NY, USA, 2006. ACM. ISBN 1-59593-410-3. doi: 10.1145/1138304.1138307. URL `http://doi.acm.org/10.1145/1138304.1138307`.

[34] Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Readings in hardware/-software co-design. chapter Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems, pages 527–543. Kluwer Academic Publishers, Norwell, MA, USA, 2002. ISBN 1-55860-702-1. URL `http://dl.acm.org/citation.cfm?id=567003.567050`.

[35] Doron Bustan, Dana Fisman, and John Havlicek. Automata construction for PSL. Technical report, IBM Haifa Research Lab, 2005.

[36] Christos G. Cassandras and Stephane Lafortune. *Introduction to Discrete Event Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[37] Walter Cazzola. Domain-specific languages in few steps - the neverlang approach. In *Software Composition - 11th International Conference, SC 2012, Prague, Czech Republic, May 31 - June 1, 2012. Proceedings*, pages 162–177, 2012. doi: 10.1007/978-3-642-30564-1_11. URL `http://dx.doi.org/10.1007/978-3-642-30564-1_11`.

[38] Stefano Centomo, Julien Deantoni, and Robert De Simone. Using SystemC Cyber Models in an FMI Co-Simulation Environment. In *19th Euromicro Conference on Digital System Design 31 August - 2 September 2016*, volume 19 of *19th Euromicro Conference on Digital System Design*, Limassol, Cyprus, August 2016. doi: 10.1109/DSD.2016.86. URL `https://hal.inria.fr/hal-01358702`.

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

[40] Kai Chen, Janos Sztipanovits, and Sandeep Neema. Toward a semantic anchoring infrastructure for domain-specific modeling languages. In *Proceedings of the 5th ACM International Conference on Embedded Software*, EMSOFT '05, pages 35–43, New York, NY, USA, 2005. ACM. ISBN 1-59593-091-4. doi: 10.1145/1086228.1086236. URL `http://doi.acm.org/10.1145/1086228.1086236`.

[41] Kai Chen, J. Sztipanovits, and Sandeep Neema. Compositional specification of behavioral semantics. In *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, pages 1–6, 2007. doi: 10.1109/DATE.2007.364408.

[42] Curtis Clifton and Gary T. Leavens. Multijava: Modular open classes and symmetric multiple dispatch for java. In *OOPSLA*, pages 130–145, 2000.

[43] Benoit Combemale, Xavier Crégut, and Marc Pantel. A Design Pattern to Build Executable DSMLs and associated V&V tools. In *APSEC*. IEEE, December 2012.

[44] Benoit Combemale, Julien Deantoni, Matias Vara Larsen, Frédéric Mallet, Olivier Barais, Benoit Baudry, and Robert France. Reifying concurrency for executable metamodeling. In *the 6th International Conference on Software Language Engineering (SLE 2013)*, oct 2013.

[45] Benoit Combemale, Julien Deantoni, Benoit Baudry, Robert France, Jean-Marc Jézéquel, and Jeff Gray. Globalizing Modeling Languages. *Computer*, 2014.

[46] Benoit Combemale, Julien DeAntoni, Benoit Baudry, Robert B. France, Jean-Marc Jezequel, and Jeff Gray. Globalizing modeling languages. *Computer*, 47(6):68–71, June 2014. ISSN 0018-9162. doi: 10.1109/MC.2014.147. URL `http://dx.doi.org/10.1109/MC.2014.147`.

[47] 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*, l'Aquila, Italy, 2015. CEUR. URL `https://hal.inria.fr/hal-01152342`.

[48] S. Cook, G. Jones, S. Kent, and A. Wills. *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley Professional, 2007.

[49] J. R. Cordy, C. D. Halpern, and E. Promislow. TXL: a rapid prototyping system for programming language dialects. In *Conf. Int Computer Languages*, pages 280–285, 1988.

[50] Juan de Lara and Hans Vangheluwe. AToM3: A Tool for Multi-formalism and Meta-modelling. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering*, volume 2306, chapter Lecture Notes in Computer Science, pages 174–188. Springer Berlin / Heidelberg, 2002. ISBN 978-3-540-43353-8.

[51] Julien DeAntoni and Frédéric Mallet. Timesquare: Treat your models with logical time. In Carlo A. Furia and Sebastian Nanz, editors, *Objects, Models, Components, Patterns - 50th International Conference, TOOLS 2012, Prague, Czech Republic, May 29-31, 2012. Proceedings*, volume 7304 of *Lecture Notes in Computer Science*, pages 34–41. Springer, 2012. ISBN 978-3-642-30560-3. doi: 10.1007/978-3-642-30561-0_4. URL `https://doi.org/10.1007/978-3-642-30561-0_4`.

[52] Julien Deantoni and Frédéric Mallet. ECL: the Event Constraint Language, an Extension of OCL with Events. Research report RR-8031, INRIA, July 2012.

[53] Julien Deantoni, Frédéric Mallet, and Charles André. On the Formal Execution of UML and DSL Models. WIP of the 4th International School on Model-Driven Development for Distributed, Realtime, Embedded Systems, April 2009. URL `https://hal.inria.fr/inria-00587100`.

[54] Julien Deantoni, Frédéric Mallet, Frédéric Thomas, Gonzague Reydet, Jean-Philippe Babau, Chokri Mraidha, Ludovic Gauthier, Laurent Rioux, and Nicolas Sordon. RT-simex: retro-analysis of execution traces. In Kevin J. Sullivan Gruia-Catalin Roman, editor, *SIGSOFT FSE*, volume ISBN 978-1-60558-791-2, pages 377–378, Santa Fe, États-Unis, 2010. doi: 10.1145/1882291.1882357.

[55] Julien Deantoni, Frédéric Mallet, Charles André, and Frédéric Thomas. Logical time @ work: the RT-Simex project. In *Sophia Antipolis Formal Approach*, Sophia, France, April 2011. URL `https://hal.inria.fr/inria-00587151`.

[56] Julien Deantoni, Charles André, and Régis Gascon. CCSL denotational semantics. Research Report RR-8628, , November 2014. URL `https://hal.inria.fr/hal-01082274`.

[57] Julien Deantoni, Papa Issa Diallo, Joël Champeau, Benoit Combemale, and Ciprian Teodorov. Operational Semantics of the Model of Concurrency and Communication Language. Research Report RR-8584, INRIA, September 2014. URL `https://hal.inria.fr/hal-01060601`.

[58] 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)*, Grenoble, France, March 2015. URL `https://hal.inria.fr/hal-01087442`.

[59] Thomas Degueule, Benoît Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. Melange: a meta-language for modular and reusable development of dsls. In Richard F. Paige, Davide Di Ruscio, and Markus Völter, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, Pittsburgh, PA, USA, October 25-27, 2015*, pages 25–36. ACM, 2015. URL `https://dl.acm.org/citation.cfm?id=2814252`.

[60] Marco Di Natale, Francesco Chirico, Andrea Sindico, and Alberto Sangiovanni-Vincentelli. An MDA approach for the generation of communication adapters integrating SW and FW components from Simulink. In *ACM/IEEE Models*. 2014.

[61] Docea Power. Aceplorer. `http://www.doceapower.com/products-services/aceplorer.html`. [Feb. 7, 2014].

[62] Emmanuel Durand. *Description et vérification d'architectures d'application temps réel : CLARA et les réseaux de Petri temporels*. PhD thesis, Nantes, ECN, Nantes, 1998. URL `http://opac.inria.fr/record=b1064627`. Th. : automatique et informatique appliquées.

[63] S. Edwards, L. Lavagno, EA Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: formal models, validation, and synthesis. *Proc. of the IEEE*, 85(3):366–390, 1997.

[64] 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):127–144, 2003.

[65] Torbjörn Ekman and Görel Hedin. The JastAdd system – modular extensible compiler construction. *Sci. Comput. Program.*, pages 14–26, 2007.

[66] Jan Ellsberger, Amardeo Sarma, and Dieter Hogrefe. *SDL : formal object-oriented language for communicating systems*. Prentice Hall, Harlow, UK, New York, Paris, 1997. ISBN 0-13-621384-7. URL `http://opac.inria.fr/record=b1104645`.

[67] Matthew Emerson and Janos Sztipanovits. Techniques for metamodel composition. In *in The 6th OOPSLA Workshop on Domain-Specific Modeling, OOPSLA 2006*, pages 123–139. ACM, ACM Press, 2006.

[68] Esper. Espertech, 2009.

[69] Madeleine Faugère, Thimothée Bourbeau, Robert de Simone, and Sébastien Gérard. Marte: Also an UML profile for modeling AADL applications. In *ICECCS - UML&AADL*, pages 359–364. IEEE Computer Society, 2007. ISBN 978-0-7695-2895-3.

[70] Peter H. Feiler and Jörgen Hansson. Flow latency analysis with the architecture analysis and design language. Technical Report CMU/SEI-2007-TN-010, CMU, June 2007.

[71] C. Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, 2002. ISSN 0018-9162.

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

[73] Lars-ake Fredlund, Bengt Jonsson, and Joachim Parrow. An implementation of a translational semantics for an imperative language. In *CONCUR*, LNCS, pages 246–262. Springer, 1990.

[74] Abdoulaye Gamatié, Sébastien Le Beux, Éric Piel, Rabie Ben Atitallah, Anne Etien, Philippe Marquet, and Jean-Luc Dekeyser. A model-driven design framework for massively parallel embedded systems. *ACM Trans. Embedded Comput. Syst.*, 10(4):39, 2011.

[75] Kelly Garcés, Julien Deantoni, and Frédéric Mallet. A Model-Based Approach for Reconciliation of Polychronous Execution Traces. In *SEAA 2011 - 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, Oulu, Finland, August 2011. IEEE. URL `https://hal.inria.fr/inria-00597981`.

[76] David Garlan and Mary Shaw. An introduction to software architecture. Technical report, Pittsburgh, PA, USA, 1994.

[77] Régis Gascon, Frédéric Mallet, and Julien DeAntoni. Logical time and temporal logics: Comparing UML MARTE/CCSL and PSL. In Carlo Combi, Martin Leucker, and Frank Wolter, editors, *Eighteenth International Symposium on Temporal Representation and Reasoning, TIME 2011, Lübeck , Germany, September 12-14, 2011*, pages 141–148, Lübeck , Germany, September 2011. IEEE. ISBN 978-1-4577-1242-5. doi: 10.1109/TIME.2011.10. URL `https://doi.org/10.1109/TIME.2011.10`.

[78] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 1992.

[79] A. Girault, Bilung Lee, and E.A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE TCAD*, 1999.

[80] A. Girault, Bilung Lee, and E.A. Lee. Hierarchical finite state machines with multiple concurrency models. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 18(6): 742–760, 1999. ISSN 0278-0070. doi: 10.1109/43.766725.

[81] Calin Glitia, Julien DeAntoni, and Frédéric Mallet. Logical time at work: Capturing data dependencies and platform constraints. In Adam Morawiec and Jinnie Hinderscheit, editors, *Proceedings of the 2010 Forum on specification & Design Languages, FDL 2010, September 14-16, 2010, Southampton, UK*, page 241. ECSI, Electronic Chips & Systems design Initiative, 2010. URL `http://www.ecsi.org/fdl2010/`.

[82] Calin Glitia, Philippe Dumont, and Pierre Boulet. Array-OL with delays, a domain specific specification language for multidimensional intensive signal processing. *Multidimensional Systems and Signal Processing*, 21(2):105–131, June 2010.

[83] Calin Glitia, Julien DeAntoni, Frédéric Mallet, Jean-Vivien Millo, Pierre Boulet, and Abdoulaye Gamatié. Progressive and explicit refinement of scheduling for multidimensional data-flow applications using UML MARTE. *Design Autom. for Emb. Sys.*, 19(1-2):1–33, 2015. doi: 10.1007/s10617-014-9140-y. URL `https://doi.org/10.1007/s10617-014-9140-y`.

[84] Carlos Gomez, Julien DeAntoni, and Frederic Mallet. Multi-view Power Modeling Based on UML, MARTE and SysML. *Software Engineering and Advanced Applications (SEAA)*, pages 17 – 20, 2012.

[85] Carlos Gomez, Julien DeAntoni, and Frederic Mallet. Power Consumption Analysis Using Multi-View Modeling. *Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 235 – 238, 2013.

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

[87] Cécile Hardebolle and Frédéric Boulanger. Multi-Formalism Modelling and Model Execution. *International Journal of Computers and their Applications*, 31(3):193–203, July 2009.

[88] Thomas A. Henzinger. The theory of hybrid automata. In M.Kemal Inan and RobertP. Kurshan, editors, *Verification of Digital and Hybrid Systems*, volume 170 of *NATO ASI Series*, pages 265–292. Springer Berlin Heidelberg, 2000. ISBN 978-3-642-64052-0. doi: 10.1007/978-3-642-59615-5_13. URL `http://dx.doi.org/10.1007/978-3-642-59615-5_13`.

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

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

[91] Wei Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, and M.R. Stan. Hotspot: a compact thermal modeling methodology for early-stage vlsi design. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14(5):501 –513, may 2006.

[92] IEEE. IEEE recommended practice for architectural description of software-intensive systems. *IEEE Std 1471-2000*, pages 1–23, 2000.

[93] IEEE. Systems and software engineering – architecture description. *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)*, pages 1 –46, 2011.

[94] A. Jantsch. *Modeling embedded systems and SoC's: concurrency and time in models of computation*. Morgan Kaufmann (an imprint of elsevier science), 2004.

[95] A. Jantsch and I. Sander. Models of computation and languages for embedded system design. *Computers and Digital Techniques, IEE Proceedings*, 152(2):114–129, Mar 2005.

[96] Bijoy A Jose, Hiren D Patel, Sandeep K Shukla, and Jean-Pierre Talpin. Generating multi-threaded code from polychronous specifications. *Electronic Notes in Theoretical Computer Science*, 238(1): 57–69, 2009.

[97] Gilles Kahn. The semantics of a simple language for parallel programming. *Information Processing*, pages 471–475, 1974.

[98] Richard M. Karp and Rayamond E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, 1966. doi: 10.1137/0114108. URL `http://link.aip.org/link/?SMM/14/1390/1`.

[99] 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.

[100] Lennart C.L. Kats and Eelco Visser. The spoofax language workbench: rules for declarative specification of languages and IDEs. In *OOPSLA '10*, pages 444–463. ACM, 2010. doi: 10.1145/1869459.1869497. URL `http://doi.acm.org/10.1145/1869459.1869497`.

[101] Amani Khecharem, Carlos Gomez, Julien Deantoni, Frédéric Mallet, and Robert De Simone. Execution of Heterogeneous Models for Thermal Analysis with a Multi-view Approach. In *FDL 2014 : Forum on specification and Design Languages*, Munich, Germany, October 2014. IEEE. URL `https://hal.inria.fr/hal-01060309`.

[102] Jörg Kienzle, Gunter Mussbacher, Benoit Combemale, and Julien Deantoni. A Unifying Framework for Homogeneous Model Composition. *Software & Systems Modeling*, pages 1–19, January 2019. doi: 10.1007/s10270-018-00707-8. URL `https://hal.inria.fr/hal-01949050`.

[103] Jacques Klein and Jean-Marc Jézéquel. Problems of the semantic-based weaving of scenarios. In *In Aspects and Software Product Lines: An Early Aspects Workshop at SPLC-Europe 05*, RENNES, France, September 2005. URL `https://hal.inria.fr/hal-00795065`.

[104] Jacques Klein and Jörg Kienzle. Reusable aspect models. In *IN: Proceedings. Of The 11th International Workshop On Aspect Oriented Modeling.*, 2007.

[105] Jacques Klein, Loïc Hélouët, and Jean-Marc Jézéquel. Semantic-based weaving of scenarios. In Robert E. Filman, editor, *Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, pages 27–38, Bonn, Germany, March 2006. ACM. doi: 10.1145/1119655.1119662. URL `https://hal.inria.fr/hal-00921480`.

[106] Jacques Klein, Franck Fleurey, and Jean-Marc Jézéquel. Weaving Multiple Aspects in Sequence Diagrams. *Transactions on Aspect-Oriented Software Development (TAOSD)*, LNCS 4620:167–199, 2007. URL `https://hal.inria.fr/inria-00505223`.

[107] P. Klint. A meta-environment for generating programming environments. *ACM TOSEM*, 2(2): 176–201, 1993.

[108] Andreas Knüpfer, Ronny Brendel, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Introducing the open trace format (otf). In *International Conference on Computational Science (2)*, pages 526–533, 2006.

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

[110] 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.

[111] Holger Krahn, Bernhard Rumpe, and Steven Volkel. MontiCore: Modular Development of Textual Domain Specific Languages. In *Objects, Components, Models and Patterns*, LNBIP. Springer, 2008.

[112] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: a framework for compositional development of domain specific languages. *International Journal on Software Tools for Technology Transfer*, 12(5):353–372, 2010. ISSN 1433-2779. doi: 10.1007/s10009-010-0142-1. URL `http://dx.doi.org/10.1007/s10009-010-0142-1`.

[113] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. ISSN 0001-0782.

[114] 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)*, Pittsburg, United States, 2015. ACM. URL `https://hal.inria.fr/hal-01185911`.

[115] E. A. Lee and D. G. Messerschmitt. Synchronous Data Flow. *Proc. of the IEEE*, 75(9):1235–1245, September 1987.

[116] E. A. Lee and A. L. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12): 1217–1229, Dec. 1998.

[117] Edward A. Lee. Multidimensional streams rooted in dataflow. In *Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, pages 295–306, 1993.

[118] Edward A. Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Computers*, 36(1):24–35, 1987.

[119] Edward A Lee and Alberto Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE TCAD*, 1998.

[120] Su-Young Lee, Frédéric Mallet, and Robert de Simone. Dealing with AADL end-to-end flow latency with UML Marte. In *ICECCS - UML&AADL*, pages 228–233. IEEE CS, April 2008. ISBN 0-7695-3139-3. doi: 10.1109/ICECCS.2008.14.

[121] Giovanni Liboni, Julien Deantoni, Antonio Portaluri, Davide Quaglia, and Robert De Simone. Beyond Time-Triggered Co-simulation of Cyber-Physical Systems for Performance and Accuracy Improvements. In *10th Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, Manchester, United Kingdom, January 2018. URL `https://hal.inria.fr/hal-01675396`.

[122] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using rapide. *IEEE Trans. Softw. Eng.*, 21(4):336–355, April 1995. ISSN 0098-5589. doi: 10.1109/32.385971. URL `http://dx.doi.org/10.1109/32.385971`.

[123] F. Mallet and C. André. On the semantics of UML/MARTE clock constraints. In *ISORC*, pages 305–312. IEEE, IEEE Computer Society, March 2009.

[124] Frédéric Mallet and Robert de Simone. *MARTE vs. AADL for Discrete-Event and Discrete-Time Domains*, volume 36 of *LNEE*, chapter 2, pages 27–41. Springer, April 2009. ISBN 978-1-4020-9713-3. doi: 10.1007/978-1-4020-9714-0_2.

[125] Frédéric Mallet, Charles André, and Julien DeAntoni. Executing AADL models with UML/MARTE. In *14th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2009, Potsdam, Germany, 2-4 June 2009*, pages 371–376. IEEE Computer Society, 2009. ISBN 978-0-7695-3702-3. doi: 10.1109/ICECCS.2009.10. URL `https://doi.org/10.1109/ICECCS.2009.10`.

[126] 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. doi: 10.1007/s11334-009-0109-0. URL `https://doi.org/10.1007/s11334-009-0109-0`.

[127] Frédéric Mallet, Julien DeAntoni, Charles André, and Robert de Simone. The Clock Constraint Specification Language for building timed causality models. *Innovations in Systems and Software Engineering*, 6:99–106, 2010. ISSN 1614-5046. doi: 10.1007/s11334-009-0109-0. URL `http://dx.doi.org/10.1007/s11334-009-0109-0`.

[128] Raphaël Mannadiar. *A multi-paradigm modelling approach to the foundations of domain-specific modelling*. PhD thesis, McGill University Libraries, 2012.

[129] Radu Mateescu and Damien Thivolle. A model checking language for concurrent value-passing systems. In Jorge Cuéllar, T. S. E. Maibaum, and Kaisa Sere, editors, *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings*, volume 5014 of *Lecture Notes in Computer Science*, pages 148–164. Springer, 2008. ISBN 978-3-540-68235-6. doi: 10.1007/978-3-540-68237-0_12. URL `https://doi.org/10.1007/978-3-540-68237-0_12`.

[130] John McCarthy. Towards a mathematical science of computation. *Information processing*, 62: 21–28, 1962.

[131] Nenad Medvidovic and Richard N. Taylor. A framework for classifying and comparing architecture description languages. In *Proceedings of the 6th European SOFTWARE ENGINEERING Conference Held Jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC '97/FSE-5, pages 60–76, New York, NY, USA, 1997. Springer-Verlag New York, Inc. ISBN 3-540-63531-9. doi: 10.1145/267895.267903. URL `http://dx.doi.org/10.1145/267895.267903`.

[132] Robin Milner. *A calculus of communicating systems*. Springer, 1982.

[133] Pieter J Mosterman and Hans Vangheluwe. Computer automated multi-paradigm modeling: An introduction. *Simulation*, 80(9):433–450, 2004.

[134] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving Executability into Object-Oriented Meta-Languages. In *MoDELS*, LNCS, pages 264–278. Springer, 2005.

[135] Object Management Group. Semantics of a Foundational Subset for Executable UML Models (fUML), V 1.2.1, January 2015. `http://www.omg.org/spec/FUML/1.2.1`.

[136] *UML Object Constraint Language (OCL) 2.0*. Object Management Group, Inc., 2003.

[137] *Meta Object Facility (MOF) 2.0 Core*. Object Management Group, Inc., 2006.

[138] *Semantics of a Foundational Subset for Executable UML Models (fUML), v1.0*. Object Management Group, Inc., 2011.

[139] OMG. *UML Superstructure, v2.2*. Object Management Group, February 2009. formal/2009-02-02.

[140] OMG. *UML Profile for MARTE, v1.0*. Object Management Group, Nov. 2009. Document number: formal/09-11-02.

[141] OMG. UML Profile for MARTE. *Object Management Group*, v1.1, October 2010.

[142] OMG. Omg. systems modeling language (sysml). *Object Management Group*, v1.2, June 2010.

[143] OMG. OMG Unified Modeling Language. *Object Management Group*, v2.4.1, August 2011.

[144] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. Technical report, CWI, 1998.

[145] Marie-Agnès Peraldi-Frati and Julien DeAntoni. Scheduling multi clock real time systems: From requirements to implementation. In *14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC 2011, Newport Beach, California, USA, 28-31 March 2011*, pages 50–57. IEEE Computer Society, 2011. ISBN 978-0-7695-4368-0. doi: 10.1109/ISORC.2011.16. URL `https://doi.org/10.1109/ISORC.2011.16`.

[146] Marie-Agnès Peraldi-Frati, Arda Goknil, Julien DeAntoni, and Johan Nordlander. A timing model for specifying multi clock automotive systems: The timing augmented description language V2. In Isabelle Perseil, Karin K. Breitman, and Marc Pouzet, editors, *17th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2012, Paris, France, July 18-20, 2012*, pages 230–239. IEEE Computer Society, 2012. ISBN 978-1-4673-2156-3. doi: 10.1109/ICECCS.2012.5. URL `http://doi.ieeecomputersociety.org/10.1109/ICECCS.2012.5`.

[147] C. A. Petri. Introduction to general net theory. In *Advanced Course: Net Theory and Applications*, pages 1–19, 1975.

[148] C.A. Petri. Concurrency theory. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and their properties*, volume 254 of *Lecture Notes in Computer Science*, pages 4–24. Springer-Verlag, 1987.

[149] Gordon D Plotkin. A structural approach to operational semantics. 1981.

[150] PSL. IEEE standard for Property Specification Language (PSL), IEEE std 1850-2005.

[151] Raghu Reddy, Robert France, Sudipto Ghosh, Franck Fleurey, and Benoit Baudry. Model composition - a signature-based approach. In *Proceedings of the AOM Workshop at MODELS'05*, Montego Bay, Jamaica, October 2005.

[152] Mara Rhepp, Herbert Stögner, and Andreas Uhl. Comparison of jpeg and jpeg 2000 in low-power confidential image transmission. In *SPC*, 2004.

[153] Ingo Sander and Axel Jantsch. System Modeling and Transformational Design Refinement in ForSyDe. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(1): 17–32, 2004.

[154] Scilab. Scilab Consortium. Scilab. `http://www.scilab.org/`, 2013.

[155] Dana S Scott and Christopher Strachey. *Toward a mathematical semantics for computer languages*. Oxford University Computing Laboratory, Programming Research Group, 1971.

[156] 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. *Software Engineering, IEEE Transactions on*, 21(4):314–335, Apr 1995. ISSN 0098-5589. doi: 10.1109/32.385970.

[157] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Cheddar: a flexible real time scheduling framework. In *SIGAda*, pages 1–8, New York, NY, USA, 2004. ACM. ISBN 1-58113-906-3.

[158] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley, 2008.

[159] Janos Sztipanovits and Gabor Karsai. Model-Integrated Computing. *IEEE Computer*, 30(4), 1997.

[160] *OMAP35x Applications Processor Technical Reference Manual*. Texas Instruments, Apr 2010.

[161] The ProMARTE Consortium. *UML Profile for MARTE, beta 3*. Object Management Group, May 2009. OMG document number: ptc/2009-05-13.

[162] J.P. Tolvanen and M. 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.

[163] Matias Vara Larsen. *BCOol : the Behavioral Coordination Operator Language*. Theses, Université Nice Sophia Antipolis, April 2016. URL `https://tel.archives-ouvertes.fr/tel-01302875`.

[164] Matias Vara Larsen and Arda Goknil. Railroad Crossing Heterogeneous Model. In *GEMOC workshop*, 2013.

[165] Matias Ezequiel Vara Larsen, Julien Deantoni, Benoit Combemale, and Frédéric Mallet. A Behavioral Coordination Operator Language (BCOoL). In *International Conference on Model Driven Engineering Languages and Systems (MODELS)*, number 18, page 462. ACM, September 2015. URL `https://hal.inria.fr/hal-01182773`.

[166] Matias Ezequiel Vara Larsen, Julien Deantoni, Benoit Combemale, and Frédéric Mallet. A Model-Driven Based Environment for Automatic Model Coordination. In CEUR, editor, *Models 2015 demo and posters*, Models 2015 demo and posters, Ottawa, Canada, October 2015. URL `https://hal.inria.fr/hal-01198744`.

[167] A. Vassighi and M. Sachdev. *Thermal and power management of integrated circuits*, chapter Thermal and Electrothermal Modeling. Springer Science+Business Media, Incorporated, 2006.

[168] Markus Voelter and Konstantin Solomatov. Language Modularization and Composition with Projectional Language Workbenches illustrated with MPS. In *SLE*, LNCS. Springer, 2010.

[169] M. Völter. From Programming to Modeling-and Back Again. *Software, IEEE*, 28(6), 2011.

[170] Glynn Winskel. Event structures. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency*, LNCS. Springer, 1987.

[171] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT press, 1993.

[172] Eric Van Wyk, Oege de Moor, Kevin Backhouse, and Paul Kwiatkowski. Forwarding in attribute grammars for modular language design. In *CC'02*, pages 128–142. Springer, 2002.

[173] Huafeng Yu, Jean-Pierre Talpin, Loïc Besnard, Thierry Gautier, Frédéric Mallet, Charles André, and Robert de Simone. Polychronous analysis of timing constraints in UML MARTE. In *IEEE Int. W. on Model-Based Engineering for Real-Time Embedded Systems Design*, pages 145–151, Parador of Carmona, Spain, 2010.