BCool: the Behavioral Coordination Operator Language

Matias Ezequiel Vara Larsen

UNIVERSITÉ NICE SOPHIA ANTIPOLIS

ÉCOLE DOCTORALE STIC

SCIENCES ET TECHNOLOGIES DE L'INFORMATION

ET DE LA COMMUNICATION

T H È S E

pour l'obtention du grade de

Docteur en Sciences

de l'Université Nice Sophia Antipolis

Mention : Informatique

presentée et soutenue par

Matias Ezequiel Vara Larsen

B-COoL: Un Metalangage pour la Spécification des Opérateurs de Coordination des Langages

B-COoL: the Behavioral Coordination Operator Language

Thèse dirigée par: Frédéric Mallet
et encadrée par: Julien DeAntoni

soutenue le 11 avril 2016, au laboratoire I3S, Sophia Antipolis

Jury :

M.	Olivier Barais	Prof.	Université de Rennes 1	*Rapporteur*
M.	Sébastien Gérard	DR	CEA LIST	*Rapporteur*
M.	Jean-Michel Bruel	Prof.	Université de Toulouse 2	*Examinateur*
M.	Frédéric Mallet	Prof.	Univ. Nice Sophia Antipolis	*Directeur de Thèse*
M.	Julien DeAntoni	MCF	Univ. Nice Sophia Antipolis	*Encadrant*

Résumé

Les appareils modernes sont constitués de plusieurs sous-systèmes de différentes sortes qui communiquent et interagissent. L'hétérogénéité de ces sous-systèmes et leurs interactions complexes rendent très délicat leur développement. L'approche d'ingénierie dirigée par les modèles apporte une solution en permettant l'expression de nombreux modèles structurels et comportementaux de natures très diverses. Dans ce contexte, il est nécessaire de construire un modèle unique qui intègre ces différents modèles afin d'y appliquer des méthodes de validation et de vérification pour permettre aux ingénieurs système de comprendre et de valider un comportement global. Cependant, la coordination manuelle des différents modèles qui composent le système est une opération source d'erreurs et les approches automatiques proposent des patrons de coordination ad-hoc pour certaines paires de langages. Dans ces approches, le patron de coordination est souvent encapsulé dans un outil dont il est difficile d'extraire les liens avec le système global. Cette thèse propose le Behavioral Coordination Operator Language (B-COoL), un langage dédié à la spécification de patrons de coordination entre des langages à partir de la définition d'opérateurs de coordination. Ces opérateurs sont employés afin d'automatiser la coordination de modèles exprimés dans ces langages. B-COoL est implémenté comme une suite de plugins qui s'appuient sur l'Eclipse Modeling Framework et présente ainsi un environnement complet pour l'exécution et la vérification de différents modèles coordonnés. Nous illustrons cette approche avec la définition d'opérateurs de coordination entre deux langages: *timed finite state machines* et *activities*. Ensuite, nous utilisons ces opérateurs afin de coordonner et d'exécuter un modèle hétérogène de caméra de surveillance.

Abstract

Modern devices embed several subsystems with different characteristics that communicate and interact in many ways. This makes its development complex since a designer has to deal with the heterogeneity of each subsystem but also with the interactions among them. To tackle the development of complex systems, Model Driven Engineering promotes the use of various, possibly heterogeneous, structural and behavioral models. In this context, the coordination of behavioral models to produce a single integrated model is necessary to provide support for validation and verification. It allows system designers to understand and validate the global and emerging behavior of the system. However, the manual coordination of models is tedious and error-prone, and current approaches to automate the coordination are bound to a fixed set of coordination patterns. Moreover, automatic approaches encode the pattern into a tool thus limiting the reasoning on the global system behavior. In this thesis, we propose the Behavioral Coordination Operator Language (B-COoL) to reify coordination patterns between specific domains by using coordination operators between the Domain-Specific Modeling Languages used in these domains. Those operators are then used to automate the coordination of models conforming to these languages. B-COoL is implemented as plugins for the Eclipse Modeling Framework thus providing a complete environment to execute and verify coordinated models. We illustrate the use of B-COoL with the definition of coordination operators between two languages: timed finite state machines and activities. We then use these operators to coordinate and execute the heterogeneous model of a surveillance camera system.

Acknowledgements

This thesis involved the collaboration of many people. My thanks go to:

- My advisors Frederic Mallet and Julien DeAntoni whose advices improved my quality as research scientist;
- Doctor Sébastien Gérard and Professor Olivier Barais, for accepting and evaluating my thesis;
- The AOSTE team whose advice improved my research and for every lunch;
- The GEMOC project for all the feedback that helps me a lot for improving my results;
- My family Julio, Silvia, Pilar, Julian, and my grandparents Dina, Pichi, Maria and Baltazar. Without your support and encouragement, my success would not have been possible.

Dedication

To Julio, for driving a fiat 147 while putting us through college.

To Silvia, for marking us breakfast every day for twenty-three years. Each.

To Julian, for joking during dinners.

To Pilar, for answering the calls during nap time.

Para Julio, por llevarnos al colegio en el fiat 147.

Para Silvia, por preparanos el desayuno cada mañana por 23 años.

Para Julian, por hacernos reir durantes las cenas.

Para Pilar, por las llamadas durante la siesta.

'For a true writer each book should be a new beginning where he tries again for something that is beyond attainment. He should always try for something that has never been done or that others have tried and failed. Then sometimes, with great luck, he will succeed.'

Ernest Hemingway

Contents

Résumé **3**

Abstract **5**

Acknowledgements **7**

1 Introduction **1**

2 Background **5**

2.1 Introduction 5

2.2 Composition Approaches 6

2.2.1 Model Composition Approaches 6

2.2.2 Language Composition Approaches 9

2.2.3 Discussion 11

2.3 Coordination Approaches 12

2.3.1 Model Coordination Approaches 12

2.3.2 Language Coordination Approaches 16

2.3.3 Discussion 20

2.4 Conclusion 21

3 Requirements for a Language to specify Coordination Patterns 23

3.1 Introduction . . . 23

3.2 Language Behavioral Interface . . . 25

3.2.1 Review of Existing Approaches . . . 25

3.2.2 Requirements . . . 29

3.3 Correspondence Rules . . . 31

3.3.1 Review of Existing Approaches . . . 31

3.3.2 Requirements . . . 32

3.4 Coordination Rules . . . 35

3.4.1 Review of Existing Approaches . . . 35

3.4.2 Requirements . . . 37

3.5 Conclusion . . . 38

4 B-COoL: the Behavioral Coordination Operator Language 41

4.1 Introduction . . . 41

4.2 Running Example: Coordination of the Heterogeneous Models of a Coffee Machine . . . 43

4.3 The Language . . . 47

4.3.1 Abstract Syntax of B-COoL . . . 48

4.3.2 Library . . . 52

4.3.3 Execution Semantics . . . 54

4.4 Implementation . . . 57

4.5 Evaluation . . . 63

4.6 Conclusion . . . 65

5 Validation **67**

5.1 Introduction 67

5.2 Use Case: Coordination of the Heterogeneous Models of a Surveillance Camera System 68

5.3 Definition of Coordination Operators between the TFSM and Activity Languages . . . 70

5.4 Use of the Operators in a Surveillance Camera System 76

5.5 Conclusion 79

6 Conclusion **81**

6.1 Overview 81

6.2 Future Works 84

Bibliography **86**

List of Figures

1.1 Stakeholders in the development of a Complex System 3

2.1 Model Composition Approaches Sketching 7

2.2 Language Composition Approaches Sketching 9

2.3 Model Coordination Approaches Sketching 13

2.4 Specification of a client-server system in Wright [AG97] 15

2.5 High level view of the approach proposed by Di Natale et al. in [DNCSSV14] 18

2.6 High level view of Ptolemy [GLL99] 19

4.1 Overview of the proposed approach 42

4.2 (At the top) An excerpt of the TFSM metamodel with a part of its language behavioral interface. (At the bottom) a TFSM model with a part of its model behavioral interface 44

4.3 (At the top) An excerpt of the Activity metamodel with a part of its language behavioral interface. (At the bottom) an Activity model with a part of its model behavioral interface 46

4.4 Coordination of the models of the coffee machine by constraining the corresponding MSE 47

4.5 Simplified View of B-COoL abstract syntax 48

4.6 Resulting coordination of the coffee machine by using the event relation Rendezvous . 50

4.7 Resulting coordination of the coffee machine by using the global event variable globalClock 51

4.8 Excerpt of MoCCML metamodel 52

4.9 State-based representation of the relation *RendezvousWithGlobalClock* 53

4.10 Resulting coordination of the Coffee Machine by using the automata relation *RendezvousWithGlobalClock* 54

4.11 Steps in the application of the B-COoL specification between the models of the coffee machine 55

4.12 The proposed workflow for the heterogeneous development of complex applications . . 58

4.13 Overview of the implementation of B-COoL and its integration into the Gemoc Studio 59

4.14 Coordinated Execution of models by using the Gemoc Coordinated Execution Engine 62

4.15 State space representation of the coordinated model of the coffee machine, encoding the set of valid schedules. The transitions in red contain the events forced to happen simultaneously by the coordination. 63

5.1 Representation of the Camera Encoder Control by using a TFSM 69

5.2 Representation of the JPEG encoding algorithm by using an activity diagram 70

5.3 Representation of the JPEG2000 encoding algorithm by using an activity diagram . . 70

5.4 Representation of the Battery Sensor by using an activity diagram 71

5.5 Coordinated model of a surveillance camera system and a partial representation of the model behavioral interface 72

5.6 *ExecuteActivityNonPeemptive* event relation 74

5.7 *AtomicActivity* event relation 75

5.8 Coordinated execution of the models of the surveillance camera system by using the Gemoc studio 78

5.9 Partial resulting timing output of the surveillance camera system 78

List of Listings

3.1 Specification of the Mascot correspondence rule by using the Epsilon Comparison Language . . . 34

4.1 Partial ECL specification of TFSM . . . 43

4.2 Partial ECL specification of Activity diagram . . . 45

4.3 B-COoL specification of the running example operator between the TFSM and Activity languages . . . 48

4.4 B-COoL specification of an operator that illustrates the use of Event Variables . . . 51

4.5 B-COoL specification of an operator that illustrates the use of a MOCCML library . . . 53

4.6 Resulting CCSL specification for the coffee machine system . . . 57

4.7 B-FLoW specification for the models of the coffee machine . . . 60

5.1 B-COoL specification of the *SyncFSMEventsAndSignals* operator . . . 71

5.2 B-COoL specification of the *StartActivityWhenEnter* operator . . . 73

5.3 B-COoL specification of the *AtomicActivity* operator . . . 74

5.4 B-FLoW specification for the Surveillance Camera System . . . 77

Chapter 1

Introduction

Nowadays devices are becoming more complex. They embed several subsystems with different characteristics that communicate and interact in many ways. For example, cars can integrate an adaptive control cruise system, GPS tracking, fuel control system and so on. Furthermore, these subsystems are widely coupled. For instance, the adaptive cruise system determines the way to get home depending on the GPS tacking, also, the fuel control system regulates the speed of the car depending on the level of fuel. This makes the design of these systems very complex. A designer has to deal with the heterogeneity of each subsystem but also with the interactions among them. To deal with this inherent complexity, the design is split into different domains, *e.g.*, mechanical, electronic, software. The development is thus tackled by different *Domain Experts*.

Model Driven Engineering (MDE) promotes the use of Domain Specific Modeling Languages (DSMLs) to model complex systems using the adequate, domain-specific, terminology and tools (see Figure 1.1). DSMLs are developed by *Language Engineers*[1] as dedicated languages to make a description of a domain relying on the expert terminology so at to reduce introducing discrepancies in the model. Models must include structural aspects but also behavioral ones. Thus defining a DSML consists in defining its syntax but also its behavioral semantics. When several DSMLs, with different syntax and or semantics, are used conjointly to model a complex system, we say the model is *heterogeneous*, *i.e.,* it is made of models that conform to different languages. Dealing with this kind of heterogeneity is the problem addressed in this thesis.

With heterogeneous models, the overall behavior emerges from its parts. To perform verification and

[1] Also called Domain Experts.

validation activities of the whole system, designers need tools to comprehend this emerging behavior, or at least some aspects of it depending on the domain of interest. It is therefore necessary to specify how models and languages are related to each other, in both a structural and a behavioral way.

In this context, the GEMOC initiative proposes to coordinate and disseminate the research results regarding the support of the coordinated use of various modeling languages, that is, the use of multiple modeling languages to support heterogeneous development of diverse aspects of a system. This thesis is part of GEMOC project and it focuses on the *coordination* [GC92a] of behavioral models to provide simulation and/or verification capabilities for the whole system specification.

Currently, Coordination Languages [GC92a] and Architecture Description Languages (ADLs) [MT97] provide dedicated languages to specify the coordination between particular behavioral models. This is usually done by *System Designers* (see Figure 1.1) that apply some coordination patterns according to their own skills and know-how. However, in large heterogeneous systems, the manual coordination of models can become tedious and error prone. To automate this task, Coordination Frameworks [BHLM02, BH08] have encoded a predefined coordination pattern inside a tool. However, the customization of these tools for a specific domain is difficult. Moreover, these approaches rely on a general purpose language (GPL) to express the coordination, thus limiting the task of a system designer to reason about how a system is coordinated.

In this thesis, we deal with the coordination of heterogeneous behavioral models by leveraging on the system designer's skills. We propose a dedicated language named B-COoL (standing for *B*ehavioral *C*oordination *O*perat*o*r *L*anguage) that allows for capturing coordination patterns between a given set of DSMLs. These patterns are specified at language level, and then used to derive a coordination model 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.

By using B-COoL, a *Language Integrator* (see Figure 1.1) defines operators that specify how events from different language behavioral interfaces interact. These operators are defined at the language level but they are applied between models to coordinate their behavior. This results in a model of coordination specified in the Clock Constraint Specification Language (CCSL), a declarative language that describes causal and temporal relationships between events. By relying on CCSL, we provide verification and validation facilities for the coordinated system. All this has been implemented as a

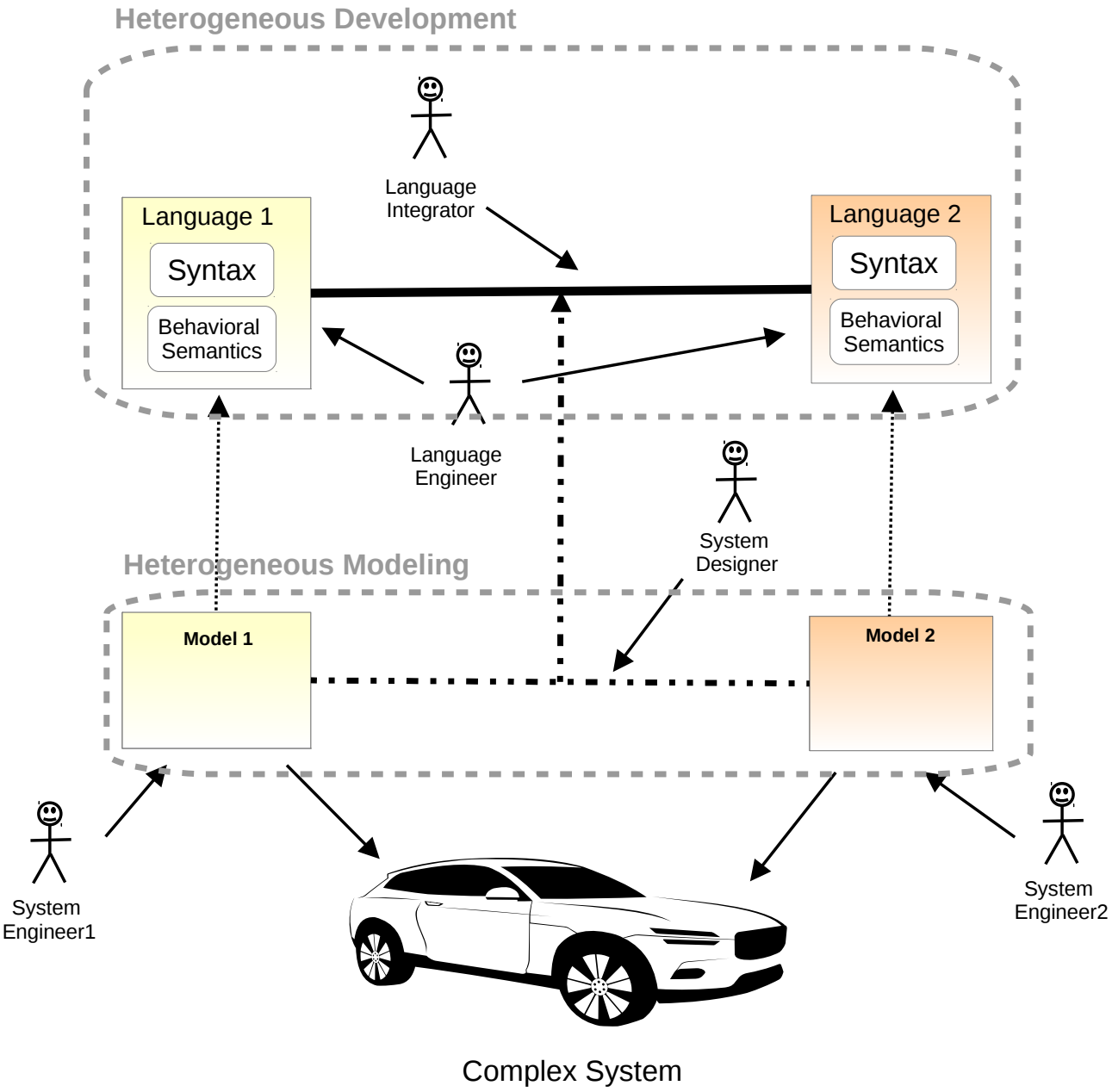


Figure 1.1: Stakeholders in the development of a Complex System

set of plugins for Eclipse as part of the GEMOC studio[2]; which integrates technologies based on Eclipse Modeling Framework (EMF)[3] adequate for the specification of executable domain specific modeling languages.

We organize the content of this thesis in six chapters. Chapter 2 presents the state-of-art approaches that give support for the developing of heterogeneous systems. First, we present approaches that propose to compose models/languages into a new model/language. Second, we present approaches that propose to coordinate the behavior of models. In particular, we focus on tools and frameworks that specify coordination patterns between languages to automate the coordination between models. We conclude this chapter by highlighting benefits and drawbacks of existing approaches.

Chapter 3 presents the requirements for a language to specify coordination patterns in the context og the heterogeneous development of complex systems. We first propose a framework to characterize the approaches that specify a coordination pattern. Then, we use this framework to compare existing approaches. From this comparison, we state the requirements to make existing approaches more flexible and well founded.

Based on these requirements, Chapter 4 presents a particular implementation of the framework named B-COoL; a dedicated language to specify coordination patterns. To illustrate B-COoL, we rely on a running example: the coordination of the heterogeneous models of a coffee machine, which is built by using timed finite state machines (TFSM) and activity diagrams. Then, we use this example through all the chapter to illustrate the syntax and semantics of B-COoL. We present the current implementation in the GEMOC studio by executing and verifying the coffee machine models.

To validate our approach, Chapter 5 presents the heterogeneous models of a surveillance video system decomposed into three subsystems. To coordinate these subsystems, we propose a set of B-COoL operators between the TFSM and activity languages. We use this example as a use case to illustrate the different steps from the specification of the coordination patterns, the coordination of a particular model and the verification of the coordinated system.

Finally, we provide the conclusion of this work, highlighting its main contributions and we give some perspectives in Chapter 6.

[2] `http://www.gemoc.org/studio`
[3] `http://eclipse.org/modeling/emf/`

Chapter 2

Background

2.1 Introduction

To deal with complexity issues in the development of applications for complex systems, Model Driven Engineering (MDE) proposes to rely on *Models*. A model is an abstraction of the real world made in order to facilitate an understanding of the way it works. In the context of MDE, a software model enables a developer to reduce the complexity of an application by ignoring non-essential details. This enables the developer to reason about the application.

The Object Management Group (OMG) proposes to specify models by relying on a language that has a well-defined form (syntax), meaning (semantics) and possible rules of analysis, inferences or proof for its constructs [mda03]. Thus, to build models, MDE proposes Domain Specific Modeling Languages (DSMLs). They are built by *Language Engineers* to describe the structure but also the behavior of a particular domain. As a result, a DSML is defined by a syntax and a semantics. The syntax is described by a metamodel that defines the concepts and relations that the language is made up. A metamodel is a model that is developed by using a metameta language, *e.g.*, MOF, ECORE. To define the semantics, the language theory proposed three types of semantic definitions: Operational [Plo81], Axiomatic [Hoa69] and Translational [FJP90]. The concurrent theory has also proposed other ways to describe the behavior of a model. This behavior is characterized by the so-called Models of Computation (MOCC) [GBA+09]. In this thesis, we focus on this approach for the description of the behavioral semantics of a language.

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 *Model of Coordination*.

2.2 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*.

2.2.1 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 input model languages, or to a different language (see Figure 2.1).

Some approaches [BCE+06, KPP06, FBFG08] have automated the composition between models 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.

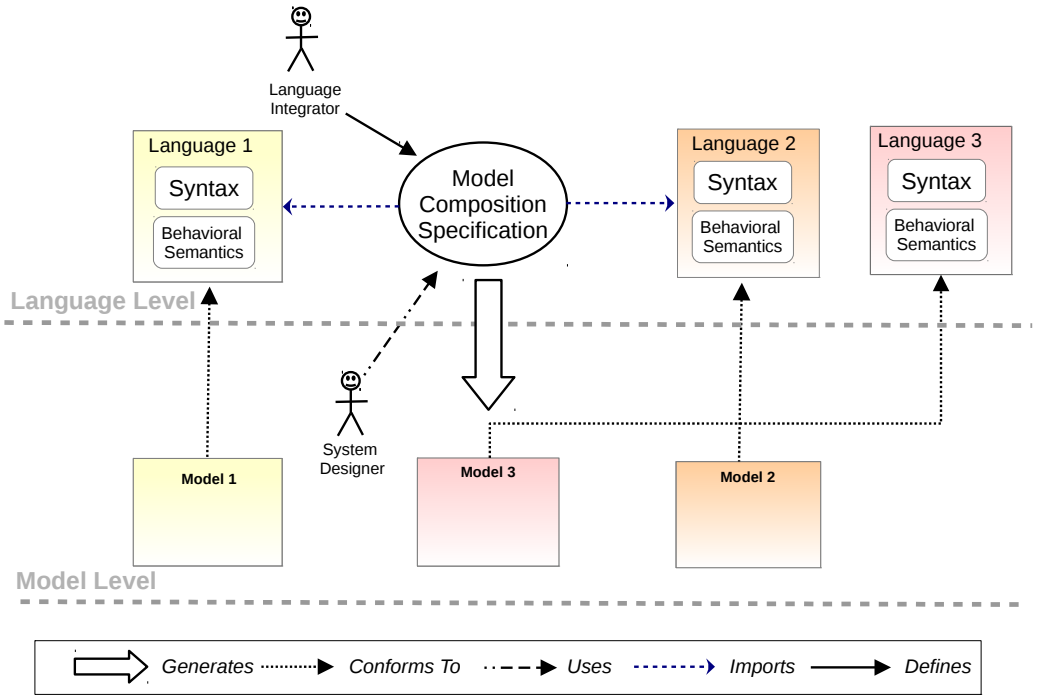


Figure 2.1: Model Composition Approaches Sketching

In [BCE+06], 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 [FBFG08], 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 [RFG+05].

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

While in Kompose these operators are generic, Epsilon [KPP06][1] 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 metamodels. Matching rules apply between models and select elements that must be composed. Then, the EML is used to define *merging*

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

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, for instance, with 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) [KFJ07, KK07, KHJ06] 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 [KK07], 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 [KJ05]. 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 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.

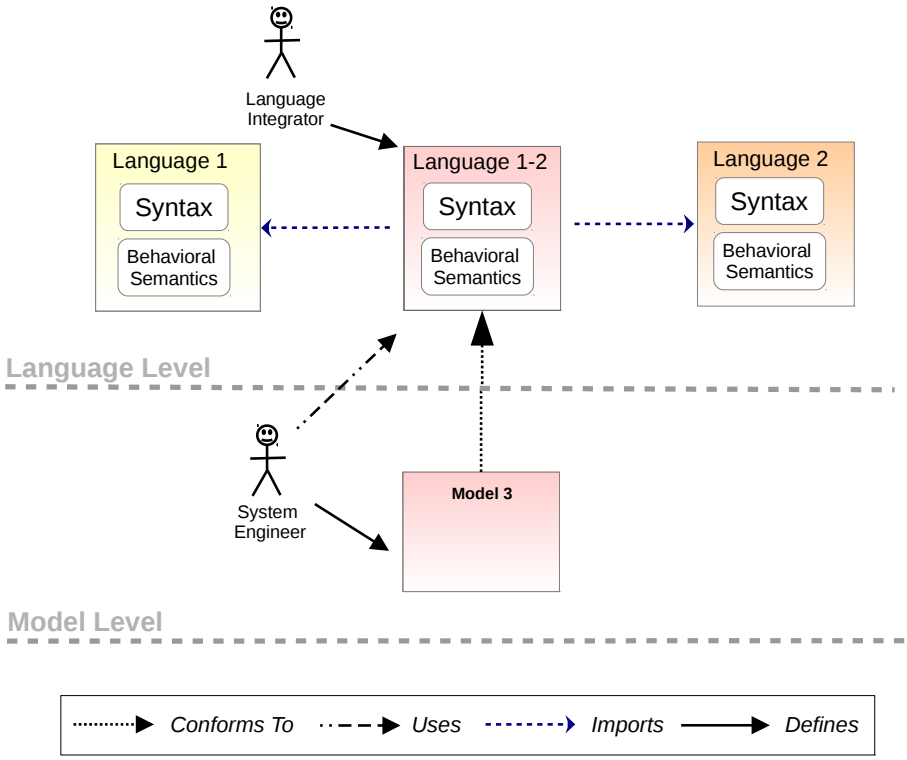


Figure 2.2: Language Composition Approaches Sketching

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 have identified that, in some cases, is necessary 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.

2.2.2 Language Composition Approaches

Language composition approaches provide techniques to compose languages into a new language (see Figure 2.2). We begin by presenting approaches that compose the syntax of languages into a new syntax [ES06]. Then, we present an approach [CSN05] that proposes to define the behavioral semantics of a language as the composition of different language behavioral semantics.

In the literature, Emerson et al. [ES06] 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 "black-box" in a second DSML, *i.e.,* the concept defined in one language refines in other in the second language.

These techniques have been implemented in the GME framework [ES06], which is based on the metametamodel MetaGME[2]. The refinement and interfacing have also been implemented in Monticore [KRV10]. In this approach, these techniques are named respectively: *inheritance* and *language embedding*. A different approach is Neverlang [Caz12] that relies on interfacing to build a custom language from features coming from different General Purpose Languages. A feature, such as the syntactical aspect of a loop, is encapsulated in a *module block*. The blocks can be composed together for generating the compiler/interpreter of the resulting language.

Semantic Anchoring [CSN05] 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[3]) 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 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 [CSN07], a SU

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

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.

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.

2.2.3 Discussion

In this section, we presented approaches that have addressed the problem of 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 [BCE+06, KPP06, FBFG08]. In particular, Epsilon [KPP06] 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 they must conform to the same metametamodel. Most of these approaches consider only the syntax of languages thus ignoring their semantics. Only a few approaches consider the semantics of languages for the composition [KFJ07, KK07, KHJ06]. However, they only compose homogeneous models, *e.g.*, sequence diagrams [KK07]. Thus, its 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.

Languages composition approaches propose to model heterogeneous systems by relying on a unified language. Such a language results from the composition of different languages. The presented techniques [ES06] focus on the composition of syntaxes into a new language syntax. Only semantic anchoring [CSN05] 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 its solution is to define semantics for heterogeneous DSMLs as the composition of semantic units. However, the developing of heterogeneous systems may involve different domain experts that use different

languages. In this context, language composition approaches do not seem 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.

2.3 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 thesis, we adopt the wording of *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 [GC92a] and Architecture Description Languages (ADLs) [MT97] to specify the coordination between behavioral models. The latter are Coordination Frameworks [BHLM02, BH08] and ad-hocs solutions [BJ01, DNCSSV14] 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.

2.3.1 Model Coordination Approaches

Model coordination approaches provide dedicated languages to specify the coordination between (heterogeneous) behavioral models (see Figure 2.3). We begin this subsection by presenting Coordination Languages, and then we continue with ADLs.

Coordination Languages [GC92a] 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. [Arb98] proposed another

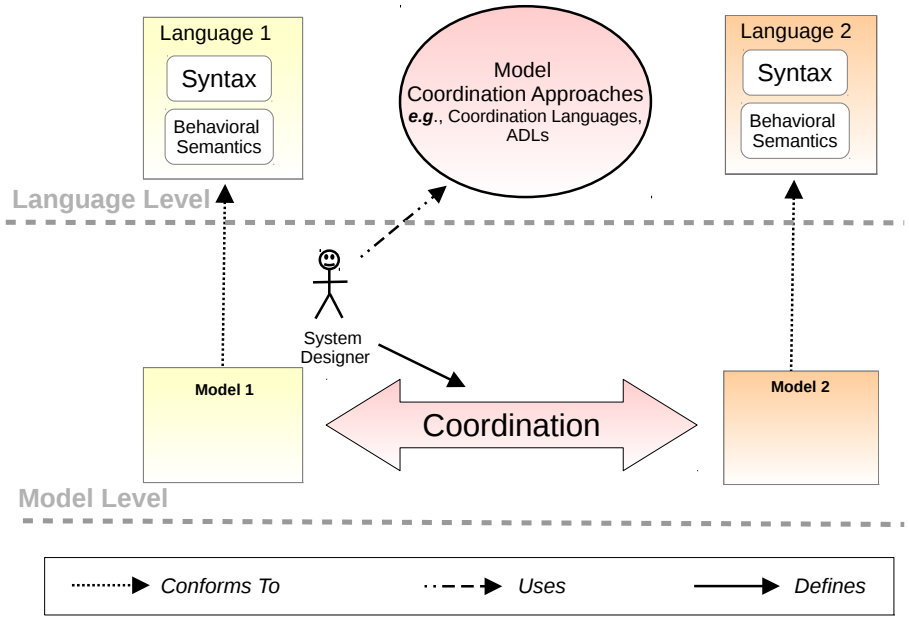


Figure 2.3: Model Coordination Approaches Sketching

classification into *endogenous* and *exogenous* languages. Endogenous languages provide coordination primitives that must be incorporated within a model for its coordination. For instance, Linda [GC92b] 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* [CHM+97], 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 [Win87] (also named signals in [LSV98]). 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 [Esp09, AHS93, BFJ+04]. While in Esper [Esp09], 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 [BFJ+04] 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.

Concurrently with Coordination Languages, the *software architecture* community has developed so-called ADLs to gain abstraction, structuring and reasoning capabilities in the development of complex systems [LKA+95, AG97, SDK+95, MT97, GS94]. 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 [MT97] that can be used on the shelf to specify domain specific interactions.

For example, Clara [Dur98] 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.

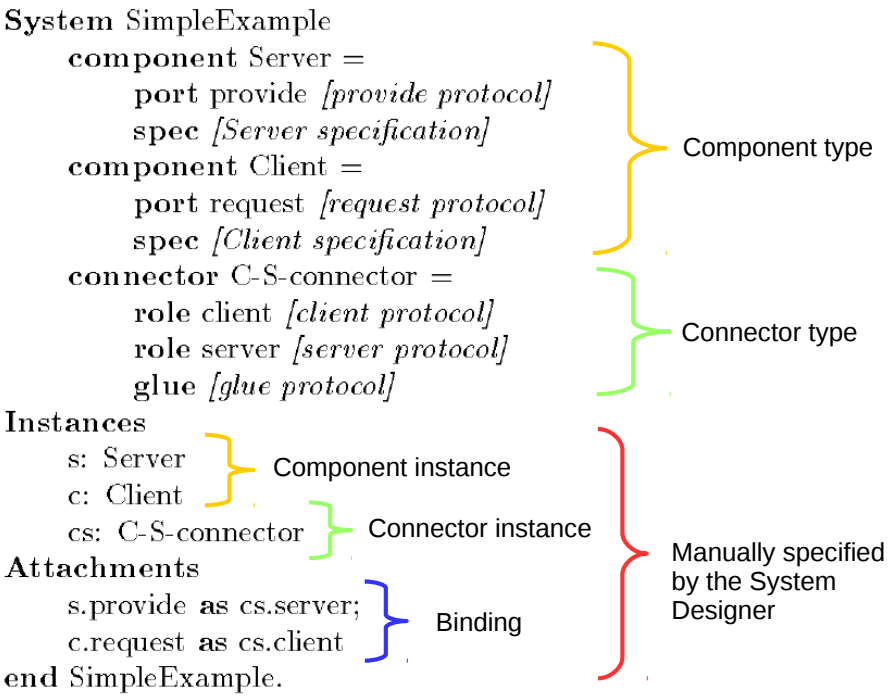


Figure 2.4: Specification of a client-server system in Wright [AG97]

Other approaches introduced the notion of *User Defined Type* [SDK+95, AG97, AA02] 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 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 [AG97], the glue is specified in a variant of CSP [Hoa85], differently in Reo, it is specified by the composition of dedicated primitives [AA02]. 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 2.4 shows a simple client-server system described in Wright [AG97]. 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 [AG97], authors claimed that connector types enables 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 2.4, 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.

2.3.2 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 [BJ01, DNCSSV14] which use a predefined set of languages. We continue with more systematic approaches named Coordination Frameworks [BH08, BHLM02].

Mascot [BJ01] is an approach focused on the integration of Matlab [Gui92] and SDL [ESH97]. 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. In addition, the approach gives a set of guidelines to help the choice between the two proposed synchronization. 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. The approach has identified and specified a coordination pattern between Matlab and SDL by providing two mechanisms to coordinate signals from both domains. In addition, it provides some guidelines to help the choice between the two mechanism. The current implementation, however, partially automates the coordination since the user has to specify what synchronization mechanism is used.

In [DNCSSV14] (see Figure 2.5), authors dealt with the integration of a language to describe the functional aspects of a system with 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

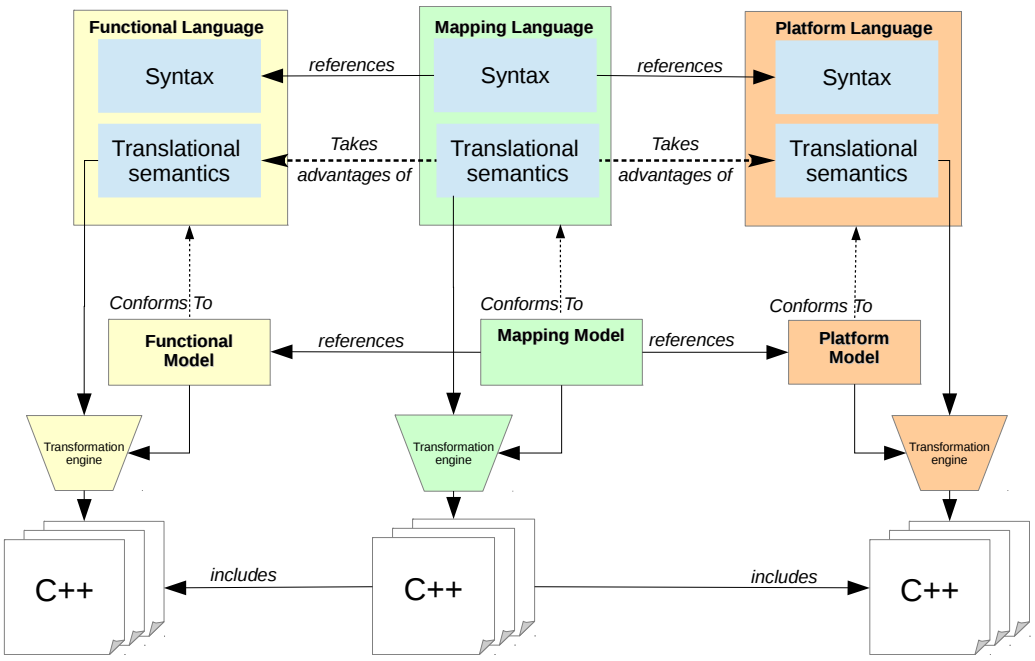


Figure 2.5: High level view of the approach proposed by Di Natale et al. in [DNCSSV14]

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.

While the previous approaches are ad-hoc solutions for two particular languages, Ptolemy [BHLM02] and ModHel'X [BH08] are systematic approaches to coordinate 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 2.6) or composite (*e.g.*, *Actor 0* and *Actor 2* in Figure 2.6), *i.e.,* made of internal, connected,

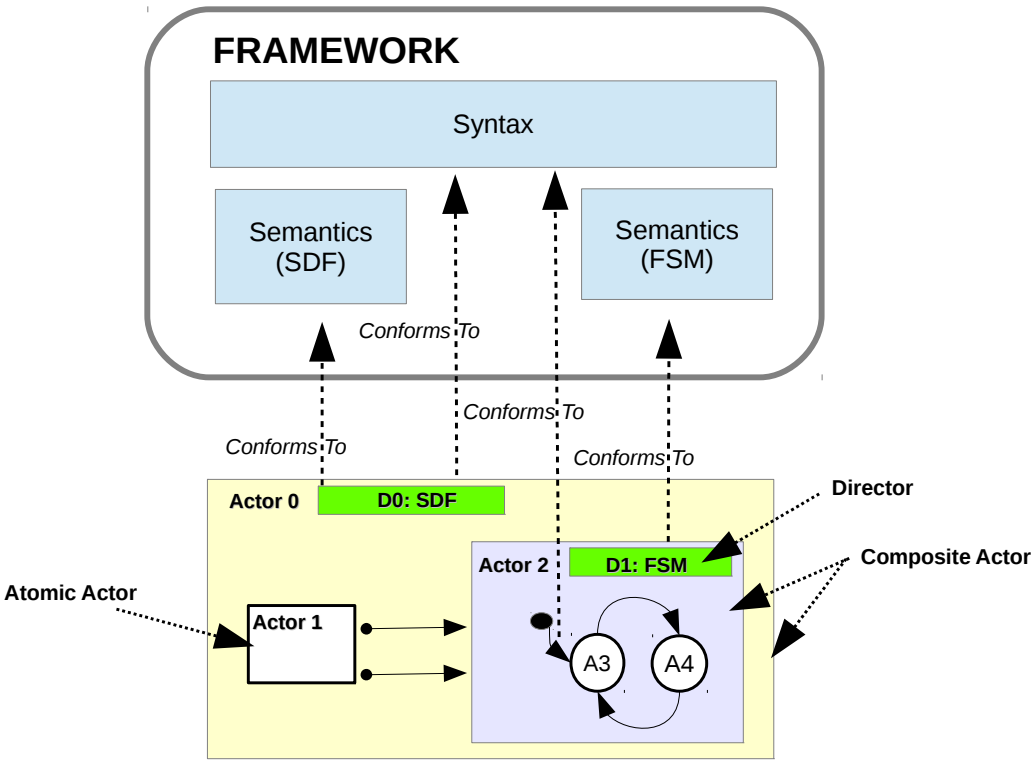


Figure 2.6: High level view of Ptolemy [GLL99]

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 2.6, *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 2.6, 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.

In conclusion, based on a fixed syntax and a generic 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 semantical aspects, they encode the necessary

glue between interfaces of composite actors and internal actors to coordinate their execution. This results in a hierarchical 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.

2.3.3 Discussion

In this section, we have 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 explicit and amenable for reasoning (for instance for Verification and Validation activities). Furthermore, they propose languages close to system designer domain. For example, ADLs provide types in order to define domain-specific connectors. ADLs have successfully identified connector types, however, a system designer has still to instantiate the required type of connectors 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 synthesis 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 other.

2.4 Conclusion

In this chapter, we have presented the state-of-art approaches that rely on two mechanisms to get a global representation of a heterogeneous system: *composition* and *coordination*.

We have presented approaches that compose heterogeneous models/languages to obtain a new model/language. The composition of models has been automated by looking for correspondences between heterogeneous models, and then composing them into a new model that conform to another language. Most of the approaches consider structural correspondences and only few consider also the behavior of models to find similarities. Furthermore, we have determined that these last approaches only compose homogeneous models. This limits the use of such approaches in complex systems where heterogeneous behavioral models may be used.

We have presented approaches that compose languages to get a new language. Most of these approaches focus on the composition of the syntax of languages into a new syntax. Only Semantics Anchoring [CSN05] proposes to compose behavioral semantics by relying on the notion of Semantic Units. These approaches propose to model a heterogeneous system by relying on a single language which results from the composition of other languages. From our point of view, language composition approaches are not suitable for separation of preoccupations and development of a single system by various domain experts who focus on a specific part of the system.

We have presented state-of-the-art approaches that deal with coordination of the behavior of heterogeneous models/languages. First, we have presented the work done by Coordination Languages and ADLs that support the coordination of heterogeneous models. However, the system designer has to manually specify each relation, thus making this task tedious and error prone. Then, we have shown how language coordination approaches have leveraged on the know-how of a system designer to automate the coordination between models. However, we have noted that the knowledge about system integration is encoded within a framework. Furthermore, in frameworks, the model of coordination is expressed by using a general purpose language thus limiting verification and validation activities.

In this thesis, we rely on the coordination of (heterogeneous) behavioral models to get the emerging system behavior of a heterogeneous system. We propose to automate the coordination between models by specifying coordination patterns between languages. In the next chapter, we elaborate on the requirements towards a language to specify coordination patterns. Such requirements tend to improve existing language coordination approaches to make them more flexible and well founded. From these requirements, we propose a language for specifying coordination patterns named B-COoL. Then, in Chapter 4, we present the implementation of our language.

Chapter 3

Requirements for a Language to specify Coordination Patterns

3.1 Introduction

Language Coordination approaches are used to capture a coordination pattern between languages. These approaches go one step beyond Coordination Languages and ADLs by leveraging on the know-how of a system designer. However, current implementations seem different and not always well characterized. The lack of a systematic way to specify a coordination pattern makes these approaches ad-hoc and not flexible. Furthermore, this prevents a wider adoption of this sort of approaches. In this context, this chapter proposes a framework to compare and understand the approaches that offer solutions to capture coordination patterns. The objective of the framework is to highlight similarities and differences between these approaches, and to express the requirements to make them more flexible and generic. For instance, it is important that they address other languages, or other ways of coordinate them. The goal is not to make a further detailed analysis of state-of-the-art approaches but rather to highlight the good points that we intend to keep in our proposition.

The proposed framework is mainly influenced by three existing comparison/categorization frameworks. Authors proposed frameworks that focus on ADLs [MT97], Coordination Languages [PA98] and Model Composition approaches [JFB08]. For example, in [MT97], authors identify the primary blocks used to build an ADL. Based on these *building-blocks*, they have proposed a framework to classify and compare several existing ADLs. A similar work is presented in [PA98] where authors surveyed and classified

several coordination languages. They focused on 1) the nature of the entities being coordinated, 2) the coordination mechanism, 3) the coordination architecture and 4) the semantics of the coordination. In [JFB08], Jeanneret et al. proposed a framework to compare model composition approaches by relying on the triplet *what-where-how* questions, which identifies respectively which elements should be composed, where elements should be inserted or modified and how the composition process works to get the expected result.

Inspired by these works, we propose a framework to characterize approaches that specify a coordination pattern. Existing approaches have in common that:

1. They specify the coordination between languages;
2. They specify correspondences between elements of the conformed models;
3. They specify how the elements selected by the correspondences must be coordinated.

Regarding the first point, existing approaches specified the coordination at the language level by relying on information (at least partially) about the behavioral semantics of the coordinated languages. Therefore, the framework identifies such a representation of the language behavioral semantics as a *Language Behavioral Interface*. Concerning the second and third points, the framework identifies respectively *Correspondence Rules* and *Coordination Rules*. The former identifies how approaches specify *correspondences* between elements from different models, i.e., the rules that defines which elements of the two models should be coordinated. The latter identifies how approaches specify the *coordination* between elements selected by the correspondences, *i.e.,* how the selected elements should be coordinated.

For this first contribution, we review some of the approaches discussed in the previous chapter and we identify precisely what we consider to be related to the language behavioral interface, the correspondence rules and the coordination rules. We also highlight the mechanisms used and what can, in our view, be improved to build a generic and flexible language to specify coordination patterns. We then propose some requirements to improve existing approaches by making each element of the framework explicit and better customizable.

We organize this chapter as follows. We begin by presenting the notion of language behavioral interface, then we continue by presenting correspondences between elements from the interfaces encoded into

the notion of correspondence rules, and finally, we present the notion of coordination rules. Based on the requirements presented in this chapter, Chapter 4 presents a particular implementation of this framework named B-COoL.

3.2 Language Behavioral Interface

This section presents the notion of *Language Behavioral Interface*. We begin by reviewing the concept of language interface in existing approaches, and then, we present requirements to make the behavioral interface of a language explicit and better customizable.

3.2.1 Review of Existing Approaches

The notion of language interface has been recently the focus of some works in the context of the globalization of modeling languages [CvdBCR15, BJL+15, DBC+15, DCB+15]. From these works, it is consensual that the content of a language interface is purpose-specific, *i.e.,* the content varies depending on the use of the interface. In particular in this subsection, we discuss the notion of a language interface for the specific purpose of specifying coordination patterns between languages. Since the coordination activity requires a view of the behavioral semantics of languages, we name such an interface *Language Behavioral Interface*.

From Chapter 2, at model level the notion of behavioral interface is used to expose information about the model behavior to allow its coordination. Similarly, at the language level, a language behavioral interface is used to expose only a part of the behavioral semantics of languages to allow the expression of coordination patterns. In other words, a language behavioral interface abstracts the behavioral semantics of a language, providing only the information required to coordinate it.

In the following, we show how existing approaches specify coordination patterns by relying on partial information about the syntax and behavioral semantics of the languages they use. In addition, we review the work done in [CDVL+13] in which authors propose to specify an executable language that exposes a part of its behavioral semantics. This work will help us to understand the benefits of an explicit language behavioral interface.

Ptolemy [BHLM02]/ModHel'X [BH08]: In these approaches, the behavioral semantics of a language is described by a director that encodes a domain in Java, *e.g.*, FSM, DE, SDF. To support

the communication between different domains, each director implements a generic interface in Java. Such an interface is made of specific set of methods (*e.g.*, *initialize()*, *prefire()*, *fire()*, *postfire()*, *wrapup()*). This mechanism provides a homogeneous view of all the domain behavioral semantics. The implementation of these methods depends on the domain. However, they must conform to a common goal, which is expressed in natural language. For instance, a part of the *fire()* function description is: "*Typically, the fire() method performs the computation associated with an actor*".

Both frameworks are based on a unique abstract syntax (*i.e.,* actor model) to represent the syntax of any language. Then, they propose composite actors to identify what actor can be refined into another actor. Hence, the coordination is done according to the hierarchy of actors. In addition, only some elements of the language syntax can be represented by a composite actor. For example, in the FSM domain only states can be a composited actor, but not the transitions. This makes that only some elements of the language syntax can contain other actors. Thus, frameworks are also aware of (a part of) the abstract syntax of the coordinated language. However, such knowledge is left implicit into the framework implementation.

The *pro* of these approaches is that the framework can technically coordinate actors from any domain without knowledge about the implementation of the actor domain. In this sense, the language behavioral interface correctly provides an abstract and homogeneous view of the language behavioral semantics. The methods represent the coordination points, *i.e.,* elements that can be used to specify the coordination between different domains.

The *cons* of these approaches are twofold. First, the interface is specified in a particular technological domain, *i.e.,* Java. This limits the representation of the language behavioral semantics as a set of methods and the coordination as a set of calls to such methods. It is consequently not possible to add extra coordination points without modifying deeply the framework itself. Second, all the domains have to implement consistently these methods, however, the meaning of the methods is given by their names together with some comments in the code. This remains very informal to ensure a correct implementation of these methods.

These frameworks base on a fixed syntax in which all languages are represented by using the actor model. This limits the expressiveness of the coordinated language syntax. However, it makes simple the interface that only contains information about the language behavioral semantics. In addition, the approach left implicit which elements from the syntax of a language can be refined, *i.e.,* can be a

composite actor.

Di Natale et al. [DNCSSV14]: In this work, authors use a translation to a common executable language (*i.e.,* C++) to provide the behavioral semantics of the coordinated languages. The coordination is supported by a mapping language, which its behavioral semantics is also translated in C++. As a result, the behavioral semantics of both the coordinated languages and the mapping language are represented in the same technology.

In this approach, information about the syntax and semantics of the functional and platform languages is contained into the mapping language. First, it contains correspondences between syntactic elements from both languages, *e.g.*, *Task*s from the functional and *Cpu*s from the platform. Second, to express the semantics of such correspondences, the translation semantics of the mapping language is based on knowledge about the translation semantics of the coordinated languages. For instance, it contains the name of the methods that are generated for each subsystem in the functional language. Such methods are part of the translational semantics of the functional language.

The mapping language is aware of both the syntax and behavioral semantics of the coordinated languages. This information is encoded into the syntax and the behavioral semantics of the mapping language. More precisely, the language behavioral interface of each coordinated language is left implicit into the mapping language. The major *cons* of this solution is that any change in the semantics of the coordinated languages compromises the semantics of the mapping language. The approach lacks of a clear specification of the coordination points, *i.e.,* the important methods (their names but also their parameters) that are resulting from the translation of some specific part of a language.

The only *pro* of this approach is that shows that the notion of language behavioral interface also makes sense when the behavioral semantics of the languages is given by a translation to another language.

Mascot [BJ01]: This is an ad-hoc solution to coordinate SDL and MATLAB. To coordinate these languages, authors have knowledge about the syntax and the behavioral semantics of the languages. For instance, authors know that MATLAB relies on a data-flow MoCC which is based on the notion of streams. Besides, they know that SDL relies on a Finite State Machine which is based on events. Based on this knowledge, authors provide different mechanism to synchronize events from SDL and streams from MATLAB. However, the information about what is coordinated is left implicit into the approach. They rely on partial information about the syntax and behavioral semantics of the

languages, but such information is not made explicit by using a language behavioral interface. The *pros* of this work is that authors have achieved to coordinate two languages that are developed by using different technologies. However, the knowledge about what elements are coordinated is implicit into the approach, thus limiting this approach to these two languages. Thus, the major *cons* is that this work cannot be easily extended to other languages.

Combemale et al. [CDVL$^+$13]: In a recent work [CDVL$^+$13] the authors propose to specify an executable language as a 4-tuple $< AS,\ DSA,\ DSE,\ MoCC >$ where the *AS* is the Abstract Syntax of the language, the *DSA* defines both the data that represents the execution state of the model and the execution functions that modify this execution state. The MOCC represents the, possibly timed, causalities and synchronization of the system by using some events and constraints between them. Then, the *Domain Specific Events* (DSE) link together the three other parts. A DSE is an event type, defined in the context of a metaclass of the *AS* that links an event from the MOCC with the call to an execution function from the *DSA*. DSE are defined by using a specific language named ECL (standing for Event Constraint Language [DM12a]) which is an extension of OCL [OMG03] with events. ECL takes benefits from the OCL query language and its possibility to augment an abstract syntax with additional attributes (without any side effects). Consequently, by using ECL, it is possible to augment AS metaclasses and add DSE.

This approach reifies elements of the behavioral semantics of the language to propose an explicit language behavioral interface. The interface is based on sets of DSE (*event types*) and *contraints*. Jointly with the DSE, the 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. Furthermore, from a DSE, it is possible to get information about the *AS* since they are defined in the context of a metaclass. Then, the context of a DSE can be used to get information from the AS. For each model conforming to a 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, authors use them here to unfold the execution of models. The DSE are then a specification of the coordination points that the model will propose.

Note that the partial representation of the language behavioral semantics is exposed by using DSE (Domain Specific Event types). In that sense, the proposed language behavioral interface is domain specific instead of generic like in Ptolemy [BHLM02] or ModHel'X [BH08].

The *pro* of this approach is to make explicit a language behavioral interface, which provides an intentional specification of the coordination points, together with information about the concurrent and time-related aspects of the language behavioral semantics. In addition, the language behavioral interface exposes a part of the abstract syntax.

In this work, however, authors do not propose any technique or method to take advantages of their interface. For instance, it would be interesting to understand what kind of analysis can be driven based on their interface.

3.2.2 Requirements

We want to highlight that all approaches rely on partial information about the languages they use. However, they have not achieved to systematically express such information in a language behavioral interface. To improve the way that approaches represent information about the coordinated languages, we propose the following requirements:

1. A language behavioral interface should specify in intention the coordination points;
2. A language behavioral interface should expose (a part of) the concurrency and the time-related aspects of the language behavioral semantics;
3. A language behavioral interface should expose (a part of) the abstract syntax.

Let us consider each of these requirements in turn. In Ptolemy and ModHel'X, the coordination is based on a language behavioral interface that is generic. The interface specifies, in intention, the coordination points (*i.e.,* Java methods) that any model provides. Similarly, Combemale et al. specify coordination points in intention by using DSE. This notion of coordination points specified in intention would have been also beneficial for the approach provided by Di Natale et al. to be more flexible with regard to the addition of a new language.

The nature of the coordination points varies from one approach to other. In Ptolemy and ModHel'X, they are implemented by methods, whereas in Combemale et al., they are implemented by using

event types. At the model level, [GS94] explains that there are important benefits of using events (with implicit invocation) in the component interfaces because it provides strong support for reuse and evolution of components. This gives support for control and timed coordination while remaining independent of the internal model 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 [Win87] 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.*, [And09] or [BCCSV05]).

These mechanisms provide an abstraction of a language behavioral semantics by exposing concurrent and time related aspects. This information allows reasoning about how to coordinate different languages. This could have been beneficial for Ptolemy or ModHel'X where the adaptation of the time related aspect is based on a deep knowledge of the internal implementation of the domains. Instead, by adding only the necessary information in the interface, such an adaptation could be easier without all the knowledge about implementation.

Together with information about the language behavioral semantics, all the reviewed approaches also make use of information about the syntax of the coordinated languages. For instance in Di Natale et al., the mapping language contains information about the syntax of the functional (*e.g.*, *Tasks*) and the platform languages (*e.g.*, *CPU*). Ptolemy and Modhel'X also contains information about which elements from the language syntax can be defined as composite actors.

In this section, we presented three requirements for a language behavioral interface to specify a coordination pattern between languages. These requirements aim at providing the needed information about the behavioral semantics of languages for coordination purpose. Although [CDVL$^+$13] did not address the coordination between languages, it is the only approach to provide this interface consciously. In the next section, we study how existing approaches get correspondences between model elements by relying on correspondence rules.

3.3 Correspondence Rules

This section presents the notion of *Correspondence Rules*. We first review the concept of correspondence rules in existing approaches, and then, we present requirements to make them explicit and better customizable.

3.3.1 Review of Existing Approaches

A correspondence is any explicit or implicit relationship between model elements. Such a relationship is specified by a system designer that knows what elements from different models are related. In the specification of coordination patterns, correspondences specify the model elements that are coordinated. To automate the process of looking for correspondences between models, the specification of a coordination pattern involves the definition of correspondence rules at language level. Such rule define, in intention, the correspondences to be instantiated at model level. In the following, we review the existing approaches by focusing on how they implemented correspondence rules and correspondences.

Ptolemy [BHLM02]/ModHel'X [BH08]: In these approaches, the notion of composite actors is used to specified the correspondences between models: when an actor is composite (*i.e.,* it contains other actors) then it coordinates its internal actors. This enables the designing of a system by following a hierarchical scheme where the level n in the hierarchy specifies the coordination of the models that are at the level $n-1$. These approaches propose a fixed correspondence rule, *i.e.,* the composite actor relation, which is encoded into the syntax of the framework.

The *pro* of these approaches is the simplicity for a designer to express the correspondences. He has only to specify what models are composited and which ones are contained inside. The correspondence rule is unique and provided into the common abstract syntax.

The main *cons* of these approaches is they rely on a unique abstract syntax to describe both the syntax of models and the syntax of the correspondences. This prevents the independent development of the models (possibly developed in different languages) and the correspondences. In addition, the use of a hierarchical design may limit the number of the supported languages. For instance, the UML sequence diagram could not be easy to introduce in the hierarchical view of the framework.

Di Natale et al. [DNCSSV14]: In this work, a mapping model is used to define the correspondences between the functional and the platform model. The mapping model is defined by using a dedicated language (*i.e.,* mapping language) that fixes what type of concepts between the platform and the functional languages can be bound together. For instance, the *SwDeployment* correspondence maps two types of concepts: one of type *CPU* and another one of type *Task*. In this sense, the approach provides a set of fixed correspondence rules.

The *pro* of the approach is that the mapping model defines explicitly the correspondences between the functional and the platform model. Such a model could be useful both reasoning and for traceability.

The *cons* of this approach is that a system designer has to manually build the mapping model depending on the current deployment. The approach has successfully identified some relationships between a functional and a platform language, which are captured by a set of predefined correspondence rules. From such a correspondence rules, the approach, however, is not able to automate the instantiation of correspondences at model level.

Mascot [BJ01]: In this work, the correspondences are implicit in the models. The approach relies on a naming convention to specify when events in SDL and streams in Matlab must be coordinated. It relies on a correspondence rule that specifies that the elements to be coordinated must have the same name. When this rule is applied between models, the framework can automatically get correspondences between model elements.

The *pro* of this approach is the use of a correspondence rule to select model elements. This makes the approach very flexible in terms of dependencies between the languages, thus easing the support for a new language.

The *cons* is the encoding of the correspondence rule in the framework. Thus, the approach is limited to find correspondences by comparing the name of the elements (*i.e.,* events from SDL and streams from Matlab). In some cases, it could be interesting to compare more than the names, *e.g.*, the types. In addition, the approach forces the use of a naming convention in the models.

3.3.2 Requirements

In the reviewed approaches, we identified different correspondence rules to get correspondences between models elements. Ptolemy and ModHel'X rely on a common abstract syntax to identify the elements

to be coordinated; Di Natale et al. propose a dedicated language to express an explicit mapping between the elements of different models; and Mascot defines some rules that allow the inference of the correspondences for specific models. However, the main drawback of these approaches is that the customization of the correspondence rules is not possible. Such a characteristic motivates the following requirements for correspondence rules and correspondences:

1. Correspondence rules should provide support for heterogeneous languages;
2. Correspondence rules should avoid creating dependencies between a predefined set of languages to enable the support of new languages;
3. Correspondence rules must be explicitly defined and customizable depending on the domain and conventions followed by different system engineers;
4. Correspondences between model elements should be explicitly represented.

We discuss each requirement in turn. The current development of complex systems is tackled by relying on several heterogeneous languages where each language has its own syntax and behavioral semantics. Thus, an approach that capture the specification of a coordination pattern must be able to identify correspondences between models that are heterogeneous in terms of the syntax. This prevents the possibility to rely on a common syntax for the languages and the correspondences like in Ptolemy and ModHel'X.

The support of heterogeneous languages should be done in a way that a new language can be easy to add. For instance, in the case of Di Natale et al., the mapping language (*i.e.,* the language used for the correspondences) depends on the coordinated languages, (*i.e.,* the functional and the platform languages). So that, if any of these languages are modified or a new language has to be added, the mapping language must be changed. Instead, correspondence rules could be defined by relying on partial information exposed in the interface. This would avoid strong dependencies between the correspondence rules and the coordinated languages.

A correspondence rule can be used to find correspondences between models that conform to two particular languages. This is the case in [KMDS13] where authors use design space exploration techniques to determine the (best) correspondences between an application and its deployment platform. Also, correspondence rules can be used to capture conventions followed by system engineers in an organiza-

tion. The complexity of the conventions can vary from a simple naming convention (*e.g.*, Mascot) to a more complex ontology based system. Based on such conventions, correspondence rules can determine what elements from different models must be coordinated.

Listing 3.1: Specification of the Mascot correspondence rule by using the Epsilon Comparison Language

```
1 rule MatchEventWithStream
2 match s : SDLMetamodel!Event
3 with t : MatlabMetamodel!Stream
4 {
5 compare {
6    return s.name = t.name;
7   }
8 }
```

To make correspondence rules explicit, a dedicated language should be used. To illustrate this, we propose to use the Epsilon Comparison Language (ECL [Kol09]) to sketch the definition of the correspondence rule in the case of Mascot. The ECL is a language dedicated to the expression of correspondence rules. In this language, correspondence rules are expressed by querying and then comparing model elements. For example, Listing 3.1 shows the specification in ECL of a matching rule named *MatchEventWithStream*. Roughly speaking, the matching rule is used to find correspondences between instances of the *Event* and *Stream* classes (Listing 3.1: line 2 and 3). These classes can be defined in different metamodels (*i.e.,* Ecore models). The class Event is defined in an metamodel referred as *SDLMetamodel* and the class Stream is defined in an metamodel referred as *MatlabMetamodel* (Listing 3.1: line 2 and 3). The comparison is done by using the attribute name defined in the context of each class (Listing 3.1: line 5). To express the comparison, the approach relies on a query language named Epsilon Object Language[1]. When two instances of these classes have the same name, the pairs are matched.

The main benefit of the use of such a dedicated language is to ease the customization of the correspondence rules, thus enabling to adapt them according to the company modeling conventions and the language's nature. For instance in Listing 3.1, the matching is equivalent to the one encoded in Mascot, however, it can be customized to add another criteria for the matching. Furthermore, conventions are independent of the languages themselves, so it is easy to add support for a new language without changing the framework.

[1] `http://www.eclipse.org/epsilon/doc/eol/`

Finally, to understand how a particular system is coordinated, correspondences between model elements must be clearly represented. This has already identified in the Ptolemy and ModHel'X by providing a dedicated syntax to specify the correspondences. Unlike these approaches, in Mascot, the correspondences are implicit into the approach thus making necessary to read the documentation to find out that the approach relies on a correspondence rule that follows a naming convention. In such a case, an explicit representation of the correspondences can help a system designer to understand the coordinated elements. Also, it allows external tools to take advantages of the correspondences. The explicit representation of correspondences should be done without creating dependencies between the languages used in the system in order to fulfill the second requirement.

In this section, we have presented some requirements to improve the way that approaches implements correspondence rules and correspondences. In the next section, we present the notion of coordination rule that specifies how the models elements selected by a correspondence must be coordinated.

3.4 Coordination Rules

In this section, we first review the notion of *Coordination Rules* in existing approaches, and then, we present some requirements to make them better defined.

3.4.1 Review of Existing Approaches

In the specification of a coordination pattern, coordination rules specify *how* the elements selected by the correspondence rules must be coordinated. Based on a coordination rule, approaches specify the interaction between elements in a correspondence. In the following, we review how existing approaches implement a coordination rule depending on the specified coordination pattern.

Ptolemy [BHLM02]/ModHel'X [BH08]: In these approaches, composite actors interact with their internal actors by invoking the methods in their interface. The coordination rule implements a hierarchical execution of actors in which the coordination is expressed in Java together with the semantics of the actors. In ModHel'X, authors made explicit the notion of semantics adaption between actors by allowing the specification of the code between two component interfaces. Such a manual adaptation is also done in Java.

These approaches, in particular Ptolemy, were the first to propose a hierarchical framework to coor-

dinate heterogeneous models. These approaches, however, do not leverage on the formal works done by their authors in the field of Model of Computation [LSV98]. The current implementation contains only few of the ideas proposed in such works. Thus, the main *cons* is that the approaches are limited in terms of verification and validation of the coordinated system.

ModHel'X went one step forward by proposing the notion of semantics adaptation between two components. However, the semantics adaptation is never reifed at language level thus making the approach close to the notion of glue in existing ADLs.

Di Natale et al. [DNCSSV14] & Mascot [BJ01]: In these approaches, the coordination rule is specified by the translational semantics of the correspondences. In Di Natale et al. [DNCSSV14], each correspondence in the mapping model is translated to a specific glue in C++. The glue encodes how elements of the functional and the platform translation semantics are coordinated. In the case of Mascot [BJ01], the coordination rule is encoded by using SDL wrappers which results in a coordination expressed in C.

The *pros* of these approaches is that they achieved to coordinate models that are developed in different technologies, *i.e.,* Maltab and SDL. They do so by expressing the semantics of both the coordination and the models in a common GPL, *i.e.,* C, C++.

The main *cons* of these approaches is that they are very limited in terms of customization and explicitness of the coordination rule. They encode the coordination rule into tools thus limiting reasoning about how a particular system is coordinated. In addition, the coordination rule depends on the implementation of the coordinated languages. For instance, in Di Natale et al., the translational semantics of the mapping language depends on the translational semantics of the functional and the platform languages. Thus, the task of adding support to a new language needs a well understanding about the implementation of the coordinated languages.

Another *cons* of these approaches is that they force to express the semantics of both the model and the coordination in the same language. The use of a GPL to express the coordination limits the verification and validation of the resulting coordinated system.

3.4.2 Requirements

In the reviewed approaches, we identified that they encode a different coordination rule depending on the coordination pattern. Such a coordination rule specifies how concepts from different languages *must* be coordinated. The coordination rule allows approaches to derivate a glue between the elements selected by a correspondence rule. Such a glue specifies how elements in a correspondence *are* coordinated. However, in these approaches, the coordination rule is encoded into a framework/tool and the coordination is expressed in a GPL. This limits the task of a system designer to provide analysis and verification of the coordinated system. To support the heterogeneous development of complex systems, a system designer has to understand well how a system is coordinated and he must be able to provide analysis of the coordinated system. These characteristics motivate the following requirements:

1. Coordination rules should be expressed independently of the implementation of the coordinated languages;

2. Coordination rules should be customizable;

3. Coordination between models should be formally defined.

Let us consider each of these requirements in turn. For an approach to specify coordination patterns, it is important that a new language be easy to add. In this sense, the coordination rule cannot depend on the implementation of coordinated languages. [DNCSSV14] and [BJ01] proposed a coordination rule whose expression is strongly linked to the implementation of the coordinated languages, thus making tedious to add a new language without modifying the whole implementation. Note that this requirement is strongly linked with the need for an explicit language behavioral interface.

In the proposed approaches the coordination rules are encoded in the framework/tools. Thus, a system designer has to modify the current implementation to change the proposed coordination. To be able to specify different coordination patterns, a system designer should be able to modify the coordination rules without altering the whole implementation. This is somehow possible in Ptolemy and ModHel'X but since the semantics of a composite actor and the coordination of its internal actors are mixed, there is a risk of altering the semantics of the actor. In ModHel'X, the semantic adaptation can be modified but only between two particular models.

None of the approaches relies on a formal language to express the coordination. The benefits of having a coordination expressed formally have been highlighted in the ADL community [AG97, LKA+95]. A formal description of the coordination together with a formal description of the semantics of models (at least partial) could provide the verification and validation of the coordinated system. In this context, a language behavioral interface could provide an abstract and formal view of the language behavioral semantics.

3.5 Conclusion

In this chapter, we have presented requirements towards a language that capture the specification of a coordination pattern. To characterize existing approaches, we have proposed a framework which is built of three elements:

- A language behavioral interface;
- Correspondence rules;
- Coordination rules.

Based on this framework, we have studied language coordination approaches and shown how they implemented the elements of the framework. We have identified that a language behavioral interface is a partial representation of the syntax and behavioral semantics of languages for coordination purpose. We have determined that an explicit language behavioral interface has several benefits. For instance, it helps to identify the coordination point thus easing the task to add support to a new language. We have noted that, when the interface is made of events, the specification of the coordination between languages can be done by keeping separately the coordination and the computation concerns. Thus, it avoids altering the coordinated language semantics.

We have identified that correspondence rules specify in intention the correspondences between model elements. We have determined that all approaches rely on a fixed set of correspondence rules. We have noted that none approach relies on a dedicated language (*e.g.*, Epsilon Comparison Language) to express the correspondence rules. We have determined that such a language would ease the task of a language integrator to find similarities between models elements.

Together with correspondence rules, we have shown that approaches rely on a notion of coordination rules that specify the interaction between elements in a correspondence. We have noted that all approaches hide the coordination rules into tool/frameworks in which the coordination is expressed in a general purpose language thus limiting verification and validation of the resulting coordinated system. We have concluded that none approaches leverage on the work done by some Coordination Languages and ADLs in which the coordination is expressed in a formal language.

Based on the requirements presented in this chapter, we propose, in this thesis, a dedicated language named B-COoL to capture coordination patterns between languages. The next chapter presents B-COoL and its implementation.

Chapter 4

B-COoL: the Behavioral Coordination Operator Language

4.1 Introduction

This chapter presents B-COoL, a dedicated language to specify coordination patterns. B-COoL is a particular implementation of the framework presented in Chapter 3. It is based on a language behavioral interface made of event types (*i.e.,* DSE [CDVL$^+$13]) as "coordination points" to drive the execution of languages. These events are used as handles or control points in two complementary ways: to observe what happens inside the model, and to control what is allowed to happen or not. When required by the coordination, constraints are used to forbid or delay some event occurrences. Forbidding occurrences reduces what can be done by individual models. When several executions are allowed (nondeterminism), it gives some freedom to individual semantics for making their own choices. All this put together makes the DSE suitable to drive coordinated simulations without being intrusive in the models.

Coordination patterns are captured as constraints at the language level on the DSE (see Figure 4.1). A *language integrator* defines *Operators* that contain 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. To do so, we rely on the context in which a DSE is defined to selected instances of such DSE. Then, a coordination rule specifies the, possibly timed, synchronization and causality relationships between the instances of DSE selected during the matching. To specify the

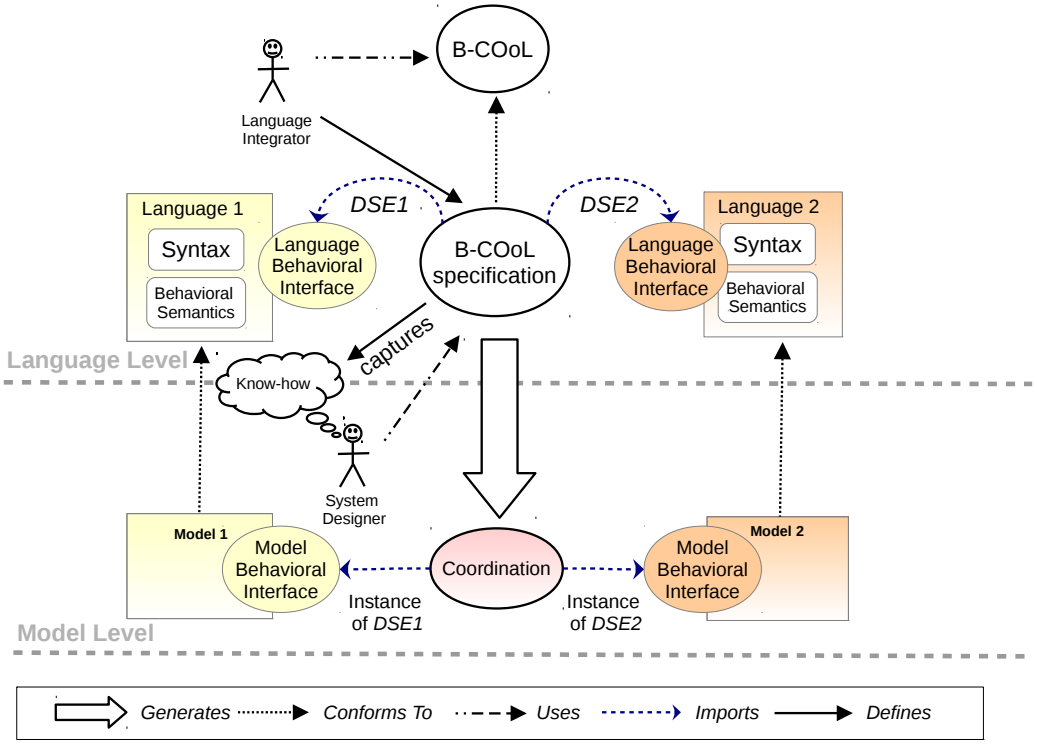


Figure 4.1: Overview of the proposed approach

coordination rule, we rely on a CCSL-based language named MOCCML [DDC+14a]. Once specified in B-COoL, integrators can shared such knowledge thus allowing reusing and tuning of coordination patterns. *System designers* can then use the B-COoL specification to generate an explicit coordination model when specific models are used. The generated coordination model is expressed in CCSL, thus allowing system designers to verify and validate the coordinated system.

In this chapter, we illustrate the use of B-COoL through a simple running example: the coordination of the models of a coffee machine. To build the model, we use two languages: a state-based language named Timed Finite State Machine (TFSM) and an action-based language named Activity. The goal here is to show that we can build an operator in B-COoL between these languages and then use it to coordinate the models of the coffee machine. We use this operator as running example through all the chapter.

This chapter is organized as follows. We begin by introducing the running example; we show the languages, their language behavioral interfaces and the models of a coffee machine. We continue by presenting the abstract syntax and the execution semantics of B-COoL. Then, we show the implementation of B-COoL and its integration into the GEMOC studio. We use the studio to specify the running example, and then, generate the coordination model for the coffee machine system. We

show how the coordination model can be executed and analyzed. To finish the chapter, we compare our approach with coordination languages and frameworks, and we conclude.

4.2 Running Example: Coordination of the Heterogeneous Models of a Coffee Machine

In this section, we present the heterogeneous models of a coffee machine. The coffee machine system is composed of a *Coin Control System* and a *Coffee Algorithm System*. To model the whole system, we use the TFSM and Activity languages, which are developed as proposed in [CDVL+13]. In the following, we present the languages and how we use them to model each subsystem.

The TFSM language is a state machine language augmented with timed transitions. Its abstract syntax is described by a metamodel (see Figure 4.2). A *System* is composed of *TFSMs*, global *FSMEvents* and global *FSMClocks*. Each *TFSM* is composed of *States*. Each state can be the source of outgoing guarded *Transitions*. A guard can be specified either by the reception of an *FSMEvent* (*EventGuard*) or by a duration relative to the entry time in the source state of the transition (*TemporalGuard*). When fired, transitions generate a set of simultaneous *FSMEvent* occurrences.

Listing 4.1: Partial ECL specification of TFSM

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

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 4.2). These DSE are part of the language behavioral interface of TFSM. A partial ECL specification of TFSM is shown in Listing 4.1 where the DSE *entering* and *leaving* are defined in the context of State (Listing 4.1: line 6) while *occurs* is defined in the context of FSMEvent (Listing 4.1: line 4). When a metaclass is instantiated, the corresponding DSE are instantiated; *e.g.*, for each instance of the metaclass *State*, DSE *entering* is instantiated. Each instance of DSE is a MSE.

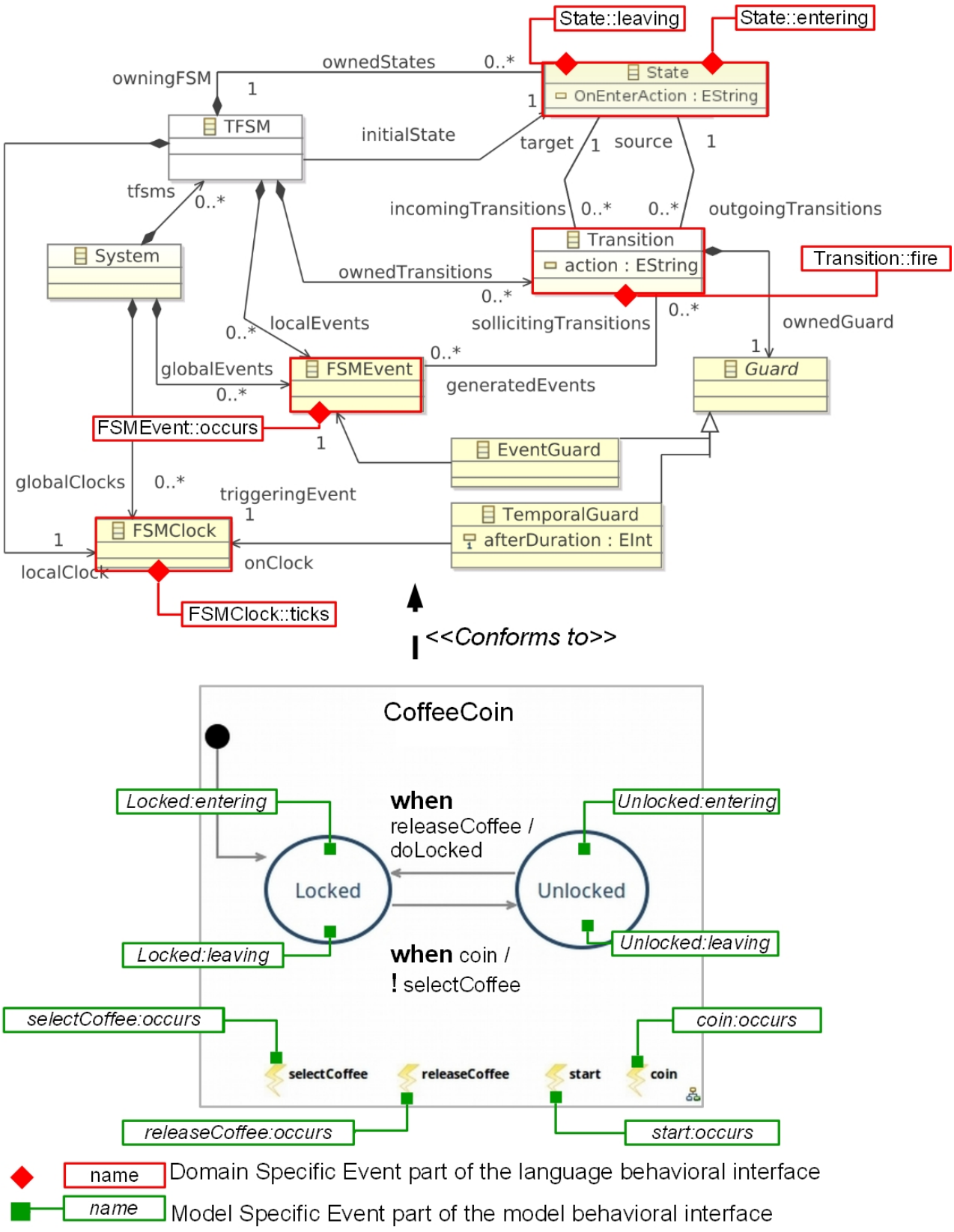


Figure 4.2: (At the top) An excerpt of the TFSM metamodel with a part of its language behavioral interface. (At the bottom) a TFSM model with a part of its model behavioral interface

We use the TFSM language to model the coin control system. We build a TFSM named *CoffeeCoin* (At the bottom of Figure 4.2). The model is composed of two states (*Locked* and *Unlocked*) and four FSMEvents (*selectCoffee*, *releaseCoffee*, *start* and *coin*). For each instance of state, a DSE *entering* and *leaving* are instantiated, *e.g.*, *Locked:entering*, *Locked:leaving*. Also, for each instance of FSMEvent, a DSE *occurs* is instantiated, *e.g.*, *releaseCoffee:occurs*, *selectCoffee:occurs*. In the state Locked, when a coin is inserted (the MSE *coin:occurs* happens), the TFSM becomes Unlocked and the MSE *selectCoffee:occurs* is triggered. In state Unlocked, the release of the coffee (the MSE *releaseCoffee:occurs* happens) makes the TFSM becomes Locked again.

Listing 4.2: Partial ECL specification of Activity diagram

```
1  package activitydiagram
2  context Activity
3    def: startActivity : Event = self
4    def: finishActivity: Event = self
5  context Action
6    def : executeIt : Event = self
7  context Signal
8    def : signalOccurs : Event = self
```

To model the coffee algorithm system, we use the action-based language Activity [CDB+15]. Figure 4.3 shows its partial metamodel. The root element is an *Activity* that is composed of *ActivityNodes* and *ActivityEdges*. Each ActivityNode can be the source of outgoing ActivityEdges. The language behavioral interface of the Activity is 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. In the context of *Action*, the DSE *executeIt* is defined that identifies the execution of an Action. Finally, in the context of *Action*, the DSEsignalOccurs is defined that identifies the occurrence of a Signal. At the bottom of Figure 4.3, the activity named *CoffeeAlgorithm* represents the simple algorithm for preparing coffee. It starts by the action *selectCoffee* that asks the user to select the kind of coffee. After selected it (the MSE *selectCoffee:executeIt* happens), the action *makeCoffee* is executed. Finally, the coffee is released (the MSE *releaseCoffee:executeIt* happens).

To represent the global behavior of the coffee machine, we have to specify how the TFSM *CoffeeCoin* and the activity *CoffeeAlgorithm* interact. More precisely, when the FSMEvent selectCoffee triggers, the Action selectCoffee must be executed. Also, when the Action releaseCoffee executes, the FSMEvent releaseCoffee must be triggered. To coordinate the execution of these models, we have to define some constraints between the MSE *selectCoffee:occurs* and *selectCoffee:executeIt*, and between

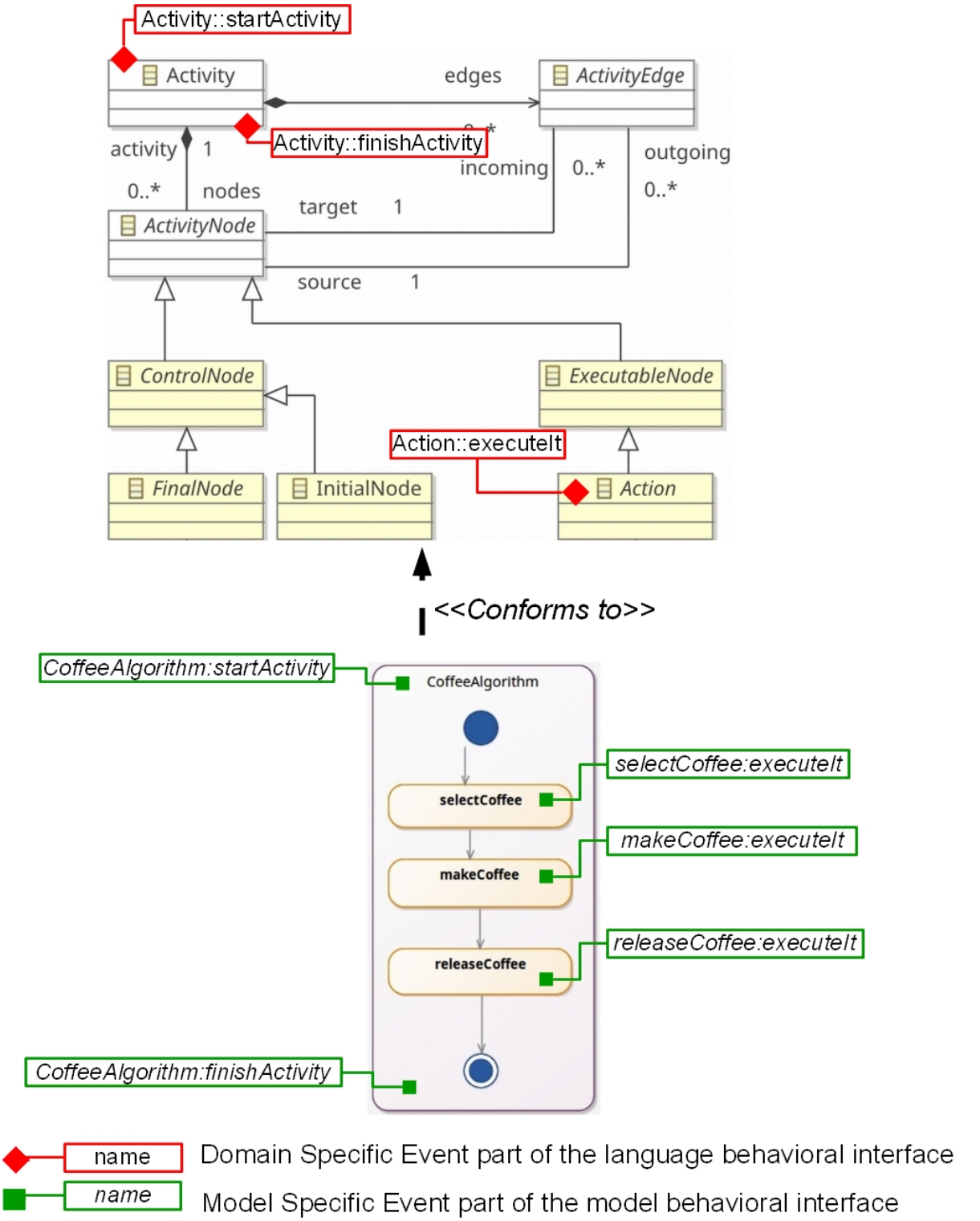


Figure 4.3: (At the top) An excerpt of the Activity metamodel with a part of its language behavioral interface. (At the bottom) an Activity model with a part of its model behavioral interface

releaseCoffee:occurs and *releaseCoffee:executeIt* (see Figure 4.4). We propose to generate these constraints by developing a simple B-COoL operator between the TFSM and Activity languages. We informally define the operator as follows: When coordinating a TFSM and an Activity model, all pairs of FSMEvents and Actions that have the same name must be strongly synchronized, *i.e.,* by using a *rendezvous* relation. This operator is defined for any pair of TFSM and Activity models, and specifies what and how instance of DSE *occurs* and instances of DSE *executeIt* must be coordinated.

In this section, we have presented the TFSM and Activity languages that we used to build the heterogeneous models of a coffee machine. The model is composed of a TFSM named CoffeeCoin and an Activity named CoffeeAlgorithm. To get the global behavior of the coffee machine, we have proposed to specify some constraints between the MSE of both models. To generate such constrains, we informally sketched a simple B-COoL operator between the TFSM and Activity languages. In the next section, we use this simple operator as running example to illustrate the syntax and semantics of B-COoL.

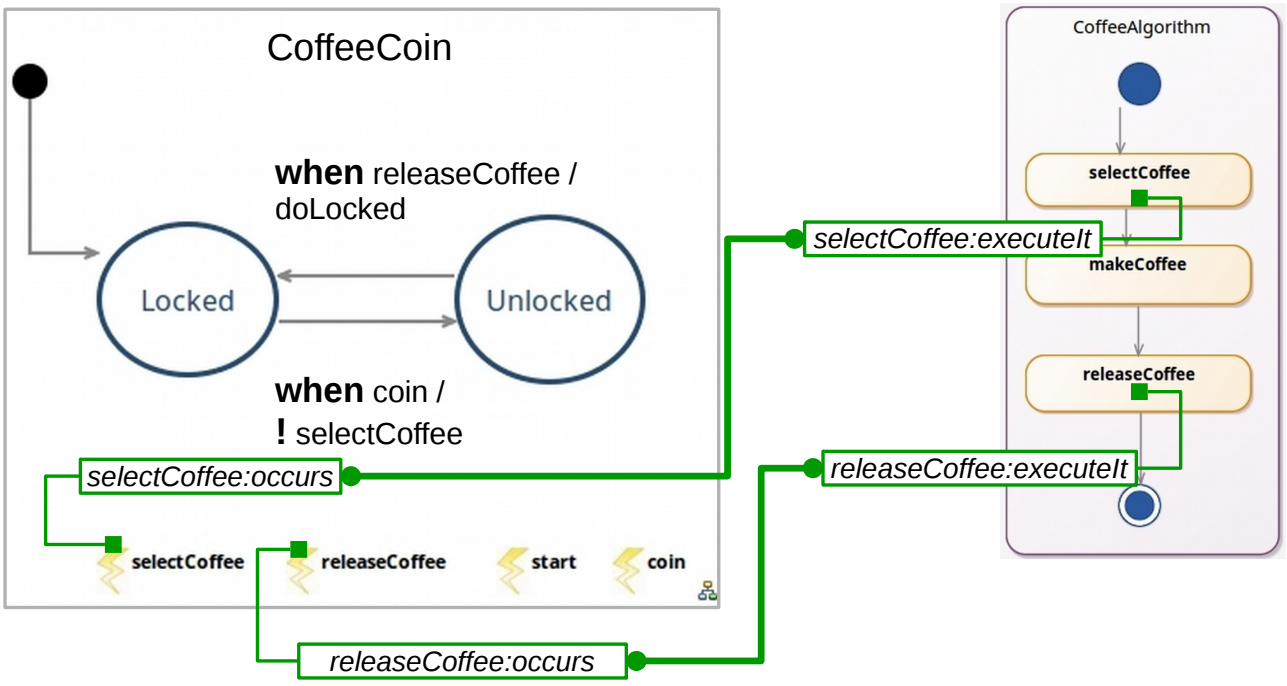


Figure 4.4: Coordination of the models of the coffee machine by constraining the corresponding MSE

4.3 The Language

In this section, we present our language B-COoL. We begin by presenting the abstract syntax and then we continue with the semantics. To illustrate each subsection, we develop the running example

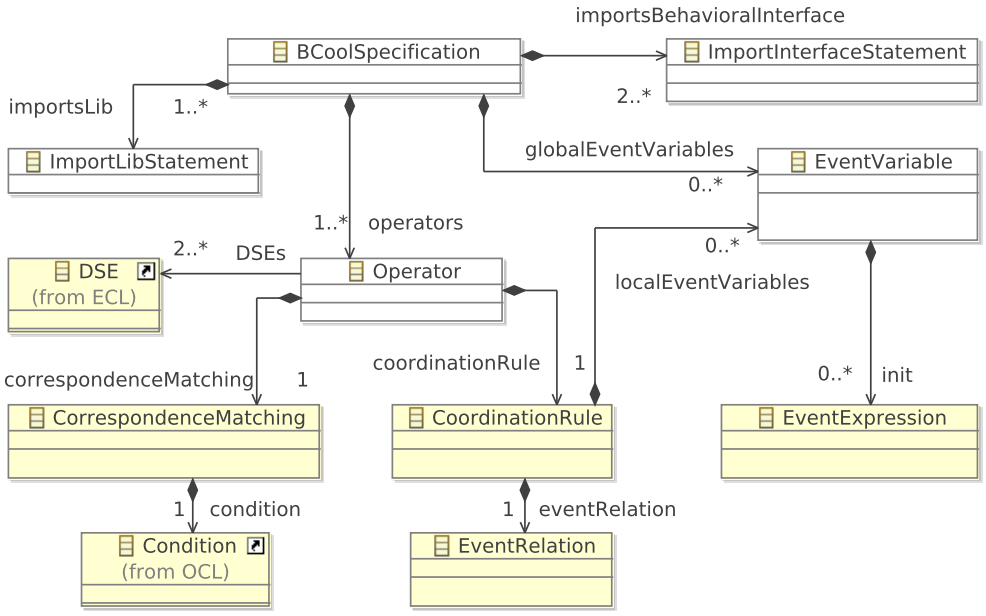


Figure 4.5: Simplified View of B-COoL abstract syntax

operator between the TFSM and Activity languages.

4.3.1 Abstract Syntax of B-COoL

The abstract syntax of B-COoL is defined by its metamodel (see Figure 4.5). The root element is a *BCoolSpecification* that contains *Operators*. Each operator refers to DSE to specify what event types it coordinates. To get the DSE, a B-COoL specification imports language behavioral interfaces (*importsInterfaceStatements*). The number of imported interfaces is related with the number of models that the specification accepts as input. Since the B-COoL specification is applied at least between two models, it imports at least two interfaces. The same interface can be imported several times to allow the specification of operators between homogeneous languages.

To build the running example operator, we need to synchronize FSMEvents and Actions. This is done by coordinating instances of DSE *occurs* and instances of DSE *executeIt*. To specify this in B-COoL, we first define a specification named *TFSMAndActivity* and we import the language behavioral interface of each language, named *activity* and *tfsm* (Listing 4.3: line 3 and 4). Then, the operator *SyncFSMEventsAndActions* is defined, which refers to the DSE *occurs* and *executeIt*, mapped as *FSMEventOccurs* and *ActionExecute* respectively (Listing 4.3: line 5).

Listing 4.3: B-COoL specification of the running example operator between the TFSM and Activity

languages

```
1 BCOoLSpec TFSMAndActivity
2 ImportLib "facilities.moccml"
3 ImportInterface "activitySemantics.ecl" as activity
4 ImportInterface "TFSM.ecl" as tfsm
5 Operator SyncFSMEventsAndActions(ActionExecute:activity::executeIt, FSMEventOccurs:tfsm::
      occurs)
6 when (ActionExecute.name = FSMEventOccurs.name);
7 do   RendezVous(ActionExecute, FSMEventOccurs)
8 end operator
```

In B-COoL, operators are used to specify when and how instances of the referred DSE must be coordinated. To do so, each operator contains a *correspondence matching* and a *coordination rule*. The former is used to select instances of DSE and the latter is used to express how the selected instances must be coordinated.

A *correspondence matching* selects instances of DSE by using a *Condition* that contains an OCL Boolean expression. To express the boolean expression, the context of the DSE can be queried to get a specific syntactic element, *e.g.*, attribute *name*. The boolean condition is thus used to compare the queried elements. The correspondence matching 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 instances of DSE. For instance, for the running example, we query the context of the DSE to get the attribute *name* (Listing 4.3: line 6). Then, the attributes are used as operands for the boolean condition. When an instance of DSE *occurs* and an instance of DSE *executeIt* have the same name, the pairs are selected and the coordination rule is applied.

The *coordination rule* specifies how the selected instances of DSE must be coordinated. To do so, it contains an *EventRelation* and possibly some *EventVariables* (*localEventVariables*).

The event relation is used to restrict the relative order of the occurrences of events used as parameters. Its actual parameters can be instances of DSE, some *EventVariables* and/or constants (*e.g.*, integer constants). For example, the running example operator must specify a strong synchronization between the events. To express that, we use the event relation *Rendezvous* between the selected instances of DSE *occurs* and *executeIt* (Listing 4.3: line 7). This relation accepts two events as arguments and forces the occurrences of these events to happen simultaneously. To illustrate this, Figure 4.6 partially shows the resulting coordination when this operator is used to coordinate the models of the coffee

machine. In the figure, the occurrences of the MSE *selectCoffee:occurs* and *selectCoffee:executeIt* are forced to happen simultaneously.

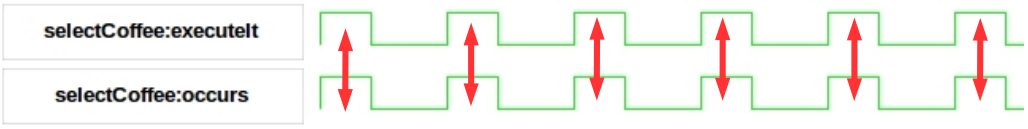


Figure 4.6: Resulting coordination of the coffee machine by using the event relation Rendezvous

Conjointly with an event relation, B-COoL enables the definition of *EventVariables* to express more complex coordination rules. An event variable can be either defined locally within the coordination rule (*localEventVariables*) 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 given parameters. 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.

To illustrate the use of event variables, we suppose the situation in which the coordination between events relies on a global synchronization clock. This is often the case in synchronous digital systems in which a clock signal is used to coordinate the actions in a circuit. In this example, we slightly modify the running example operator by considering that the selected instances of DSE *occurs* (the *sampledEvent*) will be sampled by a global clock (the *trigger*). This results in a new event named *sampledOccurs* that ticks always in coincidence with the *trigger* and each of its occurrences represents the last sample of the *sampledEvent*.

To specify this in B-COoL, we define a new specification named *TimedTFSMandActivity* (Listing 4.4). We first define a global event variable named *globalClock* (Listing 4.4: line 5). This global event variable represents the global synchronization clock. Then, in the coordination rule, we use the globalClock together with the DSE *occurs* to create a new local event named *sampledOccurs* (Listing 4.4: line 9). To initialize this local variable, we use the event expression *SampleOn*. This expression accepts two events as parameters: the *sampledEvent* and the *trigger*. The expression creates a new event by sampling the *sampledEvent* by the *trigger* event. This results in a new event that ticks always after

the *sampledEvent* and coincides with the occurrences of the *trigger* event. In our case, we sample the select instances of the DSE *occurs* (*i.e., sampledEvent*) by the event global clock (*i.e., trigger*), which results in the local event variable *sampledOccurs*. Then, in the coordination rule, we coordinate the instances of DSE *executeIt* and the resulting local event *sampledOccurs* by a Rendezvous (Listing 4.4: line 10).

Listing 4.4: B-COoL specification of an operator that illustrates the use of Event Variables

```
1 BCOoLSpec TimedTFSMandActivity
2 ImportLib "facilities.moccml"
3 ImportInterface "activitySemantics.ecl" as activity
4 ImportInterface "TFSM.ecl" as tfsm
5 Global Event globalClock;
6 Operator SyncFSMEventsAndActions(ActionExecute:activity::executeIt, FSMEventOccurs:tfsm::
      occurs)
7 when (dse1.name = dse2.name);
8 do
9 Local Event sampledOccurs = SampledOn (FSMEventOccurs, globalClock);
10   RendezVous(ActionExecute, sampledOccurs)
11 end operator
```

To illustrate the resulting coordination, we use this specification to coordinate the models of the coffee machine. Figure 4.7 shows the partial resulting coordination between the MSE *selectCoffee:occurs* and *selectCoffee:executeIt* when a global clock is used. The occurrences of the event *selectCoffee:occurs* are sampled by globalClock (in blue in Figure 4.7). This results in a new occurrence of the event *sampledOccurs* (in orange in Figure 4.7). Then, occurrences of the event *sampledOccurs* are strongly synchronized with the event *selectCoffee:executeIt* (in red in Figure 4.7).

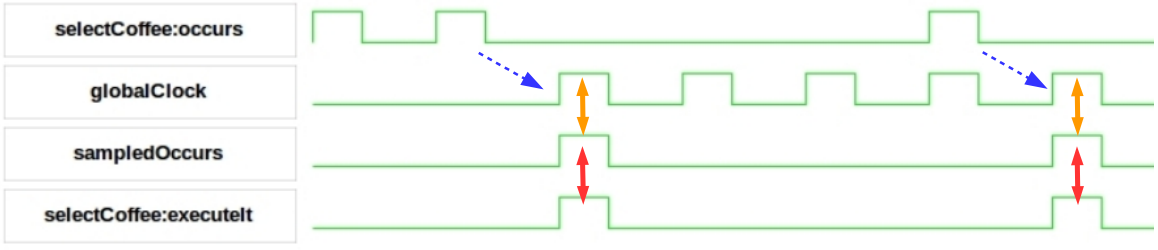


Figure 4.7: Resulting coordination of the coffee machine by using the global event variable globalClock

In this subsection, we have presented the abstract syntax of B-COoL by relying on the running example operator. To specify this operator, we used event expressions and relations that are defined in the library *facilities.moccml* (Listing 4.4: line 2). In B-COoL, expressions and relations are defined in dedicated libraries that must be imported (*ImportedLibStatement* in Figure 4.5). We introduce the

notion of library in the following subsection.

4.3.2 Library

Libraries are used to gather predefined event expressions and relations that can be imported by a B-COoL specification (*ImportedLibStatement* in Figure 4.5). B-COoL relies on MOCCML [DDC+14a] to define libraries of constraints between events. In this subsection, we overview the notion of MOCCML library.

A MOCCML library (*RelationLibrary* in Figure 4.8) is a set of declarations together with their formal parameters. It also contains some definitions, which give the actual behavior of the declarations. There are two categories of constraint definitions: the *Declarative Definitions* and the *Constraint Automata Definitions*. A declarative definition is defined as a set of constraint instances. For more details, we refer the reader to [DDC+14b] that described the declarative part inspired from the CCSL language. A Constraint Automata Definition gives state based support for the definition of relations. This helps a language integrator in the definition of coordination protocols that may be complex like, for instance, the AMBA protocol [Gui01].

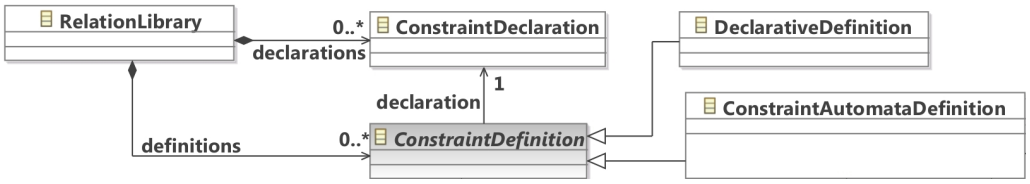


Figure 4.8: Excerpt of MoCCML metamodel

To illustrate the use of MOCCML for the definition of event relations, we propose to rewrite the coordination rule presented in Listing 4.4 by using an automata relation named *RendezvousWithGlobalClock*. The relation accepts three events as parameters: *sampled*, *trigger* and *executeIt*. The *sampled* event identifies the event that will be sampled by the *trigger*. The *executeIt* event identifies the event that will be forced to occur simultaneously with the last sample of the *sampled* event. The automata representation is made of two states: *waitSampled* and *waitTrigger* (see Figure 4.9). In *waitSampled*, the event *trigger* and *sampled* are both allowed to tick but not the *executeIt* event. When the event *sampled* happens, such occurrence will be sampled by the next occurrence of the event *trigger*. This is represented by the transition from *waitSampled* to *waitTrigger*. In this state, the event *trigger* and *executeIt* are forced to happen simultaneously. This represents the sampling of the last occurrence of

the *sampled* event. Conversely, in this state, the event *sampled* is forbidden to occur [1].

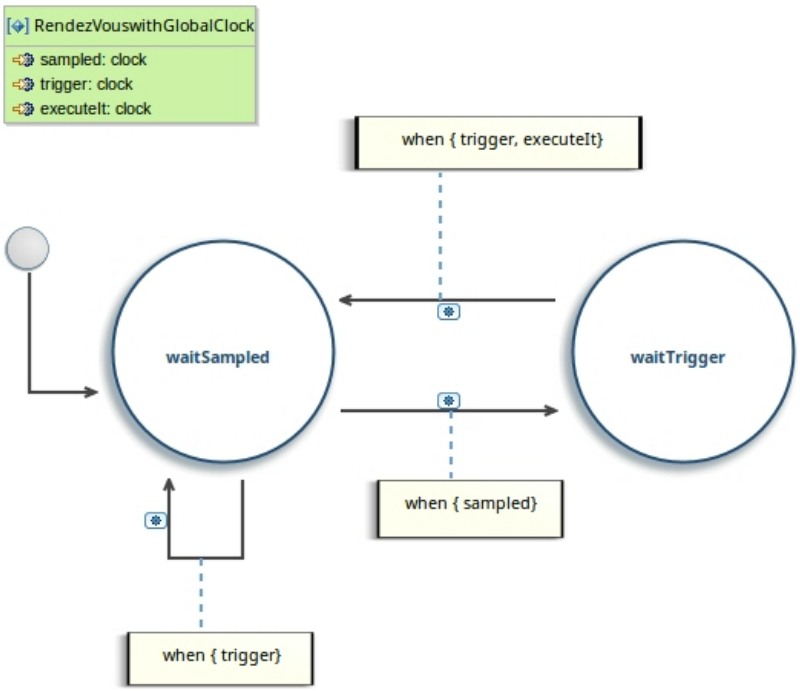


Figure 4.9: State-based representation of the relation *RendezvousWithGlobalClock*

By using this event relation, we rewrite the coordination rule as shown in Listing 4.5; the FSMEventOccurs is the *sampled* event, the globalClock is the *trigger* event and the ActionExecute is the *executeIt* event.

Listing 4.5: B-COoL specification of an operator that illustrates the use of a MOCCML library

```
8 do:
9   RendezVouswithGlobalClock(FSMEventOccurs, globalClock, ActionExecute)
10 end operator
```

Figure 4.10 shows the resulting coordination of the models of the coffee machine by using the relation RendezvousWithGlobalClock. In blue, we show the occurrences of the *selectCoffee:occurs* (sampled event) that are sampled by global clock (trigger event). When an occurrence is sampled, this makes the event *selectCoffee:executeIt* (executeIt event) to occur simultaneously with the occurrence of the global clock (in red in Figure 4.10). This represents the sampling of the last occurrence of the event *selectCoffee:occurs*.

Libraries enable integrators to organize all relevant event relations and expressions by modeling domains. This improves the readability of a B-COoL specification by gathering domain specific relations,

[1] For this example, we chose to forbid the occurrences of the *sampled* event in the WaitTrigger state to prevent missing samples.

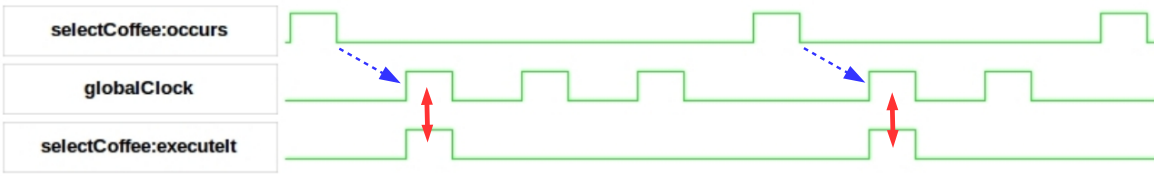


Figure 4.10: Resulting coordination of the Coffee Machine by using the automata relation *RendezvousWithGlobalClock*

which can be reused in other specification. By relying on a MOCCML library, the application of B-COoL operators results in a CCSL specification. We can use CCSL tool (*e.g.*, TimeSquare[DM12b]) to analyze and execute the generated coordination model. In the next subsection, we further explain how the coordination model is generated from a B-COoL specification by presenting the semantics of B-COoL.

4.3.3 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, we rely on the application of the running example operator (see Listing 4.3) between the models of the coffee machine.

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 running example operator is applied between the models of the coffee machine, the first step consists in extracted the MSE of the Activity *CoffeeAlgorithm* and the TFSM *CoffeeCoin* that results in two sets of MSE named respectively $\mathcal{I}_\mathcal{M}1$ and $\mathcal{I}_\mathcal{M}2$ (step 1 in Figure 4.11).

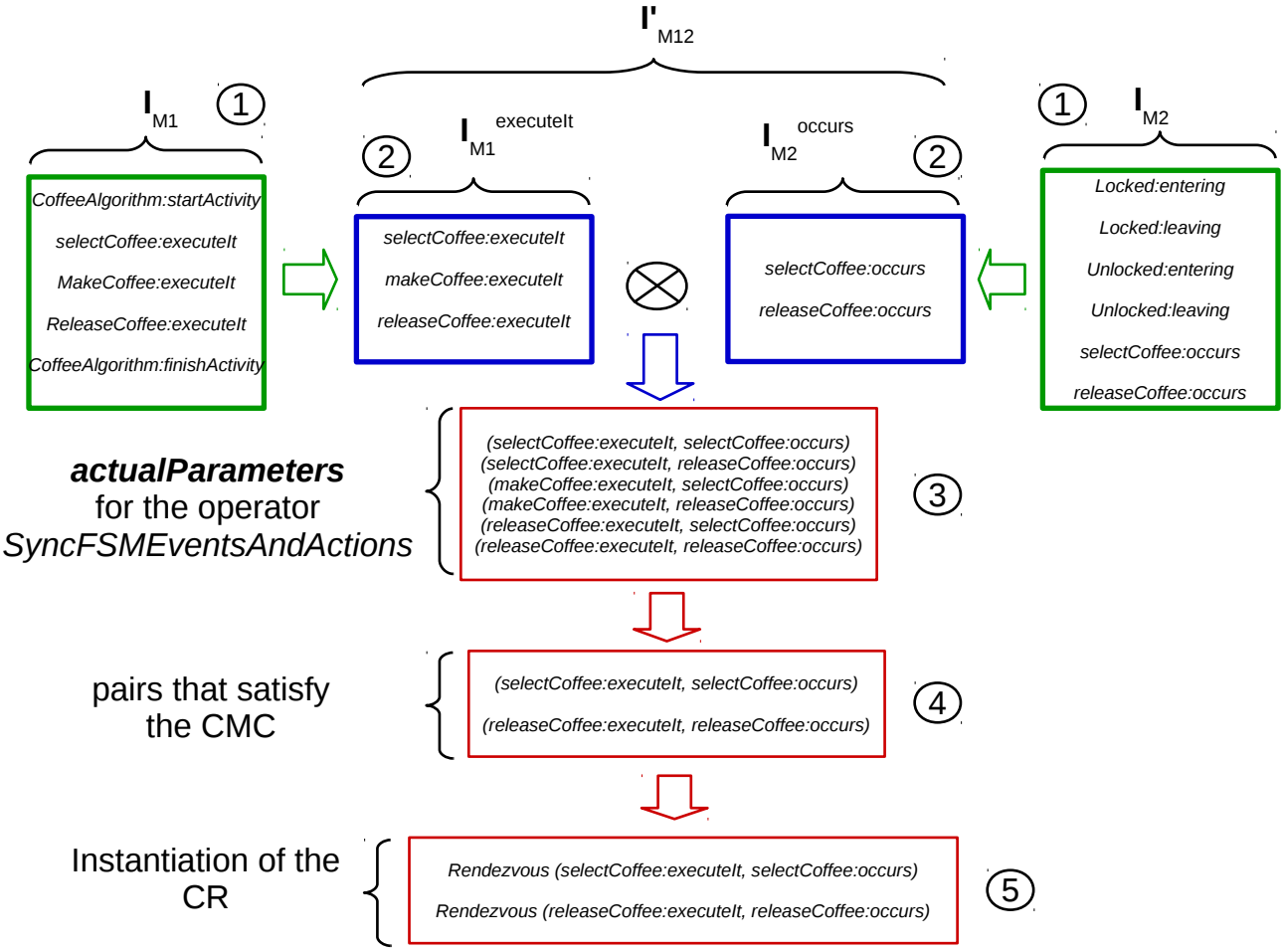


Figure 4.11: Steps in the application of the B-COoL specification between the models of the coffee machine

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 4.11). For instance, the running example operator has as parameters the event types *Action::executeIt* and *FSMEvent::occurs*. Thus, the set $\mathcal{I}'_{\mathcal{M}_\mathcal{I}}$ is composed of two set named $\mathcal{I}_\mathcal{M}1executeIt$ and $\mathcal{I}_\mathcal{M}1occurs$ that corresponds respectively with events of type *Action::executeIt* and *FSMEvent::occurs* (step 2 in Figure 4.11).

Second, a classical Cartesian product is applied on $\mathcal{I}'_{\mathcal{M}_\mathcal{I}}$. It results in a set containing the list of actual parameters to be used with the operator, *i.e.,* each set in the result of the Cartesian product represents the actual parameters of the operator (step 3 in Figure 4.11). 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 running example 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:executeIt*, and *releaseCoffee:occurs* and *releaseCoffee:executeIt* (step 4 in Figure 4.11). 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. 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. For the coffee machine, the event relation rendezvous is instantiated twice; one time for each set in $actualParams$ that satisfies the CMC (step 5 in Figure 4.11).

Currently, the application of a B-COoL operator generates a CCSL specification that represents the coordination. Listing 4.6 shows the partial CCSL specification for the coffee machine. The specification begins by importing the CCSL specification of each model (Listing 4.6: line 3 and 4). Then, the main block contains the coordination specification that is made of two instances of the event relation rendezvous (Listing 4.6: line 9 and 11). 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. Also, we have presented MOCCML for the definition of libraries. In the next section, we present the current implementation of B-COoL as a set of Eclipse plugins into the GEMOC studio.

Listing 4.6: 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  SyncFSMEventsAndActionselectCoffee_executeItselectCoffee_occurs [ RendezVous ]
11      ( ClockA -> "coffeeAlgorithm::selectCoffee_executeIt", ClockB -> "coffeeCoin::
            selectCoffee_occurs")
12      Relation  SyncFSMEventsAndActionsreleaseCoffee_executeItreleaseCoffee_occurs [
            RendezVous ]
13      ( ClockA -> "coffeeAlgorithm::releaseCoffee_executeIt", ClockB -> "coffeeCoin::
            releaseCoffee_occurs")}}
14 }
```

4.4 Implementation

This section presents the implementation of B-COoL into the GEMOC studio[2]; which integrates technologies based on Eclipse Modeling Framework (EMF) [3] 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 [CDB+15].

B-COoL takes advantages of this collaborative environment by adding coordination facilities. Figure 4.12 illustrates the proposed workflow in which a language integrator uses the language workbench

[2] http://www.gemoc.org
[3] http://eclipse.org/modeling/emf/

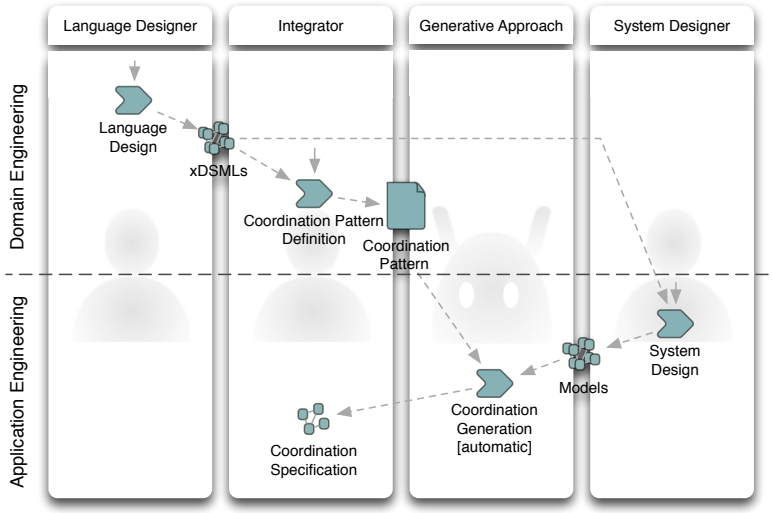


Figure 4.12: The proposed workflow for the heterogeneous development of complex applications

to develop B-COoL operators to specify coordination patterns between languages. Then, a system designer can use these operators in the modeling workbench to coordinate models. In this section, we illustrate the use of the language workbench by developing the running 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 4.13). The B-COoL abstract syntax has been developed using Ecore (*i.e.,* the metalanguage associated with EMF) and the textual concrete syntax has been developed in Xtext [4], thus providing advanced editing facilities. For the running example operator, we use the TFSM and Activity languages that are integrated into the studio. Then, we use B-COoL to specify the Listing 4.3 (Figure 4.13: 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 4.13: step 2), and then, uses it to launch the *Gemoc Coordinated Execution Engine* (Figure 4.13: 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 sys-

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

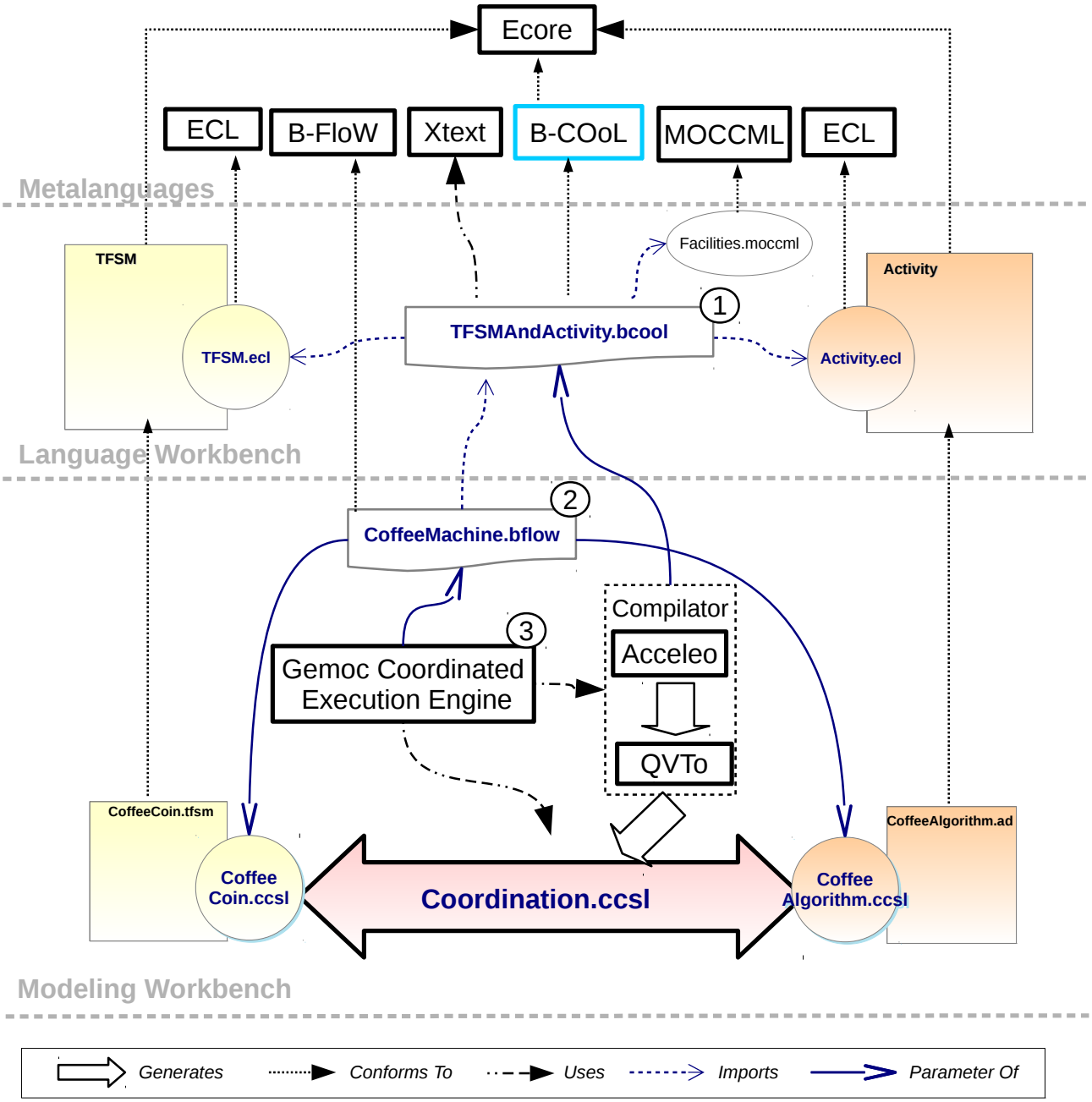


Figure 4.13: Overview of the implementation of B-COoL and its integration into the Gemoc Studio

tem designer to specify how operators of a B-COoL specification are applied on a set of models (Figure 4.13: 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 2). 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 *CoffeAlgorithm.ad* (Listing 4.7: line 3 and 4). Then, the specification contains a *Flow* that defines which operators are used and on which models 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 6 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, this is further discussed in Chapter 5.

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

```
1 BCOoLFlow CoffeeMachine
2 ImportBCOoL  "TFSMAndActivity.bcool" ;
3 Model CoffeCoin "coffeecoin.tfsm"
4 Model CoffeeAlgorithm "coffeeAlgorithm.ad"
5 Flow
6   applies SyncFSMEventsAndActions between (CoffeeAlgorithm, CoffeCoin);
7 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 [5] that translates the B-COoL specification into a QVTo [6] transformation (*Compilator* in Figure 4.13). Then, the Gemoc Coordinated Engine invokes the generated QVTo transformation which takes as parameters the models to be applied and the operators that must be applied. This information is retrieved from the B-FLoW specification. The QVTo transformation is finally applied between the corresponding models thus generating a model of coordination in CCSL.

To execute the coordinated model, the Gemoc Coordinated Execution Engine first initializes the *Gemoc Execution Engine* of each individual model. These engines compute the next valid step for each

[5] http://www.eclipse.org/acceleo/
[6] https://projects.eclipse.org/projects/modeling.mmt.qvt-oml

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 in a set of global valid steps.

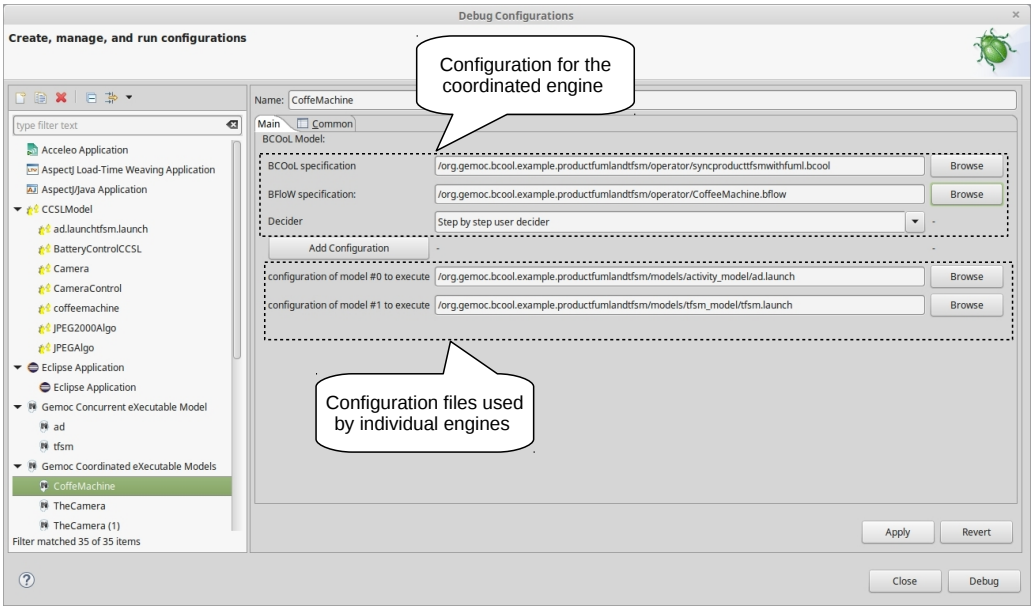
To illustrate the use of the Coordinated Execution Engine, we coordinate and execute the models of the coffee machine. First, we configure the launcher that contains information about the B-COoL specification, the B-FLoW specification and the configuration launcher for each model. In Figure 4.14(a), by clicking on *Debug*, the execution is launched. Then, each individual engine is initialized together with the engine for the coordination (Figure 4.14(b): 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 4.14(b): point 3). Also, the workbench provides the animation of models (Figure 4.14(b): point 4).

The modeling workbench provides tools to analyze the resulting CCSL specification (Figure 4.14(b): 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 4.15 shows the resulting state-space exploration for the coordinated model of the coffee machine[7]. Each state represents a valid step in the execution. In the transitions, the guards contain the events that tick simultaneously when a transition is taken. For example, in red, the figure 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.

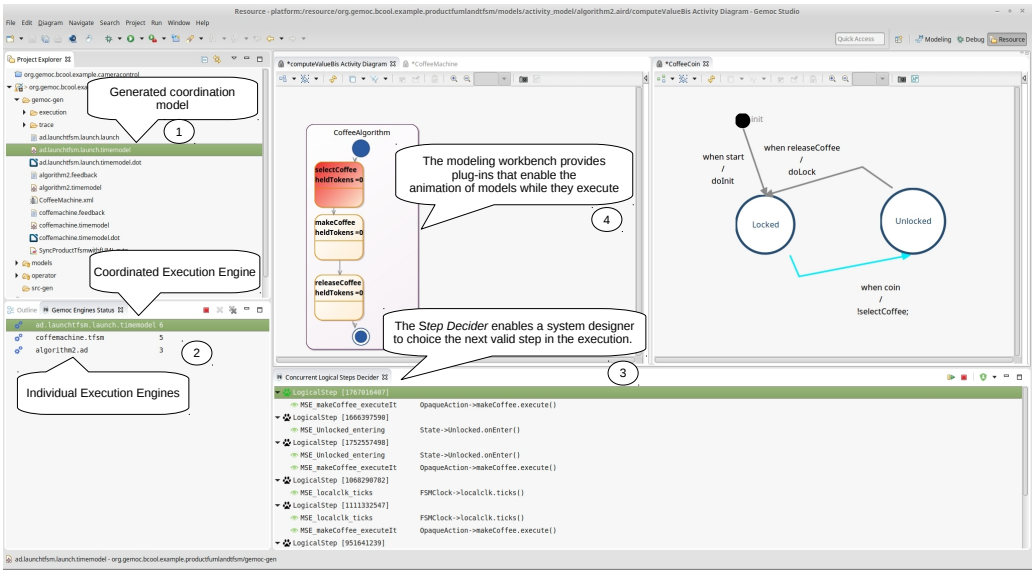
The project B-COoL is hosted on Github[8] as part of the GEMOC project[9], thus making the source code publicly available. B-COoL is currently integrated into the GEMOC studio[10]. To try the coffee machine example, the reader needs to download the studio and then to follow the tutorial from the companion website[11]. In addition, the website contains more examples with full descriptions.

In this section, we have presented the implementation of B-COoL into the GEMOC studio. The

[7] The graph in DOT format can be downloaded from the companion website.
[8] `http://github.com`
[9] `http://github.com/gemoc/coordination`
[10] The reader can download studio the from `http://gemoc.org/studio-download/`
[11] `http://timesquare.inria.fr/BCOoL`



(a) Configuration of the launcher of the Gemoc Coordinated Execution Engine



(b) Coordinated Execution and animation of the models of the coffee machine

Figure 4.14: Coordinated Execution of models by using the Gemoc Coordinated Execution Engine

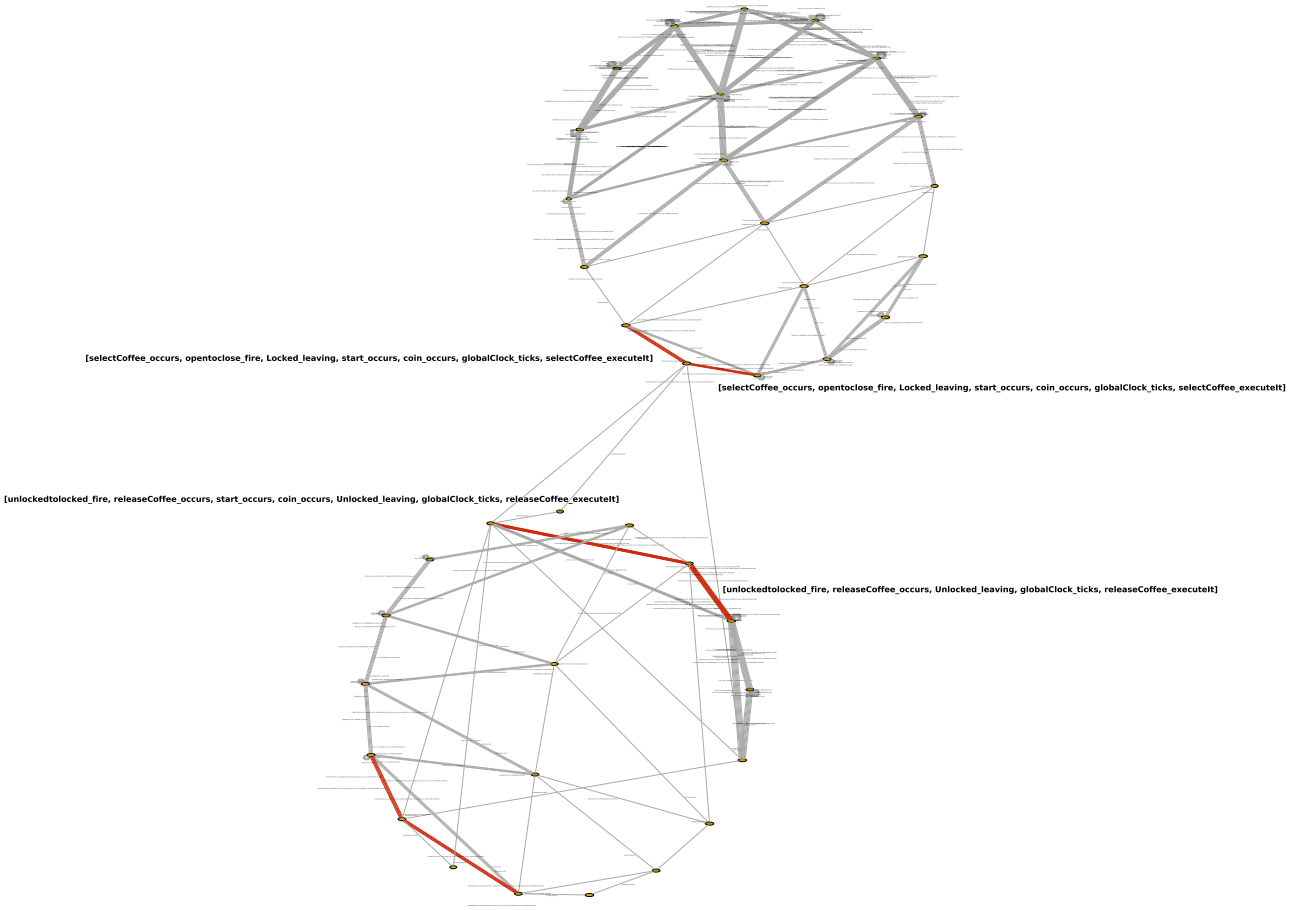


Figure 4.15: State space representation of the coordinated model of the coffee machine, encoding the set of valid schedules. The transitions in red contain the events forced to happen simultaneously by the coordination.

workbench eases the tasks of language integrators and system designers to coordinate models. Integrators can develop operators between languages that a system designer can use to coordinate, execute and validate their models. In the next section, we compare our approach with coordination languages and coordination frameworks.

4.5 Evaluation

In this section, we evaluate the benefits of B-COoL in terms of four criteria. Also, we use these criteria to compare our approach with Coordination Languages and Coordination Frameworks. These criteria are:

1. Definition of a coordination pattern between languages;

2. Generation or synthesis of the coordination between models;

3. Analysis capabilities of the coordinated system;

4. Tooling support.

Regarding to first point, in B-COoL, the definition of a coordination pattern between languages is based on operators. In particular, coordination rules explicitly define the semantics of the resulting coordination. Notice that an integrator can vary the semantics of the resulting coordination by only modifying the coordination rules of the operators. In frameworks like Ptolemy, an integrator is unable to change the proposed coordination pattern without 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 [DNCSSV14], where the transformation needs to be changed. Since this state-of-the-art approaches 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 language integrator experts thus easing the understanding and adaptation of the B-COoL specification.

Concerning the coordination between models (point number two), the definition of domain specific coordination operators enables the generation of the coordination between models. The manual coordination of models (as proposed by coordination languages) requires a system designer that specifies each relation. The reader can notice that the number of relations increases with the number of model elements involved in the coordination. Our proposition is to leverage this task for the system designer at the language level and then to generate all the required relations accordingly.

Regarding system execution and verification (point number three), 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 execution of the coordinated system [BHLM02]. Furthermore, in Ptolemy and ModHel'X the coordination is expressed in Java so that the verification and validation remains limited. In our approach, by relying on CCSL to express the coordination, a system designer is able to provide execution and verification of the coordinated system.

In terms of tooling support (point number four), current coordination frameworks like Ptolemy [BHLM02] and ModHel'X [BH08] provide a dedicated environment to develop and coordinate heterogeneous models. They rely on a common syntax based on actors and semantics given by Models of Computation. In addition, they enable the system designer to hierarchically coordinate models. The environments include a graphical editor, an execution engine, plotters and so on. These environments, however, are ad-hoc solutions to manage both the development and the coordination of heterogeneous models. Differently, in our approach, the studio is the integration of several plug-ins that deal with different aspects of the heterogeneous development of models, *e.g.*, the GEMOC studio for the design and implementation of DSMLs, the Sirius animator for graphical representation, TimeSquare for the analysis of model execution. Our approach takes advantages of this collaborative environment, and it provides the means for modeling coordination.

Coordination frameworks do not offer a clear separation between the task of a language integrator and the task of a system designer. They only focus on the task of a system designer. Differently, in our approach, we provide a language workbench to develop B-COoL operators and capture coordination patterns between languages. Then, for system designers, we provide a modeling workbench to coordinate, execute and validate models. Therefore, the integrated studio has managed the tasks of both stakeholders by providing dedicated workbenches.

4.6 Conclusion

In this chapter, we have presented B-COoL, a dedicated (meta)language that enables integrators to capture the specification of coordination patterns between heterogeneous languages. B-COoL is a particular implementation of the framework presented in Chapter 3. To illustrate the syntax and the semantics of B-COoL, We defined a simple operator between the TFSM and Activity languages. We have presented the current implementation of B-COoL into the GEMOC studio that provides a language workbench and a modeling workbench. In the language workbench, a language integrator can develop B-COoL operators. Then, in the modeling workbench, a system designer can use these operators to coordinate and execute their models. Furthermore, the modeling workbench provides tools to analyze the coordinated model. We have shown the use of the language workbench by developing the running example operator, and then, we have shown the use of the modeling workbench by coordinating and verifying the heterogeneous model of the coffee machine. To finish the chapter, we have compared B-COoL with coordination frameworks and coordination languages by relying on four

criteria. From this analysis, we have determined that our approach makes a clear separation between the task of a language integrator and a system designer. Furthermore, this separation is supported by dedicated workbenches. In the next chapter, we propose to validate our approach by using as use case the coordination of the heterogeneous models of a surveillance camera system. We rely on the integrated studio to develop a set of coordination operators, and then, we use them to coordinate and execute the system.

Chapter 5

Validation

5.1 Introduction

In this chapter, we validate our approach by defining different coordination patterns between the TFSM and Activity languages, which were introduced in Chapter 4. To motivate and explain the patters, we rely on the coordination of the models of a surveillance camera system. The system is composed of a *Camera Control Encoder* and a *Battery Sensor*. We model the system by using the TFSM and Activity languages, which results in heterogeneous models that need to be coordinated.

To coordinate these models, we propose in this chapter the definition of three B-COoL coordination operators between the TFSM and Activity languages. These operators are used to capture three coordination patterns between these languages. One operator synchronizes FSMEvents and Signals by relying on their names. A second operator specifies a hierarchical coordination in which the entering and leaving of states is synchronized with the execution of an activity. In addition, we propose a third operator that specifies a timing coordination. More precisely, we specify that the execution of an activity is atomic from point of view of the TFSM language. Thus, during the execution of an activity, the time in the TFSM does not elapse. In this chapter, we show how to use B-COoL to specify and customize these coordination patterns. Also, we show how the resulting coordination can be used for analysis.

We organize this chapter as follows. We begin by presenting the heterogeneous models of a surveillance camera system. To get the global system behavior, we propose to coordinate these models by using

three operators in B-COoL. Then, we use these operators to coordinate the models, and to execute the coordinated system by using the GEMOC studio. Finally, we conclude.

5.2 Use Case: Coordination of the Heterogeneous Models of a Surveillance Camera System

This section presents the heterogeneous models of a surveillance camera system. To model the system, we use the TFSM language for modeling the control subsystems (*i.e.,* the controller for the camera encoder), whereas for the dataflow aspects of the system we use the Activity language (*i.e.,* the encoding algorithms and the battery sensor). In the following, we present each subsystem and how we model them by using the corresponding language. We finish this section by presenting the necessary coordination between these models to get the global system behavior. In the following section, we propose three operators to generate the coordination between these models.

The video surveillance system is composed of a *Battery Sensor* and *Camera Encoder Control*. The camera encoder control takes pictures by using either the *JPEG2000* or *JPG* algorithm depending on the status of the battery. The TFSM named *CameraEncoderControl* represents the camera encoder control (see Figure 5.1). When the TFSM model is in state *JPEG2000Encoder*, the JPEG2000 algorithm is used. When in state *JPEGEncoder*, the encoding algorithm is replaced by a mere JPEG algorithm. Each state has a temporal transition that happens every 40 ticks of the *ms* local clock. Each tick of this clock represents one millisecond. Thus, in each state, the camera takes 25 frames per second which is a reasonable frame rate for video surveillance [1]. The transition from one state to another is done when either the *BatteryisHigh:occurs* event or the *BatteryisLow:occurs* event occurs, depending on the current state.

In the camera encoder, states represent either the JPEG encoder or the JPEG2000 encoder. Roughly speaking, the JPEG[2] algorithm encodes a picture by grouping it in blocks which are transformed by a *forward transformation*. Each block of pixels is transformed to frequency coefficient using either a *Fourier transform* (JPEG) or *Wavelet transform* (JPEG2000). The transformed blocks are *quantized* and then passed to a *Run-Length coder* which compress the data. At the end, the block is *transmitted*. We model these algorithms by using the Activity language: the activity named *doJPEG* (see Figure 5.3) represents the JPEG algorithm and the activity named *doJPEG2000* (see Figure 5.2)

[1] `https://en.wikipedia.org/wiki/Closed-circuit_television_camera`
[2] `http://www.jpeg.org`

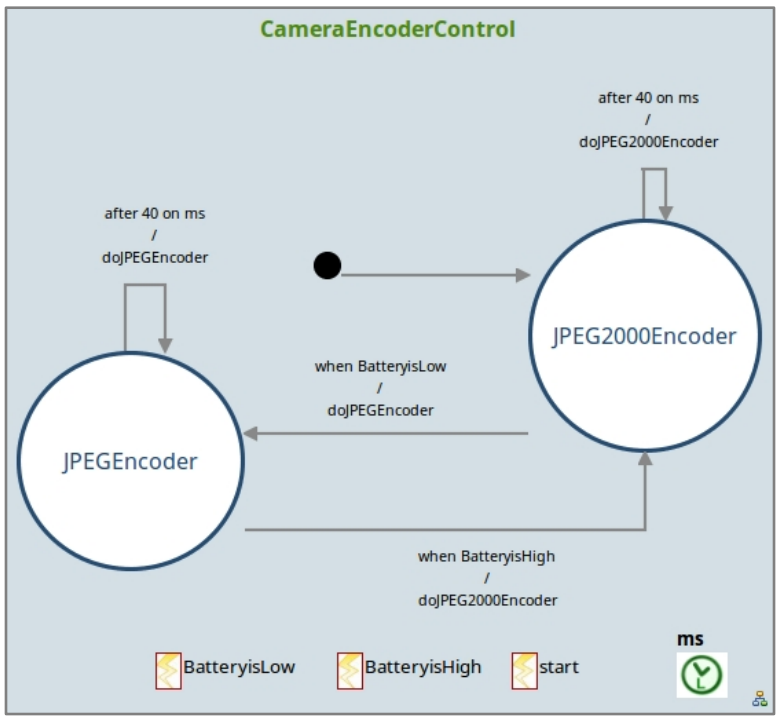


Figure 5.1: Representation of the Camera Encoder Control by using a TFSM

represents the JPEG2000 algorithm .

The camera control encoder is powered by a battery. When the battery is low, the battery sensor makes the camera use the *JPG* algorithm, thus reducing the quality of the picture but also the energy consumption [RSU04]. When the battery is high, the JPEG2000 algorithm is used instead. The activity diagrams named *BatterySensor* (see Figure 5.4) represents the simple algorithm implemented in the battery sensor. Depending on the status of the battery, the algorithm sends either the signal *BatteryisLow* or *BatteryisHigh*, which correspond with the occurrence of the MSE *BatteryisLow:signalOccurs* and *BatteryisHigh:signalOccurs* respectively.

To get the global behavior of the surveillance camera system, we have to specify how these models are coordinated. Figure 5.5 shows the proposed coordinated model. The activity BatterySensor and the TFSM CameraEncoderControl are coordinated by relying on Signals and FSMEvents. More precisely, the trigger of the Signals *BatteryisHigh* and *BatteryisLow* must be synchronized with the occurrences of the FSMEvents *BatteryisHigh* and *BatteryisLow*, *i.e.,* synchronization between the MSE *BatteryisHigh:signalOccurs* and *BatteryisHigh:occurs*, and the MSE *BatteryisLow:signalOccurs* and *BatteryisLow:occurs*. In addition, the execution of the states of the TFSM CameraEncoderControl must be coordinated with the activities doJPEG and doJPEG2000. To coordinate these models, we have to coordinate the entering and leaving of a state (*i.e.,* DSE *entering* and *leaving*) and the execution of the

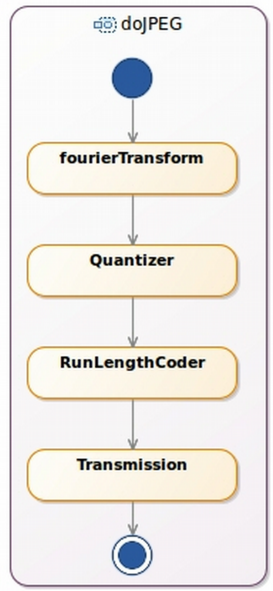


Figure 5.2: Representation of the JPEG encoding algorithm by using an activity diagram

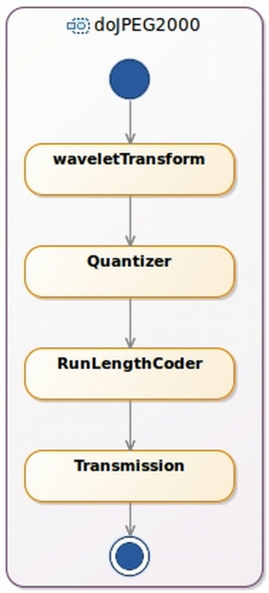


Figure 5.3: Representation of the JPEG2000 encoding algorithm by using an activity diagram

corresponding activity (*i.e.,* DSE *startActivity* and *finishActivity*). Finally, to ensure that the camera fulfills the required frames per second, we have to specify how the time in the TFSM elapses during the execution of activities. In other words, we have to specify how the MSE *ms:ticks* is coordinated with the execution of the activities that represents the encoding algorithms, *e.g.*, *doJPEG2000:startActivity* and *doJPEG2000:finishActivity*, *doJPEG:startActivity* and *doJPE:finishActivity*. We propose to generate the necessary constrains to coordinate these models by using B-COoL operators to specify three coordination patterns between these languages. In the next section, we present the definition of these operators.

5.3 Definition of Coordination Operators between the TFSM and Activity Languages

This section presents the B-COoL specification of three operators between the TFSM and Activity languages: *SyncFSMEventsAndSignals*, *StartActivityWhenEnter* and *AtomicActivity*. In the following, we describe each operator and we discuss their semantics. We then present the B-COoL specification according with the chosen semantics.

The *SyncFSMEventsAndSignals* operator differs from the running example operator because it synchronizes FSMEvents and Signals. Whereas in the running example the occurrences of FSMEvents

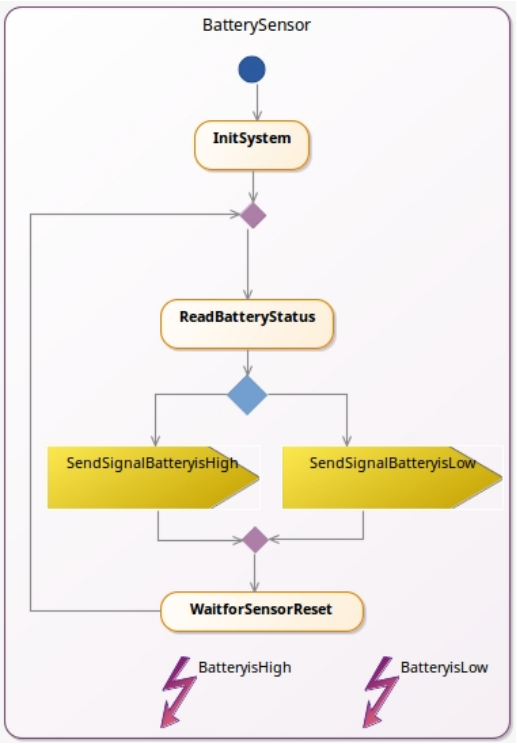


Figure 5.4: Representation of the Battery Sensor by using an activity diagram

were synchronized with the starting of Actions, here they are synchronized with the occurrences of Signals, *i.e.,* by coordinating instances of DSE *occurs* and *signalOccurs*. This operator only requires a slight modification of the specification presented in the running example (see Listing 4.3). The only modification is the type of the DSE used in the operator (see Listing 5.1: line 1). The rest of the definition (*i.e.,* the correspondence matching and coordination rule) is unchanged (see Listing 5.1: line 2 and 3). We want to highlight that the adaptation has been done only by identifying the new DSE to be constrained. This should also be the case for other coordination pattern.

Listing 5.1: B-COoL specification of the *SyncFSMEventsAndSignals* operator

```
1 Operator SyncFSMEventsAndSignals(SignalOccurs:activity::signalOccurs, FSMEventOccurs:tfsm::
    occurs)
2 when (SignalOccurs.name = FSMEventOccurs.name);
3 do   RendezVous(SignalOccurs, FSMEventOccurs)
4 end operator
```

The *StartActivityWhenEnter* operator specifies a hierarchical coordination pattern between the TFSM and Activity languages, unlike hierarchical coordination frameworks where the semantics is hidden, this operator explicitly specifies how the hierarchical coordination is implemented. In our case, we chose the semantics in which entering in a specific state of a TFSM model triggers the execution of a given Activity. When leaving a state, several semantic variation points may be chosen. The

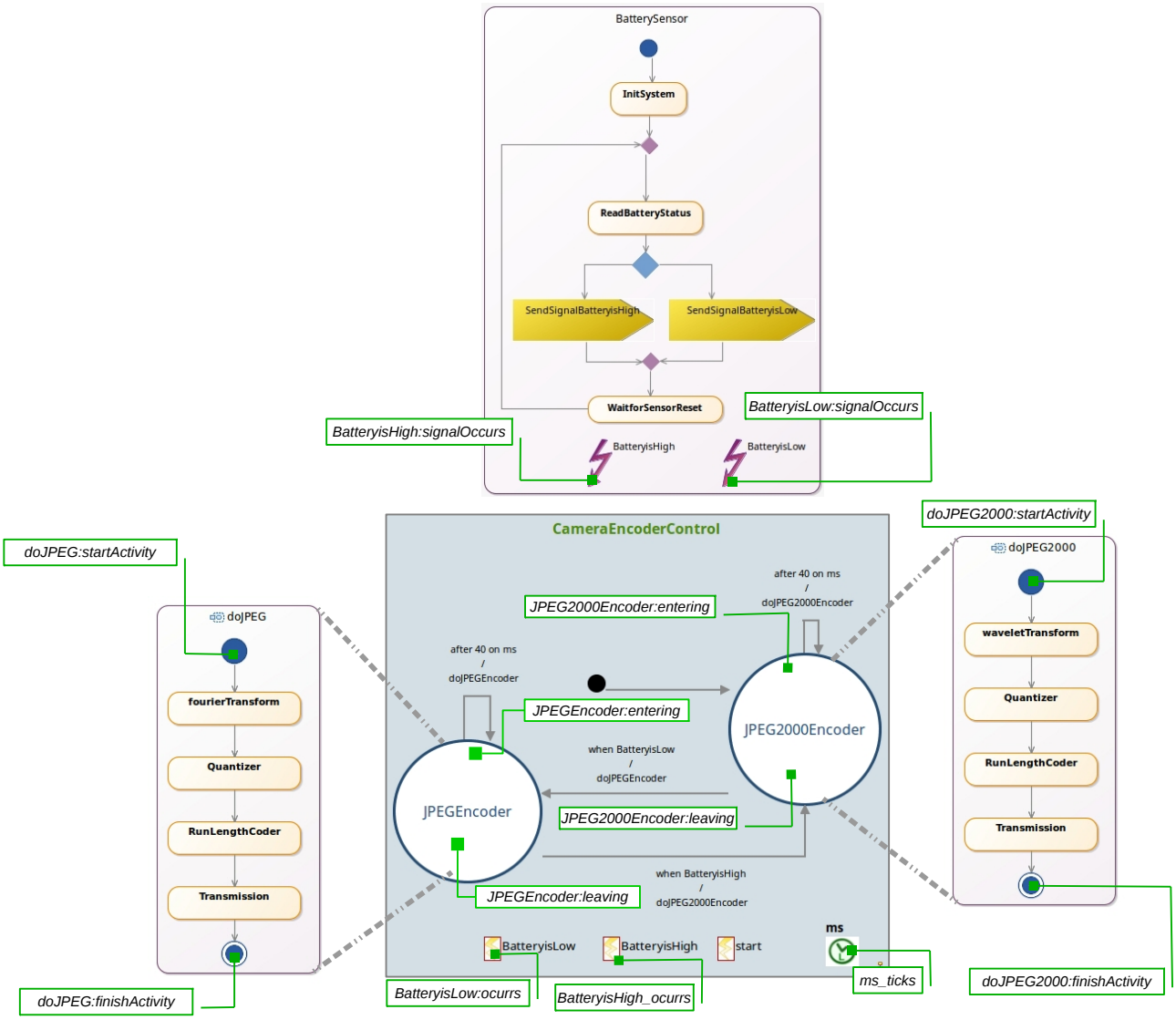


Figure 5.5: Coordinated model of a surveillance camera system and a partial representation of the model behavioral interface

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 our case, we chose non-preemptive transitions because the activity should not be interrupted until the information has been sent. Thus, we ensure that the activity executes at least one time.

We define the *StartActivityWhenEnter* (Listing 5.2: line 1) operator that coordinates the entering and leaving of a state with the execution of an activity. The entering into a state is identified by the DSE *entering* defined in the context of State. Instances of such DSE have to be coordinated with instances of the DSE *startActivity*. Similarly, leaving a state is identified by DSE *leaving* and finishing

an activity is identified by DSE *finishActivity*. To identify pairs of such events, the correspondence matching selects instances of DSE *startActivity* and *finishActivity* by using their context (Listing 5.2: line 2). The pairs selected identify the starting and finishing of the same activity. Similarly, we select instances of DSE *entering* and *leaving* that correspond with the same state. Besides, we select the activities that represent a state by comparing the *OnEnterAction* defined in the states and the name of the activities. OnEnterAction is a string defined in the context of State (see Figure 4.2) 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 (Listing 5.2: line 2).

Listing 5.2: B-COoL specification of the *StartActivityWhenEnter* operator

```
1  Operator StartActivityWhenEnter (activityStart : ad::startActivity , activityStop : ad::
       finishActivity , enterState : tfsm::entering , leaveState : tfsm::leaving)
2  when ((activityStart = activityStop) and (enterState = leaveState) and (activityStart.name
       = enterState.onEnterAction));
3  do
4    ExecuteActivityNonPeemptive (enterState , leaveState , activityStart , activityStop)
5  end operator;
```

To coordinate the selected instances of DSE, the coordination rule uses the event relation *ExecuteActivityNonPeemptive* (Listing 5.2: line 4), which is defined by using MOCCML (see Figure 5.6). The relation takes four events as parameters: the events *modeEnter* and *modeLeave* that represents respectively the entering and leaving of a state; and the events *startActivity* and *finishActivity* that represents respectively the starting and finishing of an activity. The state-based representation is made of two states named *waitEnterState* and *waitFinishActivity*. In waitEnterState state, the events modeEnter, modeLeave and startActivity are allowed to tick. Only when modeEnter and startActivity happen simultaneously, the state *waitFinishActivity* is reached and the event modeLeave is forbidden to occur, *i.e.,* the state cannot be left. In *waitFinishActivity* state, only the finishActivity is allowed to happen. When this event occurs, *i.e.,* the activity has finished, the state *waitEnterState* is reached and the modeLeave is allowed to occur. The use of this relation in the coordination rule results in transitions that cannot preempt the execution of the internal activities. The entering a state makes the activity to start to execute synchronously. Then, only after the activity has finished, the state can be left. This makes the internal activity executes at least once before leave the state. We want to highlight that an integrator can vary the semantics of the coordination by only modifying the event relation in the coordination rule. For instance, by modifying the event relation, the transition may become preemptive.

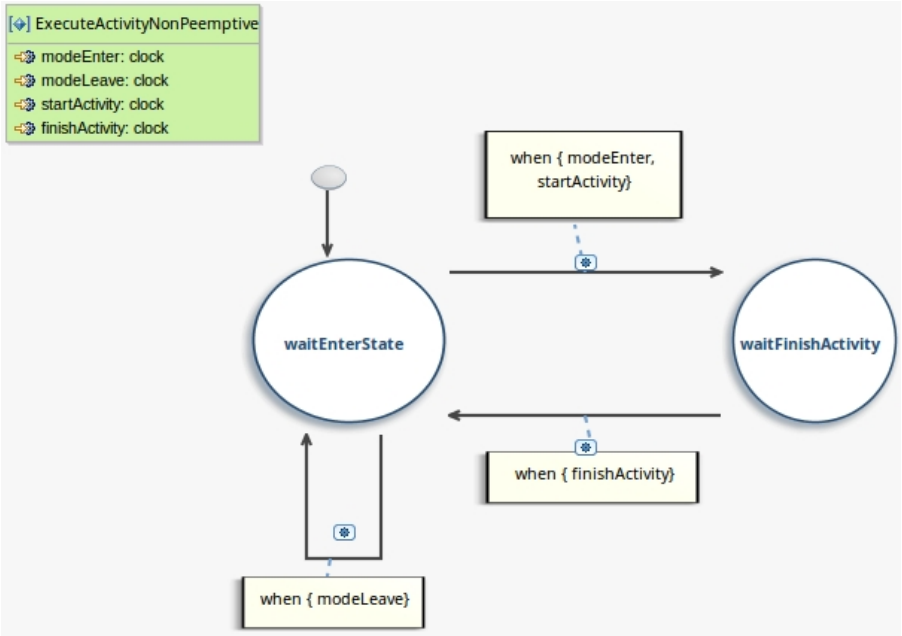


Figure 5.6: *ExecuteActivityNonPeemptive* event relation

In the *AtomicActivity* 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 an activity. In these languages, the time is represented differently. In the TFSM language, each state machine has a *localClock* used to measure the time while the Activity language is untimed. The local clock is a *FSMClock*, which defines a DSE named *ticks* whose occurrences represent a physical time increment. In the Activity language, the duration of activities can be represented as the time between the DSE *startActivity* and DSE *finishActivity*. 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*. Again, several semantic variation points may be choose. For instance, the coordination rule could express that the execution of the activities takes a fixed amount of time. In our example, we propose to enforce the execution of the activity to be atomic with respect to the time in the TFSM model. As a result, there are no occurrences of the DSE ticks of the corresponding local clock during the execution of the activity.

To specify this in B-COoL, we define the operator *AtomicActivity* (Listing 5.3: line 1) that specifies how time is consumed during the execution of the activities. The correspondence matching selects instances of DSE *startActivity* and *finishActivity* by using their context. Thus, it selects the instances that belong to the same activity. Note that the correspondence matching does not filter instances of DSE *ticks*, as a result, the selected activities will be atomic respect to all the clocks in the TFSM.

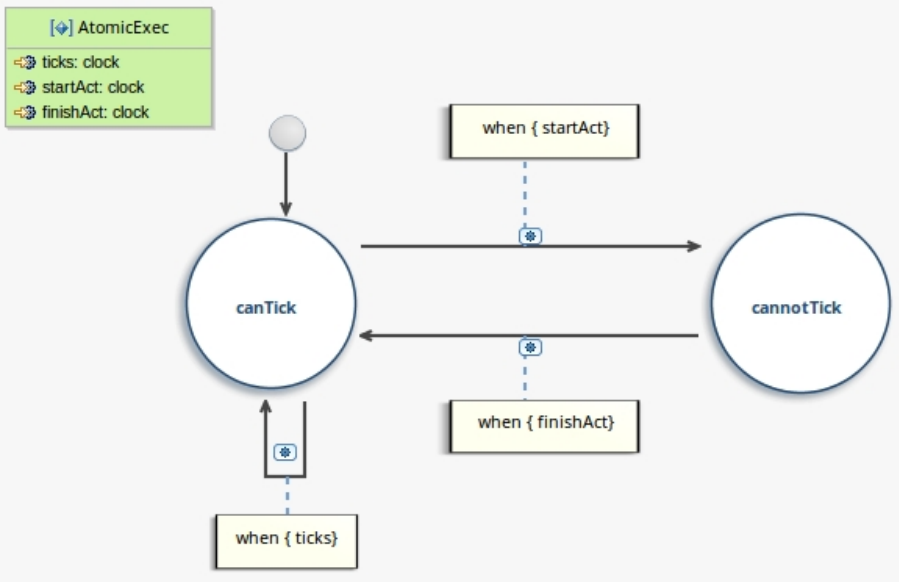


Figure 5.7: *AtomicActivity* event relation

Listing 5.3: B-COoL specification of the *AtomicActivity* operator

```
1  Operator AtomicActivity (activityStart : ad::startActivity , activityStop : ad::
       finishActivity, timeTicks : tfsm::ticks)
2  when (activityStart = activityStop );
3  do
4    AtomicExec (activityStart, activityStop, timeTicks)
5  end operator;
```

To express the coordination rule (Listing 5.3: line 4), we use the event relation *AtomicExec* which makes the execution of the activities atomic with respect to the local clock of the TFSM. The event relation accepts three events as parameters: *ticks*, *startAct* and *finishAct* (see Figure 5.7). While the event ticks represents the ticking of the local clock, the startAct and finishAct events represent respectively the starting and finishing of an activity. In *canTick* state, the event ticks is allowed to occur, thus making the time elapse. When the startAct event happens, *i.e.,* the activity starts to execute, the *cannotTick* state is reached and the event ticks is forbidden to occur, *i.e.,* the time in the TFSM does not elapse while the activity executes. Only when the event finishAct happens, the *canTick* state is reached, and *ticks* is allowed to occur again. In the operator, we use this relation to make the execution of the activities atomic, *i.e.,* there is no occurrence of the DSE ticks of the corresponding local clock during the execution of the activity.

In this section, we have used B-COoL to define three coordination operators that deal with both control and timing aspects of the coordination between the TFSM and Activity languages. Unlike hi-

erarchical coordination frameworks where the semantics is hidden, these operators explicitly specified how the coordination is implemented. More precisely, coordination rules explicitly define the semantics of the resulting coordination. Furthermore, we ease the definition of relations by using MOCCML, a dedicated language to express constraints between events. Thus, an integrator can vary the semantics of the coordination by only modifying the event relation in the coordination rule. Frameworks like Ptolemy do not support such a variation without changing the current implementation of the framework. This means modifying the implementation of a *director* written in Java, which needs a good knowledge of the framework. In our approach, we are using a language dedicated to language integrator experts thus easing the understanding and adaptation of the B-COoL specification. In the next section, we use these operators to coordinate and execute the models of the surveillance camera system.

5.4 Use of the Operators in a Surveillance Camera System

In this section, we use the operators previously defined to coordinate the heterogeneous models of a surveillance camera system. To do so, we propose to use B-FLoW to specify how the operators are applied on the models that compose the surveillance camera system. In the following, we first present the B-FLoW specification, and then we use it together with the Gemoc Coordinated Execution Engine to execute the system.

To coordinate the surveillance camera system, we use B-FLoW to specify how the operators *SyncFSMEventsAndSignals*, *StartActivityWhenEnter* and *AtomicActivity* are applied between the different models. We define a B-FLoW specification named *CameraSystem* (Listing 5.4: line 1) that begins by importing the B-COoL specification (Listing 5.4: line 2), and then, it specifies the models to be coordinated (Listing 5.4: line 3 to 6). The specification contains a *Flow* that defines which operator is applied and on which models. We describe in turn what operator is applied and what coordination is generated.

First, the operator SyncFSMEventsAndSignals has to be applied between the activity BatterySensor and the TFSM CameraEncoderControl (Listing 5.4: line 8). This results in the synchronization of the corresponding Signal and FSMEvent by relying on theirs names. The application of this operator results in *two* instances of the CCSL relation rendezvous between the MSE *BatteryisHigh:signalOccurs* and *BatteryisHigh:occurs*; and the MSE *BatteryisLow:signalOccurs* and *BatteryisLow:occurs*.

Then, the operator StartActivityWhenEnter has to be applied between the TFSM CameraEncoderControl and the activity doJPEG (Listing 5.4: line 9), and between the TFSM and the activity doJPEG2000 (Listing 5.4: line 10). This results in a synchronization between the entering and leaving of the states JPEGEncoder and JPEG2000Encoder, and the execution of the activities doJPEG and doJPEG2000. The application of this operator results in *two* instances of the MOCCML relation ExecuteActivityNonPeemptive between the MSE *doJPEG:startActivity*, *doJPEG:finishActivity*, *JPEGEncoder:entering* and *JPEGEncoder:leaving*; and the MSE *doJPEG2000:startActivity*, *doJPEG2000:finishActivity*, *JPEG2000Encoder:entering* and *JPEG2000Encoder:leaving*.

Finally, to coordinate the time between the TFSM and the activities, the operator AtomicActivity has to be applied between the TFSM CameraEncoderControl and each activity (Listing 5.4: line 11 and 12). This results in a synchronization between the local clock *ms* of the TFSM CameraEncoderControl and the execution of the activities. The application of this operator results in *two* instances of the MOCCML relation AtomicActivity between the MSE *doJPEG2000:startActivity*, *doJPEG2000:finishActivity* and *ms:ticks*; and the MSE *doJPEG:startActivity*, *doJPEG:finishActivity* and *ms:ticks*.

Listing 5.4: B-FLOW specification for the Surveillance Camera System

```
1 BCOoLFlow CameraSystem
2 ImportBCOoL  "TFSMAndActivity.bcool" ;
3 Model BatterySensor "batterysensor.ad"
4 Model CameraEncoderControl "cameraencodercontrol.tfsm"
5 Model doJPEG "doJPEG.ad"
6 Model doJPEG2000 "doJPEG2000.ad"
7 Flow
8   applies SyncFSMEventsAndSignals between (BatterySensor, CameraEncoderControl);
9   applies StartActivityWhenEnter between (CameraEncoderControl, doJPEG);
10  applies StartActivityWhenEnter between (CameraEncoderControl, doJPEG2000);
11  applies AtomicActivity between (CameraEncoderControl, doJPEG);
12  applies AtomicActivity between (CameraEncoderControl, doJPEG2000);
13 end Flow;
```

The resulting model of coordination is thus composed of six relations, *i.e.*, two CCSL relations and four MOCCML relations. In the modeling workbench, we use the B-FLOW specification together with the Gemoc Coordinated Execution Engine to execute the coordinated system. Figure 5.8 illustrates the coordinated execution of the models. In the figure, four Gemoc execution engines are launched that correspond with the four models of the surveillance camera system. Also, one coordinated execution engine is launched that corresponds with the coordination between these models.

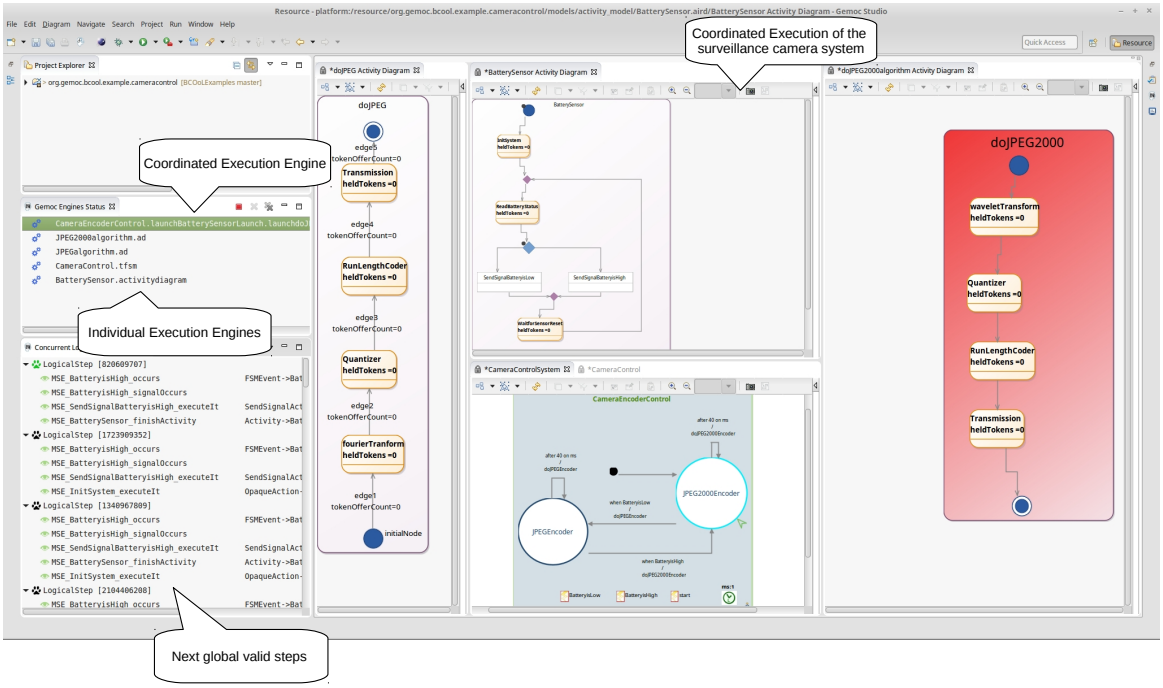


Figure 5.8: Coordinated execution of the models of the surveillance camera system by using the Gemoc studio

Figure 5.9 illustrates the partial timing output of the execution of the camera. As a result of the coordination, the MSE *BatteryisHigh:occurs* and *BatteryisHigh:signalOccurs* are strongly synchronized (in red in Figure 5.9). When the camera entered into the JPEG2000Encoder state (in magenta in Figure 5.9), the activity doJPEG2000 executes and the time in the TFSM does not elapse (in cyan in Figure 5.9). Only after the activity has finished, the time can elapse thus the MSE *ms:ticks* is allowed to tick. Also, only after the activity has finished the state is allowed to be left. In this example, the state-space graph of the coordinated system is not finite. Thus, we could not get results on the scheduling state-space of the coordinated system.

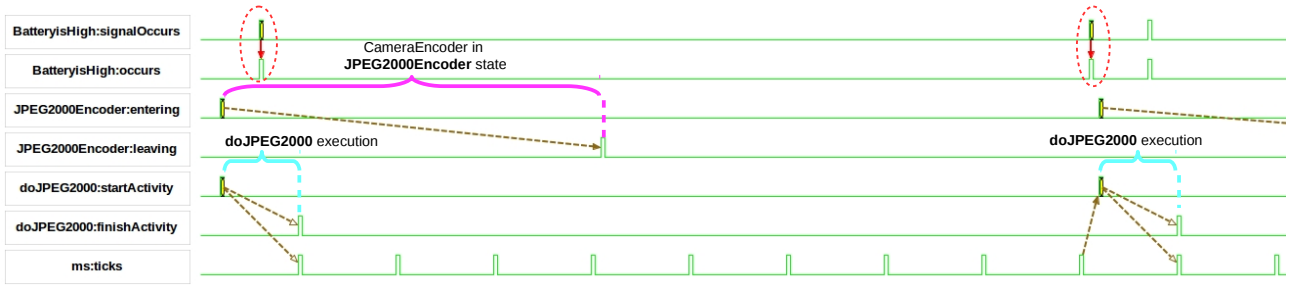


Figure 5.9: Partial resulting timing output of the surveillance camera system

In this section, we have shown the use of B-FLOW to specify how the operators defined in the previous section are applied between the models of the camera. Then, we used this specification in the modeling workbench to execute and verify the coordinated system. By relying on CCSL to express

the coordination, we could provide execution and verification of the coordinated system. We want to highgligt that in coordination frameworks these tasks are limited since they express the coordination in a GPL like Java in Ptolemy. The reader can find the complete example in the companion website, which contains the models together with a detailed procedure to execute and verify them. In addition, the site includes a video that shows the complete workflow by using the GEMOC studio.

5.5 Conclusion

In this chapter, we have validated the use of B-COoL by defining three operators between the TFSM and Activity languages. These operators were used to specify three coordination patterns between these languages. We defined the operator *SyncFSMEventsAndSignals* that specifies a coordination between FSMEvents and Signals by relying on their names. We defined this operator by slightly modifying the running example operator. Then, we specified the operator *StartActivityWhenEnter* that specifies a coordination between the entering and leaving of a state and the execution of an activity. Finally, we defined the operator *AtomicActivity* to specify how the time is coordinated between these languages.

We have noted that some of these coordination patterns are common in coordination frameworks, *e.g.*, Ptolemy, ModHel'X. However, their specification is encoded inside a tool and expressed in a GPL, thus limiting the customization of the coordination and the validation of the coordinated system. Conversely, in our approach, operators are explicitly defined by using B-COoL, which eases the customization of the operators.

We have used these operators to coordinate the heterogeneous models of a surveillance camera system. To do so, we used B-FloW to specify how the operators are applied between the different models. Then, in the modeling workbench, we used the B-FloW specification to execute the coordinated system.

During this thesis, we have defined more operators that can be found in the companion website. For example, we developed the *SyncronizedProduct* operator between the TFSM languages [VLDCM15]. This is an operator that coordinates TFSM models by synchronizing FSMEvents. In a more recent work [CBC+16], we have investigated the use of B-COoL into *Capella* [3]. The Capella modeling

[3] `https://www.polarsys.org/capella/`

workbench is an Eclipse application implementing the ARCADIA [4] method providing both a DSML and a toolset which is dedicated to guidance, productivity and quality. The Capella DSML aggregates a set of 20 metamodels and about 400 meta-classes involved in the five engineering phases (aka. Architecture level) defined by ARCADIA. In this context, B-COoL has been used to define an operator named *ModeEnteringActivateFunctionalChain* that coordinates the action of entering and leaving a mode with the activation of a functionalChain.

In the next chapter, we summarize the most important contributions of this thesis and we propose various perspective paths that could be a guide to continue this work.

[4] `https://www.polarsys.org/capella/arcadia.html`

Chapter 6

Conclusion

In this thesis, we have proposed B-COoL, a dedicated (meta)language that enables language integrators to specify coordination patterns between heterogeneous languages. By relying on this specification at language level, system designers can automatically generate the coordination between heterogeneous models. Such a model of coordination can be used to verify and execute the coordinated system. In the following, we summarize the main findings of this thesis. We finish this chapter by proposing future works.

6.1 Overview

This thesis has focused on the coordination of heterogeneous behavioral models to provide execution and verification of the global system behavior. We studied the state-of-art approaches that aim at getting the global representation of a heterogeneous system, in both structural and behavioral way.

First, we have studied approaches that propose to compose models into a new model. The composition of models has been automated by looking for correspondences between heterogeneous models, and then composing them into a new model. We have noted that most of the approaches consider structural correspondences and only few consider also the behavior of models to find similarities. These approaches have achieved to automate the composition of models by expressing the composition at language level. However, these approaches only consider structural models.

Then, we have studied approaches that compose languages to get a new language. Most of these approaches focus on the composition of the syntax of languages into a new syntax. Only Semantics

Anchoring [CSN05] proposes to compose behavioral semantics. We have determined that these approaches proposed to model a heterogeneous system by relying on a single language which results from the composition of other languages.

We have presented a different kind of approaches that propose to coordinate the behavior of models. They propose to specify the interaction between heterogeneous behavioral models by using a dedicated language, *i.e.,* Coordination Languages and ADLs. To ease the task of a system designer, ADLs' community has successfully identified the need of connector types. Thus, a system designer has only to instantiate and bind connector types as needed by its architecture. Furthermore, in some of these approaches, the coordination is expressed in a formal language thus providing reasoning about the coordinated system. We have noted, however, that in coordination languages and ADLs, the coordination is specified between particular models. With the increasing number and heterogeneity of the components, the manual coordination of models can quickly become difficult and error prone. We have determined that, 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.

Then, we have presented Coordination Frameworks and ad-hoc solutions which 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. These approaches have captured the specification of a behavioral coordination pattern inside a tool/framework to automate the instantiation and binding of connector types. These approaches go one step beyond Coordination Languages and ADLs by leveraging on the know-how of a system designer. However in these approaches, we have noted that the knowledge about how a system is coordinated is hidden inside a tool, thus limiting reasoning. Moreover, they express the coordination in a general purpose language thus limiting the verification and validation of the coordinated system. Based on this state-of-art approaches, we have determined that:

- The specification of coordination patterns between languages is specified at language level;
- The specification of coordination patterns should be done by using a dedicated language;
- The coordination between models should be generated by using a formal language to enable system designers to verify and validate the coordinated system.

We have determined that the lack of a systematic way to specify a coordination pattern makes existing approaches ad-hoc and not flexible. Furthermore, this prevents a wider adoption of this sort of approaches. To understand how existing tools and frameworks have achieved to capture a given coordination pattern, we proposed a framework for the specification of coordination patterns. We have determined three building-blocks:

- A Language Behavioral Interface, which exposes partial information about the syntax and the behavioral semantics of languages for coordination purpose;
- A Correspondence Rule, which specifies what and when elements from different languages must be coordinated;
- A Coordination Rule, which specify how the selected elements must be coordinated.

Then, we have used this framework to compare existing approaches. Based on this comparison, we stated the requirements to make them more flexible and better customizable. More precisely, we proposed the requirements for a language to specify coordination patterns. These requirements tend to improve existing approaches in the customization of coordination patterns between heterogeneous languages.

Based on these requirements, we proposed B-COoL, a dedicated (meta)language to capture coordination patterns between languages. B-COoL is a particular implementation of the proposed framework. We based on a language behavioral interface made of event types, *i.e.,* DSE. These event types act as coordination points on the language behavioral semantics. Then, we proposed to specify coordination patterns by using operators that define a correspondence matching that selects instances of DSE, and a coordination rule that defines how the selected instances of DSE must be coordinated. Using B-COoL, the know-how of a system designer is made explicit, stored and shared in libraries. Furthermore, such coordination patterns, expressed at the language level, can be applied on particular models to automatically generate the corresponding coordination model in CCSL language. The use of the formal CCSL language to express the coordination allowed us to provide execution and verification of the coordinated system.

We implemented B-COoL as a set of Eclipse plugins integrated into the GEMOC studio. The studio proposes a language workbench and a modeling workbench. In the language workbench, an language integrator can develop operators between languages. Then, in the modeling workbench, a

system designer can use these operators to automate the coordination of models. We have proposed a dedicated language named B-FLOW that allows a system designer to specify what operators are applied on a set of models. Then, from this specification, a system designer can generate a model of coordination in CCSL so that the whole system can be executed and verified.

To validate our approach, we have presented the coordination of the heterogeneous models of a surveillance camera system. We modeled the different parts of the system by using the TFSM and Activity languages. To coordinate these models, we specified in B-COOL three coordination patterns that we captured by using three operators. Unlike coordination frameworks in which the semantics of the coordination is hidden, we have explicitly specified these patterns by using a dedicated language. We used this specification to generate the coordination for the surveillance camera system, and then we executed the overall system.

6.2 Future Works

B-COOL provides some perspectives to extend and to improve the work carried out in this thesis. We list the propositions we consider essential to the continuity of this work:

- **Extending B-COoL to support the coordination of data:** System designers build coordination models to specify how models interact. The interactions between models can rely on events but also on data, *i.e.,* data-driven coordination. In this case, a model exchanges data with another model. With B-COOL, we have managed interactions that rely on events, *i.e.,* control-driven coordination. To support the specification of coordination patterns that involve the exchange of data between models, we have to extend B-COOL to support data driven coordination. More precisely, we have to add a way to specify when a value of a variable in a model is carried to a new value in a variable in another model. In B-COOL, such information should be encoded in a *data coordination rule*. In addition, the current coordination rule should be used to synchronize the events associated with the change of the value of a variable. Thus, the resulting model of coordination would have some CCSL specification but also some code that represents the exchange of data between models. During the execution of models, such an exchange of data could be handled by the heterogeneous engine. Thus, a configuration file should be also generated to tell the engine what and when data from different models must be exchanged.

- **Generalizing the specification of correspondences by using a dedicated language:** In B-COoL, the correspondence matching can capture implicit and explicit correspondences between elements of models. Currently, the explicit correspondence is only supported if one of the metamodel is modified to refer concepts from a different metamodel. To avoid such modification, we need a language to specify correspondences between concepts from different languages without modifying the metamodels (*e.g.*, like in megamodels [BJV04]). This is, for instance, the case of Atlas Model Weaver [DDFV08], a tool that enables to create links between model elements. In [BBF+06], such links are used for model composition. Once links between models are established, a model transformation written in ATL can compose the model elements into a composed model. In our case, these links could be used to identify the elements to be coordinated. This is interesting, for instance, in the case of allocation in which there is a model of the hardware, a model of the application and a mapping model that is often generated by using some heuristic. Such a mapping model contains the links between the application model and the hardware model that represent the deployment. The mapping model could be the input for a B-COoL specification to generate the coordination model that represents the deployment between the hardware model and the platform model.

- **Using B-COoL for the synthesis of the orchestrator for Co-Simulation of Functional Mock-up Units:** The analysis of Cyber-Physical Systems (CPS) involves the use of physical components (for instance described by differential equations evaluated according to the continuous time simulation paradigm) but it also involves the use of cyber components (usually relying on discrete time or discrete event simulation paradigms). Each language comes with existing tooling and several simulation tools and techniques are needed for CPS simulation and analysis. In this context, the Functional Mock-up Interface (FMI) [fmi] is a tool independent standard to support the co-simulation of dynamic models. The FMI standard provides a well defined specification and API to integrate heterogeneous simulation components. One key requirement for Co-Simulation via FMI is to develop a Master Algorithm that orchestrates the steps of Co-Simulation [fmi14]. For instance, the master algorithm has to control the data exchange and the time advancement among individual simulations. The FMI standard, however, does not describe or limit the implementation of the master algorithms. Currently, it is specified each time a particular system has to be co-simulated, which remains tedious and error prone. An interesting future work would be to generate such algorithm from a B-COoL specification and the particular model used. Currently, the control and timing coordination is well managed by

B-COoL. However, to exploit the data exchange proposed by the FMI standard, it is mandatory to first extend B-COoL to support data-driven coordination.

- **Generalizing the specification of coordination patterns between existing modeling languages:** The development of heterogeneous system is currently done by using different existing modeling languages like Matlab, SDL or Modelica, which are developed by using very different technologies. To specify coordination patterns between these languages, we have to investigate how to add support for existing modeling languages in B-COoL. Currently, to capture the specification of coordination patterns, we rely on partial information about the syntax and the behavioral semantics of languages that is contained in the language behavioral interface. It is thus mandatory that current modeling languages expose part of its syntax and behavioral semantics. The language behavioral interface may be a standard way to do it. In other words, modeling languages could provide a language behavioral interface for coordination purpose.

Bibliography

[AA02] F. Arbab and Farhad Arbab. A channel-based coordination model for component composition. In *Mathematical Structures in Computer Science*, pages 329–366. University Press, 2002. 2.3.1

[AG97] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, July 1997. (document), 2.3.1, 2.4, 2.3.1, 3.4.2

[AHS93] F. Arbab, I. Herman, and P. Spilling. An overview of manifold and its implementation. *Concurrency: Pract. Exper.*, 5(1):23–70, February 1993. 2.3.1

[And09] Charles André. Syntax and Semantics of the Clock Constraint Specification Language (CCSL). Research Report RR-6925, INRIA, 2009. 3.2.2

[Arb98] Farhad Arbab. What do you mean, coordination? In *Bulletin of the Dutch Association for Theoretical Computer Science (NVTI*, pages 11–22, 1998. 2.3.1

[BBF+06] Jean Bézivin, Salim Bouzitouna, Marcos Didonet Fabro, Marie-Pierre Gervais, Fréderic Jouault, Dimitrios Kolovos, Ivan Kurtev, and Richard F. Paige. *Model Driven Architecture – Foundations and Applications: Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006. Proceedings*, chapter A Canonical Scheme for Model Composition, pages 346–360. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. 6.2

[BCCSV05] Albert Benveniste, Benoît Caillaud, Luca Carloni, and Alberto Sangiovanni-Vincentelli. Tag machines. In *ACM Emsoft*, 2005. 3.2.2

[BCE+06] 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. 2.2.1, 2.2.3

[BFJ+04] L. Barroca, J.L. Fiadeiro, M. Jackson, R. Laney, and B. Nuseibeh. Problem frames: A case for coordination. In *Coordination*. 2004. 2.3.1

[BH08] F. Boulanger and C. Hardebolle. Simulation of Multi-Formalism Models with ModHel'X. In *Proceedings of ICST'08*, pages 318–327. IEEE Comp. Soc., 2008. 1, 2.3, 2.3.2, 2.3.2, 3.2.1, 3.3.1, 3.4.1, 4.5

[BHLM02] 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. 1, 2.3, 2.3.2, 2.3.2, 3.2.1, 3.3.1, 3.4.1, 4.5

[BJ01] P. Bjureus and A. Jantsch. Modeling of mixed control and dataflow systems in mascot. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(5), Oct 2001. 2.3, 2.3.2, 3.2.1, 3.3.1, 3.4.1, 3.4.2

[BJL+15] Barrett Bryant, Jean-Marc Jézéquel, Ralf Lämmel, Marjan Mernik, Martin Schindler, Friedrich Steinmann, Juha-Pekka Tolvanen, Antonio Vallecillo, and Markus Völter. Globalized domain specific language engineering. In Benoit Combemale, Betty H.C. Cheng, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe, editors, *Globalizing Domain-Specific Languages*, volume 9400 of *Lecture Notes in Computer Science*, pages 43–69. Springer International Publishing, 2015. 3.2.1

[BJV04] Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. On the Need for Megamodels. In *Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications,(2004)*, Vancouver, Canada, October 2004. 6.2

[Caz12] Walter Cazzola. Domain-specific languages in few steps - the neverlang approach. In Thomas Gschwind, Flavio De Paoli, Volker Gruhn, and Matthias Book, editors, *Software Composition - 11th International Conference, SC 2012, Prague, Czech Republic, May 31 - June 1, 2012. Proceedings*, volume 7306 of *Lecture Notes in Computer Science*, pages 162–177. Springer, 2012. 2.2.2

[CBC+16] Benoit Combemale, Cédric Brun, Joël Champeau, Xavier Crégut, Julien Deantoni, and Jérome Le Noir. A Tool-Supported Approach for Concurrent Execution of Heterogeneous Models. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, Toulouse, France, 2016. 5.5

[CDB+15] Benoit Combemale, Julien Deantoni, Olivier Barais, Arnaud Blouin, Erwan Bousse, C'edric Brun, Thomas Degueule, and Didier Vojtisek. A Solution to the TTC'15 Model Execution Case Using the GEMOC Studio. In *8th Transformation Tool Contest (workshop TTC 2015)*, 2015. 4.2, 4.4

[CDVL+13] Benoit Combemale, Julien Deantoni, Matias Vara Larsen, Fr'ed'eric Mallet, Olivier Barais, Benoit Baudry, and Robert France. Reifying Concurrency for Executable Metamodeling. In *SLE*, 2013. 3.2.1, 3.2.2, 4.1, 4.2

[CHM+97] Barbara Chapman, Matthew Haines, Piyush Mehrota, Hans Zima, and John Van Rosendale. Opus: A coordination language for multidisciplinary applications. *Sci. Program.*, 1997. 2.3.1

[CSN05] 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. 2.2.2, 2.2.2, 2.2.3, 2.4, 6.1

[CSN07] 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. 2.2.2

[CvdBCR15] Tony Clark, Mark van den Brand, Benoit Combemale, and Bernhard Rumpe. Conceptual model of the globalization for domain-specific languages. In Benoit Combemale, Betty H.C. Cheng, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe, editors, *Globalizing Domain-Specific Languages*, volume 9400 of *Lecture Notes in Computer Science*, pages 7–20. Springer International Publishing, 2015. 3.2.1

[DBC+15] Julien Deantoni, Cédric Brun, Benoit Caillaud, RobertB. France, Gabor Karsai, Oscar Nierstrasz, and Eugene Syriani. Domain globalization: Using languages to support technical and social coordination. In Benoit Combemale, Betty H.C. Cheng, Robert B.

France, Jean-Marc Jézéquel, and Bernhard Rumpe, editors, *Globalizing Domain-Specific Languages*, volume 9400 of *Lecture Notes in Computer Science*, pages 70–87. Springer International Publishing, 2015. 3.2.1

[DCB+15] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. Melange: A meta-language for modular and reusable development of dsls. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, pages 25–36. ACM, 2015. 3.2.1

[DDC+14a] 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. 4.1, 4.3.2

[DDC+14b] 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. 4.3.2

[DDFV08] Marcos Didonet Del Fabro and Patrick Valduriez. Towards the efficient development of model transformations using model weaving and matching transformations. *Software & Systems Modeling*, 8(3):305, 2008. 6.2

[DM12a] Julien Deantoni and Frédéric Mallet. ECL: the Event Constraint Language, an Extension of OCL with Events. Research report, INRIA, 2012. 3.2.1

[DM12b] Julien Deantoni and Frédéric Mallet. TimeSquare: Treat your Models with Logical Time. In *TOOLS*, 2012. 4.3.2

[DNCSSV14] 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 Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran, editors, *Model-Driven Engineering Languages and Systems*, volume 8767 of *Lecture Notes in Computer Science*, pages 353–369. Springer International Publishing, 2014. (document), 2.3, 2.3.2, 2.3.2, 2.5, 3.2.1, 3.3.1, 3.4.1, 3.4.2, 4.5

[Dur98] Emmanuel Durand. *Description et vérification d'architectures d'application temps réel : CLARA et les réseaux de Petri temporels.* PhD thesis, Nantes, ECN, Nantes, 1998. Th. : automatique et informatique appliquées. 2.3.1

[ES06] 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. 2.2.2, 2.2.2, 2.2.3

[ESH97] Jan Ellsberger, Amardeo Sarma, and Dieter Hogrefe. *SDL : formal object-oriented language for communicating systems.* Prentice Hall, Harlow, UK, New York, Paris, 1997. 2.3.2

[Esp09] Esper. Espertech, 2009. 2.3.1

[FBFG08] Franck Fleurey, Benoit Baudry, Robert France, and Sudipto Ghosh. Models in software engineering. chapter A Generic Approach for Automatic Model Composition, pages 7–15. Springer-Verlag, Berlin, Heidelberg, 2008. 2.2.1, 2.2.3

[FJP90] Lars-åke Fredlund, Bengt Jonsson, and Joachim Parrow. An implementation of a translational semantics for an imperative language. In *Proceedings on Theories of Concurrency : Unification and Extension: Unification and Extension*, CONCUR '90, pages 246–262, New York, NY, USA, 1990. Springer-Verlag New York, Inc. 2.1

[fmi] Functional mock-up interface. Technical report. 6.2

[fmi14] *Model-Based Integration Platform for FMI Co-Simulation and Heterogeneous Simulations of Cyber-Physical Systems*, volume Proceedings of the 10th International Modelica Conference, Lund University, Solvegatan 20A, SE-223 62 LUND, SWEDEN, 03/2014 2014. Modelica Association and Linkoping University Electronic Press. 6.2

[GBA+09] Antoon Goderis, Christopher Brooks, Ilkay Altintas, Edward A. Lee, and Carole Goble. Heterogeneous composition of models of computation. *Future Gener. Comput. Syst.*, 25(5):552–560, May 2009. 2.1

[GC92a] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):96–, February 1992. 1, 1, 2.3, 2.3.1

[GC92b] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, February 1992. 2.3.1

[GLL99] 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. (document), 2.6

[GS94] David Garlan and Mary Shaw. An introduction to software architecture. Technical report, Pittsburgh, PA, USA, 1994. 2.3.1, 3.2.2

[Gui92] User's Guide. *MATLAB: High-performance Numeric Computation and Visualization Software*. MathWorks, 1992. 2.3.2

[Gui01] User's Guide. Amba specification (rev2.0) and multi layer ahb specification. Technical report, 2001. 4.3.2

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *COMMUNICATIONS OF THE ACM*, 12(10):576–580, 1969. 2.1

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985. 2.3.1

[JFB08] Cédric Jeanneret, Robert France, and Benoit Baudry. A reference process for model composition. In *Proceedings of the 2008 AOSD Workshop on Aspect-oriented Modeling*, AOM '08, pages 1–6, New York, NY, USA, 2008. ACM. 3.1

[KFJ07] 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. 2.2.1, 2.2.3

[KHJ06] 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. 2.2.1, 2.2.3

[KJ05] 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. 2.2.1

[KK07] Jacques Klein and Jörg Kienzle. Reusable aspect models. In *IN: Proceedings. Of The 11th International Workshop On Aspect Oriented Modeling.*, 2007. 2.2.1, 2.2.3

[KMDS13] Emilien Kofman, Jean-Vivien Millo, and Robert De Simone. Application Architecture Adequacy through an FFT case study. In *JRWRTC2013 - 7th Junior Researcher Workshop on Real-Time Computing*, Sophia Antipolis, France, October 2013. Sebastian Altmeyer. 3.3.2

[Kol09] DimitriosS. Kolovos. Establishing correspondences between models with the epsilon comparison language. In RichardF. Paige, Alan Hartman, and Arend Rensink, editors, *Model Driven Architecture - Foundations and Applications*, volume 5562 of *Lecture Notes in Computer Science*, pages 146–157. Springer Berlin Heidelberg, 2009. 3.3.2

[KPP06] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. Merging models with the epsilon merging language (eml. In *In Proc. ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (Models/UML 2006*, 2006. 2.2.1, 2.2.3

[KRV10] 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. 2.2.2

[LKA+95] 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. 2.3.1, 3.4.2

[LSV98] Edward A Lee and Alberto Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE TCAD*, 1998. 2.3.1, 3.4.1

[mda03] Mda guide version 1.0.1, 2003. 2.1

[MT97] 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. 1, 2.3, 2.3.1, 3.1

[OMG03] OMG. *UML Object Constraint Language (OCL) 2.0*, 2003. 3.2.1

[PA98] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. Technical report, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, The Netherlands, 1998. 3.1

[Plo81] G. D. Plotkin. A structural approach to operational semantics, 1981. 2.1

[RFG+05] 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. 2.2.1

[RSU04] Mara Rhepp, Herbert Stögner, and Andreas Uhl. Comparison of jpeg and jpeg 2000 in low-power confidential image transmission. In *SPC*, 2004. 5.2

[SDK+95] 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. 2.3.1

[VLDCM15] Matias Ezequiel Vara Larsen, Julien Deantoni, Benoit Combemale, and Frédéric Mallet. A Behavioral Coordination Operator Language (BCOoL). In Timothy Lethbridge, Jordi Cabot, and Alexander Egyed, editors, *International Conference on Model Driven Engineering Languages and Systems (MODELS)*, number 18, page 462, Ottawa, Canada, 2015. ACM. to be published in the proceedings of the Models 2015 conference. 5.5

[Win87] Glynn Winskel. Event structures. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*. 1987. 2.3.1, 3.2.2