université de bretagne occidentale

THÈSE / UNIVERSITÉ DE BRETAGNE OCCIDENTALE

sous le sceau de l'Université européenne de Bretagne

pour obtenir le titre de

DOCTEUR DE L'UNIVERSITÉ DE BRETAGNE OCCIDENTALE

Mention : Informatique

École Doctorale Santé, Information, Communication, Mathématique, Matière

présentée par

Papa Issa DIALLO

préparée au

Lab-STICC UMR CNRS 6285

École Nationale Supérieure de Techniques Avancées

ENSTA Bretagne

Un cadre de définition de la sémantique basée MoC des modèles de systèmes dans le contexte de l'intégration d'outils

A Framework for the definition of a System Model MoC-based Semantics in the context of Tool Integration

Thèse soutenue le 16 Mai 2014

devant le jury composé de :

Koen BERTELS
Professeur, Delft University of Technology / *Président du Jury*

Ingo SANDER
Enseignant-chercheur, Royal Institute of Technology (KTH) / *Rapporteur*

Jean-Michel BRUEL
Professeur, Université de Toulouse / *Rapporteur*

Loïc LAGADEC
Professeur, ENSTA Bretagne / *Directeur de Thèse*

Joël CHAMPEAU
Enseignant-chercheur, ENSTA Bretagne / *Encadrant*

Acknowledgements

This thesis "*A Framework for the definition of a System Model MoC-based Semantics in the context of Tool Integration*" took place within the research laboratory on Model-Driven Engineering at ENSTA Bretagne (STIC-IDM), and was funded by the iFEST ARTEMIS project.

There are many people that helped me accomplish this work. This is all for you !

I want to thank from the bottom of my heart my supervisor Mr Joël Champeau and PhD advisor Mr Loïc LAGADEC. For their availability and support on technical and scientific aspects during this thesis, and moreover for their human quality. I have been privileged to work with you, and I will be eternally grateful.

I thank the members of the Jury and the reporters, Mr Koen BERTELS, Mr Ingo SANDER, and Mr Jean-Michel BRUEL for agreeing to assess this work and for having donated their time to attend to the defense.

I thank all the colleagues from the iFEST project. But also, all members of the STIC-IDM laboratory: researchers, engineers, PhD students and students with which I had the pleasure to work in recent years. I do not forget Michèle and Annick.

A special nod to: Ali Koudri, Vincent Leilde, Stephen Creff and Christophe Guychard. I cannot thank you enough for your availability and your supports.

Finally, I thank,

my friends: Papy, Ousman, M. FALL, F.K DIOP, Diouf Seck, Ndokh Ndiaye, GALLEDOU family, CISSE family, J.G. Each of you is a part of who I am.

my family: DIEYE family, BADIANE family, DIALLO family.

My uncle D. DIEYE, more than a father you are a model for me. Believing in someone is priceless, so I will stain to live up to your expectations for the rest of my days.

My mother, for all the sacrifices conceded for the success of my sisters and me. I promise an eternal and unconditional love.

April 24, 2014
Papa Issa DIALLO

Abstract

L'utilisation des systèmes embarqués (EmS) connait un essor conséquent dans plusieurs domaines actuels tels que la téléphonie, l'industrie automobile et l'avionique. Dans ces différents domaines, la croissance des besoins en termes de fonctionnalités a pour conséquence l'augmentation de la taille et de la complexité des systèmes conçus. Dans ce contexte, les chaines de conception des systèmes deviennent de plus en plus complexes et requièrent l'utilisation d'outils provenant de différents domaines d'ingénieries.

L'intégration des paradigmes hétérogènes associés aux outils posent beaucoup de problèmes de fiabilité à l'échange des modèles entre outils d'une même chaine de conception. Par exemple, dans le cadre des EmS, les outils d'ingénierie dirigés par les modèles (IDM) ne sont pas acceptés par les communautés de recherches pour la conception formelle d'EmS qui requièrent des bases solides et formelles de définition des sémantiques d'exécution pour réaliser les activités d'analyses, de validation et de synthèse des systèmes embarqués. En effet, les outils IDM dédiés aux EmS ne sont à ce jour pas encore suffisamment matures concernant l'expression et la prise en compte de la sémantique d'exécution formelle mettant explicitement en avant les modèles de concurrence des systèmes. Par ailleurs, la théorie du calcul est identifiée comme le domaine permettant de décrire de manière formelle les modèles de concurrences qui sont utilisés pour la description de systèmes embarqués.

La motivation de cette thèse est de mettre en oeuvre cette théorie du calcul pour réduire l'écart existant entre différents outils de conception qui possèdent des sémantiques d'exécution de modèles différentes dans une chaine de conception. La thèse propose une méthodologie d'identification et de comparaison des sémantiques d'exécution de modèles qui se base sur la théorie des Modèles de Calcul (MoCs) et leur classification existante, ainsi qu'un langage de capture des sémantiques basées MoC. Ces dernières sont utilisées pour enrichir les modèles et préserver leur sémantique entre les outils d'une chaine de conception. Pour illustrer l'utilisation de l'approche, nous avons défini un flot de conception permettant de connecter trois outils impliqués dans diverses activités du processus "Design & Implementation" (Spécification, Analyse, Exploration de l'espace des choix de Conception). La chaine d'outils présentée adresse la connexion de l'outil UML Modeler (IBM Rhapsody) (pour la spécification et l'analyse), Forsyde (cadre de simulation *multi-MoC* et de synthèse) et Spear (pour l'exploration de l'espace des choix de Conception et l'analyse). La chaine est appliquée sur un modèle de Radar simplifié fourni comme cas d'utilisation dans le cadre du projet iFEST.

Embedded systems (EmS) are increasingly used in various areas such as telephony, automotive and avionics industries. In these different areas, the growth of functionality requirements causes an explosion of the size and complexity of the systems.

In this context, system design flows are becoming more complex and require the use of tools from different engineering domains. The heterogeneous paradigms on which the tools rely on pose as well many reliability problems when it comes to consistent data exchanges between tools. For example, nowadays, the high-level modeling (e.g. Model-Driven Engineering) tools are unaccepted by research communities for the formal design of systems that require solid grounds on the execution semantics to carry out analysis, validation and synthesis of embedded systems activities. Indeed, the Model-Driven Engineering tools dedicated to EmS design are not yet sufficiently mature on aspects involving expression of the formal execution semantics reflecting the concurrency model of system modules. Besides, the theory of computation is identified as the field to describe formally the concurrency models that are used for the description of embedded systems.

Our motivation is to use this theory to reduce the gap between different design tools that have different semantics for executing models in a design flow. We propose a methodology for the identification and comparison of the concurrency model of systems based on the theory of the Models of Computation (MoCs) and their existing classifications; we also propose a language to capture MoC-based semantics which is used to enrich system models and preserve their semantics through a tool chain. To prove the effectiveness of our approach, we defined a design flow connecting three tools that are involved in various activities of the Design & Implementation process (Specification, Analysis, Design Space Exploration). The tool chain highlights the connection of the UML modeling tool (IBM Rhapsody) (for specification and analysis), Forsyde (for *multi-MoC* simulation and synthesis) and Spear (Design Space Exploration and analysis). The chain is applied on a simplified version of a Radar Streaming application provided as use case in the context of the iFEST project.

Contents

List of Figures

List of Tables

1 Introduction

Contents

1.1 Context

With the recent hardware (HW) technological improvements (e.g. FPGA-Based Multi-Processor Systems), the developed systems are qualitatively improved in terms of performance. Indeed, HW technologies offer significant computing power improving the reaction of systems as well as decreasing their processing times. Accordingly, there is more and more embedded applications in various areas such as transportation (e.g. avionics, automotive and railway), multimedia (e.g. video, telephony, and imaging), medical, etc. In fact, Embedded Systems (EmS) are present in most of the domains with important constraints on performance and reactivity. On the automotive domain, the part given to embedded systems is significantly growing both on critical aspects (e.g. airbag, brake, auto-pilot system) or other less critical aspects such as (onboard computer, integrated GPS, etc.).

In these mentioned domains, the development of embedded systems uses design processes that are becoming more and more complex. The Complexity is compounded by the increasing ability of systems to integrate new components.

1.1.1 Current EmS Design Trends

To deal with the complexity of the systems in terms of sizes and features, development processes integrate steps for high-level specification and analysis. The newly integrated design steps advocate modular design with: separation of concerns, abstraction of programming languages, designs on several levels of abstractions, analysis and verification steps at each level of abstraction. The B method is one of the design methodologies promoting: the use of multiple levels of abstraction to describe the systems, and to proceed by successive refinement until a final implementation is generated.

The language abstractions allow defining formalisms that are more easily manipulated by engineers for: specification, reasoning, etc. The separation of concerns enhances modularity[1]. Modularity facilitates the reuse of implemented functional blocks.

Several methodologies have emerged these last decades (e.g. languages and high-level design techniques) implementing these development methodologies [55] [131] [104], [68] [157]. With these new methodologies, the management of the complexity is better ensured since the specification and analysis phases incrementally reduce errors. Additionally, automation mechanisms between the abstraction levels allow reducing human interventions and ensure faster refinements of models between abstraction levels.

For the abstraction of languages and separation of concerns, Model-Driven Engineering (MDE) [177] significantly contributes to the description of Domain Specific Languages (DSL) and Domain Specific Modeling Languages (DSML) allowing to raise the abstraction level of common languages (e.g. C / C + +, VHDL).

Multiple languages for the description of software (SW) and HW were proposed in research or industry (e.g. AADL [50], UML[173], MARTE [148] and SysML [73]), and many code generators were specified for these languages [200] [23]. Modeling languages allow defining application models, architecture models, and HW platforms that can be used for analysis and incremental system specification. These formalisms hide fine grain system properties and put the focus on the relevant aspects of a system. In this context, systems are seen as a set of interacting modules designed using different tools during the progressive design steps (e.g. abstraction-refinement steps).

[1] A module is a sub-part of a system to perform a specific functionality of the system. In this thesis, the term module is used in the same way as subsystem.

Unfortunately, the more improvements of design methodologies are noticed, the more we are witnessing a proliferation of specific tools for the realization of the different modules, their analyses, but also their integration.

As shown in Figure 1.1, during the initial stages, the engineering process helps to specify the required sequences of activities leading to the achievement of the system (e.g. activity 1, activity 2, activity 3). For each activity (or set of activities), combinations of tools help to meet the requirements established by this activity. The diversity of tools in the development process is a real problem for integration environments that usually take into account only a subset of predefined tools.

1.1.2 Integration Framework Requirements

Nowadays, integration environments require solutions responding to several scientific and technical issues such as lifecycle management, design automation or tool's interoperability. Tool interoperability approaches are interested in: how to exchange data (i.e. syntactical interoperability); how to manage the semantics of the exchanged data (i.e. semantics interoperability); how to manage access to the data repositories; how to define the technical interfaces of tools; how to manage and implement services for function calls in modules, etc. Design automation allows having faster design flows with less manual intervention. Several approaches [75][171][169] have managed to provide solutions in this sense in order to describe tool chains.

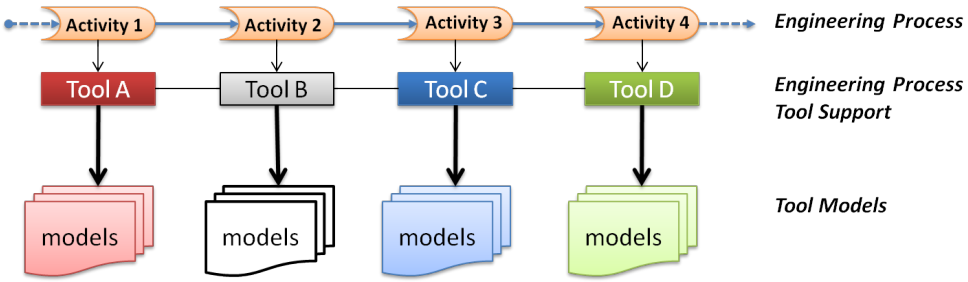

Figure 1.1: An Example of ToolChain Description

For the management of the data exchanges, the Tool Integration Frameworks traditionally define common infrastructures and formalisms for the communication of data e.g. the Interface description language (IDL) [143] of the Common Object Request Broker Architecture (CORBA) [99], the Extensible Markup Language (XML) [22], or the model-based techniques e.g. the Meta-Object Facility (MOF) [142], or the Query/View/Transformation (QVT) [156]. More recently the Open Services for Lifecycle Collaboration (OSLC) [12] infrastructure defines services and common formats ensuring the exchange of data between heterogeneous models.

Several researches on tool integration (e.g. Caesar, iFEST[2]) define frameworks for system design and highlight the need to solve these syntaxes and semantics issues. In particular, the iFEST project of which this thesis is a part and thus receives funds thereof promotes such similar goals, among others.

[2]The iFEST project (industrial Framework for Embedded Systems Tools) aims at specifying and developing an tool integration framework for HW/SW co-design of heterogeneous and multi-core embedded systems. http://www.artemis-ifest.eu/

iFEST is a European Artemis project targeting the development of a Tool Integration Framework for the design of embedded systems. The Framework must allow to integrate or remove, on the fly, tools for design activities such as Requirement and Analysis (R & A), Design and Implementation (D & I), or Verification and Validation (V & V), etc.

The iFEST project uses OSLC infrastructure for the connection between the tools and each tool provides a specification determining how it connects to the Framework and complying with the OSLC standard.

1.2 Problematic and Motivation

As shown in Figure 1.2, SLM tools often offer accurate abstractions and model transformation mechanisms improving the automation process for model exchanges. Unfortunately, such tools fail to accomplish what is done best by traditional EmS formal design tools, i.e. the formal description of the semantics of models (including the execution semantics). Not considering these semantics the earliest possible might end up into inconsistent model.

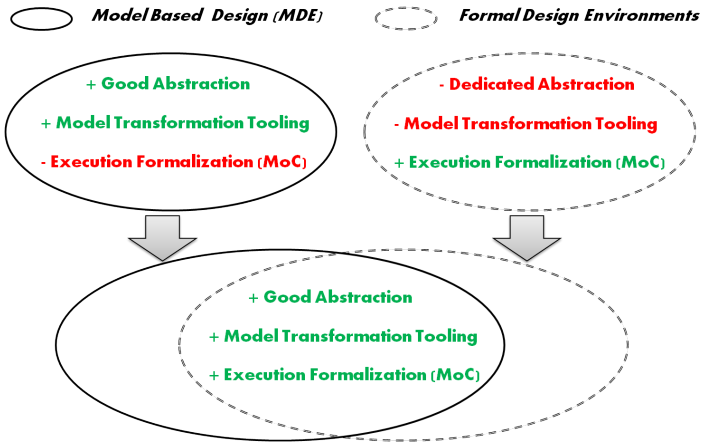

Figure 1.2: Advantages and Disadvantages of both Modeling and Formal Design Approaches

Beside the technical interoperability matters, obviously, interoperability cannot be assumed unless all the semantics issues are handled: the module's engineering domain's semantics, the tool's execution semantics and the inter-module's communication semantics. For EmS design, the semantics are related to engineering domains (e.g. signal processing, control-command), and includes: static properties or execution properties of a domain.

The preservation of the module's semantics between tools is currently not guaranteed. The main reason for this disillusionment comes from the fact that the semantics of models are not sufficiently studied and correlated with the semantics of the tools. In fact, the semantics of the tools is rarely known to developers. If the consistency of the models (at a semantics level) is not guaranteed from one tool to another, then the analysis activities don't maintain the reliability, and correctness of systems.

Indeed, the heterogeneity of the modules causes that their consistent and reliable exchanges between tools is difficult to guarantee. Firstly, the formalisms used to describe heterogeneous models are often different. Secondly, the levels of semantics expressiveness of the formalisms are different. Finally the execution semantics implemented in

the tools may vary from one tool to another. So far, especially in the integration of tool chains, these semantic shortcomings are solved in an ad-hoc manner which is likely to increase the skepticism of the industrials.

The problematic of the semantics of exchanged models in tool chains was already raised in several works for tool integration as in [75]. Indeed, for design flows based on the use of the metamodels for communication, there is no explicitly defined formalism to address these semantic breaches. Consequently, this shortcoming is reflected during the transformation and analysis steps of models belonging to different engineering domains. For instance, for system engineering, we note that nowadays, there is a real need for reducing the semantic gap between model-based development formalisms and the formalisms traditionally used for embedded system development that implement semantics based on formal Models of Computations (MoC) [77] [174].

The computational models are formal semantics describing an abstract representation of the manner in which the elements of a model are evolving. They describe the static and dynamic semantic properties governing the execution of a model. The MoC semantics is then directly implemented by the execution engine or incrementally integrated in the form of behavioral MoC libraries i.e. (protocols, *Scheduler*s, etc).

During the EmS design steps, refined models are semantically derived from the model's parallelism properties as well as their real time properties. These properties are often formally described by MoCs relating to the engineering domains. The definition of a global MoC for a system is equivalent to describing the set of MoCs of each of its modules and the MoCs representing their interconnection.

Our interest for the MoCs is justified by the fact that they will enable to explicitly identify underlying semantics in different engineering domains of the (D & I) activities (e.g. signal processings, control) and the semantics underlying the tool's execution engine.

In order to analyze the consequences of not taking into account semantics (static and dynamic) in the design flows, we propose to analyze the relationships between tools of a design flow from a semantics viewpoint.

1.2.1 Relationships between Tools from a Semantics Viewpoint

These early definitions characterize the tools based on their ability to express MoCs i.e. their MoC awareness.

Definition of *MoC Aware* Tool: An *MoC Aware* tool is a tool that offers implementations or abstractions of MoCs. These MoCs are necessary to complete and analyze models of a system. The MoC Aware tools generally propose effective implementations for a proper interpretation of the specification, except for cases where the specification's model properties are not compatible with the execution semantics of the analysis tools.

Definition of *MoC Unaware* Tool: An *MoC Unaware* tool is a tool that does not natively relate to any forms of MoC representation. For instance, the execution engines of the analysis tools are mainly based on standard principles for execution (Turing machine, etc) without any consideration for other existing MoCs (e.g. related to engineering domains). This type of tool cannot be used for analysis of models with strong semantics, due to the risk of obtaining results that are unreliable.

For the *MoC Aware* analysis tools, the MoCs defined in execution engines are fine grained and strongly coupled to the platform on which they are implemented. They offer finer description granularity and are used to implement the mechanisms of execution and simulation. In this case, their execution engines are able to provide an interpretation of a program or of an executable model.

In the following paragraphs, we identify the coupling scenarios of tools for specification and analysis (*MoC Aware / MoC Unaware*) abstracted in Figure 1.3. The identification of the relationships between tools highlights the issues related to MoC semantics. The arrows define the model exchanges from a source to a target tool and the question marks refer to questionings on the consequences of the lack of semantics definition between tools. The square boxes represent tools, and boxes with rounded corners inside are interconnected architectural models (*AM*).

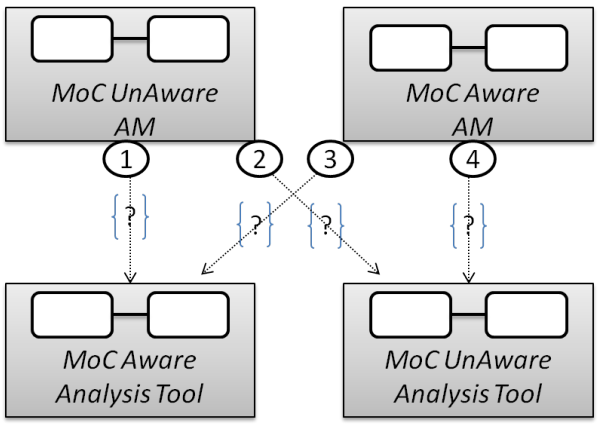

Figure 1.3: Different Scenario couplings Specification and Analysis Tools

Scenario 1 describes the analysis of an *MoC Unaware AM* in an environment where the execution engine implements an MoC-based execution semantics. The lack of MoC properties may result from the lack of concepts for the addition of the missing MoC properties. A consistent analysis cannot be made on this type of models since the analysis tool does not manipulate the correct information related to the MoC properties to provide a correct execution.

Scenario 2 poses the problem of coupling *AM* and analysis tools that are *MoC Unaware*. In this context, the analyses produced are not specific to an engineering domain and in fact, are not relevant at all. The lack of description of the properties and the semantics of the models can be due to a lack of adequate formalisms to express specific properties.

Scenario 3 presents two *MoC Aware* tools for *AM* specification and analysis. This particular case is divided into two cases. In the first case, specification and analysis tools are properly connected and are not incompatible. Most of the EmS tools are based on this scenario by defining *tightly coupled* tools where different semantics are mastered and analyses are accurate because the coupling between specification models and simulation engine is relatively well defined. However, the choice of tools is not open inducing several drawbacks such as the difficulty of outdated tool replacements. In the second case, the problem is related to the

incompatibility of the MoCs between specification and analysis tools. In this case, the analysis tool already has an implementation of an incompatible MoC compared to the ones that are expressed in the specification models.

Scenario 4 is problematic in a sense that the target tool cannot interpret the semantics implemented in the *AM*.

For analysis, different specification tools can use the same runtime engine (e.g. Figure 1.4: *Spec A* and *Spec C* specification tools are connected to *Simu A*); or the specification tools can rely on different execution engines.

In the first case, if there is a formal and consistent semantic translation between elements of *Spec A*→*Spec C*, then *Spec A*→*Simu A* simulation has (a priori) the same results as simulation from *Spec C*→*Simu A* models; if there isn't a consistent semantics translation between the specifications and simulation tools, this reflects the issues expressed in Scenario 2.

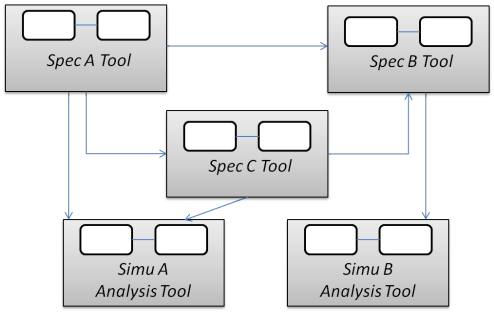

Figure 1.4: Highlighting Execution Semantics issues in Tool Interoperability

In the second case, if there is a formal and correct semantic translation between elements of *Spec A* and elements of *Spec B*, then we can consider the simulation results (i.e. execution traces) of simulation *Spec A*→*Simu A* and *Spec B*→*Simu B* as follows: if *Simu A* and *Simu B* tools have the same execution semantics then *Spec A*→*Simu A* and *Spec B*→*Simu B* simulation results will be (a priori) the same; if the execution semantics of *Simu A* are different from those of *Simu B*, then it is necessary to add an adaptation between (*Spec A* and *Spec B*), or to edit the *Spec B* model so as to obtain the same simulation results (semantic adaptation). Semantic adaptations can also be set between the specification tool *Spec B* and the analysis tool *Simu B*. In this case, we return to Scenario 2 of Figure 1.3. The semantic adaptations are based on the properties to be manipulated and their link of compositionality.

The heterogeneous *AM*s pose the same problems as those presented in the scenarios listed above. Heterogeneous analyses are made by two different methods: using analysis tools implementing heterogeneous semantics, or by the co-simulation tools. In the case of "single" analysis tool, the difficulty focuses on compositionality of modules and their ability to interact in a coherent manner while preserving the overall behavioral semantics. The co-simulation tools also face strong issues: synchronization, timing, interfacing and translation of the artifacts that are exchanged, etc.

Remarkable works have resulted in so-called high-level "single" design and analysis tools to study and analyze the semantics of system models based on MoC. The best-known tools include: Ptolemy [43] developed by UC Berkeley, the ForSyDe framework [174] from KTH, or the HetSC [77] from the University of Cantabria, to cite only these examples. Around the MDE, the Modhel'X [19] and [30] frameworks also allow analysis of the dynamic semantics of heterogeneous models. Despite the efforts, several

disadvantages come from the fact that these tools are hardly acceptable for system engineering at high-level of abstraction. Among other disadvantages, firstly the fact that the levels of abstraction offered by these tools still remain very close to programming languages (C++, Java, etc), or, at best, on the so-called high-level language i.e. SystemC. Thus they are hardly manipulated by systems engineers who are more interested at defining and reasoning on semantic models specified from abstract formalism such as modeling languages (UML, AADL, etc). Moreover, one can legitimately question the consequences when models are reused and exchanged with other tools (exogenous transformation).

1.2.2 Summary

The different scenarios that we have presented indicate the need for semantic consistency between specification and analysis tools for the reliability and the preservation of the semantics of models. To achieve this goal, the following points are important for improving the interoperability of tools during design and analysis of systems:

- *Aim 1*: clarify the semantic differences induced by formalisms and engineering domains as soon as possible to ensure the reliability and consistency of models during all design stages. Thus, there is a need for methodologies to deduce possible adaptations between heterogeneous MoC representations [38];
- *Aim 2*: use the identified semantic differences to capture adaptation models preserving the semantics of the exchanged models between design tools. The captured models are defined using a formalism to express MoCs, in our case based on metamodels. The added value is to ensure that the models produced in each tool can be reused coherently through an integrated tool chain without losing semantics between tools.

1.3 Contributions

Our contribution in this thesis aims at improving the intermediate steps of exchanges between tools where the semantics are important. We address the design stages ranging from high-level specification, analysis to refinement onto finer descriptions such as SystemC that provides facilities for heterogeneous simulation and code generators for HW synthesis.

We believe that semantics preservation can be achieved based on three criteria that are developed in the sections below.

The study of engineering domain and tool semantic properties helps to unambiguously reason on the system behaviors, and to be able to separate the semantics that belong to applications from the tool semantics; while being able to relate both semantics.

1.3.1 Making MoC explicit for behaviors and semantics preservation between models

For each engineering domain and tool, we can find a set of formally defined properties and execution rules. Such formal properties and rules are given by theory a of computation (MoC) [96] [113]. A MoC is generally defined by:

- a static representation: the set of properties (parameters, data type) that is specific to a domain. These properties make explicit the information relevant to determine the execution;
- a dynamic representation: the description of how the behavior dynamically evolves over time, and its impact in terms of Input/output (I/O);

As an illustration, let us take the example of image processing (which can be extended to multimedia processing's in general). An image processing application is based on the definition of several modules (Filters, Stream Parser, Dequantizers, etc.) that work in parallel or in sequential to read and modify an input image and write an output one.

The sequential or parallel execution depends on several parameters including: the available platform (single processor, multiprocessor), and the type of dependency relationships (partial or total order) between the various components (e.g. precedence, succession, independence, etc). Those relations influence the execution and scheduling policies; each application module can also be completed by properties that represent e.g. the maximum size of data read or written.

These parameters are similar for many multimedia applications that define static properties (e.g. size of data in modules I/O), and properties which are induced by the dependency relationships between the different modules of an application. For example, multimedia applications use the SDF MoC [113] for early analysis. This MoC is later refined through several steps to reach the final implementation.

Studying explicitly the MoCs related to tools and engineering domains is an additional step to moving towards a better understanding of the semantics interoperability problems. The next concern raises the importance of the semantics correlation to determine their compositionality. Logically, if the semantics are not compliant, there is a significant risk of producing inconsistent models.

1.3.2 Compositionality between Heterogeneous Semantics based on MoC Classification

To meet the challenges posed by *Aim 1*, we define a methodology addressing the identification of semantics: either to determine and adapt semantics between models; or to index connections and exchanges between tools that are not relevant. The theoretical results of the semantics comparison may also play a role in the choice of tools to integrate into a design chain. Indeed, there is no logic into connecting tools with incompatible semantics and this information can be obtained before any physical integration. At the modeling level, the semantics of the exchanged elements are studied with respect to the static semantics of the elements and the behavioral semantics (or dynamic semantics) both are MoC-based. According to the execution rules imposed on models, we consider the semantic adjustments to be added to preserve the semantics of the executable models. The semantic adaptations are described in the form of reusable models whenever they allow a response to a specific need for adaptation.

Studying the compositionality between the various MoCs is mandatory to provide consistent semantics interoperability for environments implementing different MoC. In this context, it is important to determine to which extent the MoCs share common properties, and to identify there differences. The properties ensure the preservation of (static, dynamic) semantics on several levels of abstraction and between different tools.

Once the identification of semantics (MoC-based) and the compositionality issues are resolved, then we can focus on the capture of semantics to enrich and adapt models with distinct semantics.

The compliance relationships between MoC have been addressed in [176] [62] [88]. However, it was in the context of heterogeneous behavior simulation within single tools. For tool integration, there is still a lack of contributions addressing the explicit study of semantics compliance between tools and models. In this thesis we provide such methodology.

1.3.3 Defining a formalism to express MoC-Based Execution Control

While the selection of formalisms for capturing static properties is relatively simple and can be done with metamodels for example; the representation of dynamic properties to adapt model's behaviors is much more difficult to provide.

Indeed, such formalism must be flexible and abstract enough to be compliant with different MoC semantics. For instance, at some levels of abstraction, if the analysis tool's execution semantics is DE, the intermediate adaptation models will manipulate control-events to connect with the DE execution engines and also represent dynamic execution rules of the MoC that conforms to an engineering domain.

In the literature, and more particularly in MDE and tool integration, approaches are lacking to provide: abstract and flexible representation of MoC-based dynamic semantics; mechanisms for adaptation between different environments and tools.

To meet the challenges posed by *Aim 2*, we redefine the Cometa language the preliminary work of which was directed by [106]. This first experimentation gave the opportunity to express a set of MoC semantics from a dedicated DSML.

The Cometa language has been extended to a more flexible DSML, which allows new previously unaddressed semantics to be taken into account, and the language's semantics is now formalized. The new DSML allows, in the context of model-based tool integration, the capture of reusable semantics between interconnected (D & I) tools with different semantics.

The MoCs will be used at every design stage between the models and tools to express an abstract representation of the manner in which the tool models must be executed in the source and target environments.

The iFEST project is the ideal framework to design, to experiment and to validate a solution aiming to respond to the two aims previously stated. The targeted objectives (i.e. *Aim 1* and *Aim 2*) make sense for (D & I) and (V & V) activities of iFEST.

1.4 Document Layout

Our concerns are focused on three main issues that are: the need to make the semantic differences explicit between the models from various engineering domains, the need to express and formalize solutions to analyze these semantic differences, the need to demonstrate the ability to integrate such semantics into model-based tool chains resulting in the preservation of semantics between different heterogeneous models and tools. The following chapters describe the work that was carried out to solve the shortcomings raised.

Chapter 2 presents a state of the art on embedded systems and their development. In this part, we give descriptions of the different recommendations for the design of systems, and we also present some Frameworks for embedded system design.

Chapter 3 also features a state of the art on MDE techniques and their basis: this part highlights the basic grounds defined in this research area as well as the useful elements for the description of the semantics of models in an MDE context. This chapter also gives state-of-the-art on MoC when it comes to embedded systems. Several examples of known MoC families in the literature are presented. Finally, we give a description of a few Frameworks using the MoC approaches, some of which were stated earlier.

Chapter 4 presents the new Cometa DSML, its formalization and operational rules allowing executability of models described from Cometa. The DSML will be later used for the description of adaptation between tools with different MoCs.

Chapter 5, presents a typical use of our approach for the prototyping of a design flow emphasizing the design of a Radar Streaming Application. The experimentation addresses several steps of the (D & I) activities from the choice of the tools in the development process, until the final design with use of semantic adjustments between the various tools. Finally, Chapter 6 presents the conclusion of this work and the perspectives.

2

Real Time Embedded Systems Design

Contents

2.1 Introduction

Embedded systems are increasingly important in the design of various products. In fact, they currently represent a considerable proportion of various sectors such as the automotive industry, avionics, telephony, medicine, etc.

In this chapter, we look at the evolution of embedded systems over several decades. This work is important because it reflects the enormous progress that has been made in this domain. Through this progress, engineers can integrate entire systems on electronic chips. Such technological advances have paved the way for new achievements, which have led to the increase in system functionality and consequently to the heterogeneity of systems. Indeed, the new system designs incorporate various functionalities such as wireless devices, GPS, multimedia (image, video processing), etc. Each of these technologies is implemented using dedicated tooling and paradigms.

The heterogeneous and complex systems necessarily have an impact upon the development processes used for their development. We also address the evolution of development techniques over the last few years. Particularly, we address new design methodologies that are intended to improve the development phases and that propose the best practices and recommendations for embedded system development.

In the current development context, designers are facing a series of issues (e.g. heterogeneity, size) which have a strong impact upon the development time, productivity and reliability of the final products. Because of the increasing heterogeneity of the systems, the number of tools required in the design flows are also increased. Consequently, these problems are mainly due to: a lack of maturity of the solutions developed for connecting tools (integration); lack of trusted automated or semi-automated development processes; or even the complete failure of solution providers to enable semantic interoperability i.e. guarantee reliability and consistency of models that are exchanged between the various tools during the development phases.

In this thesis we tackle this problem of semantic interoperability of models produced by the design tools in the domain of embedded systems (EmS).

2.2 Historical study and basic concepts for Embedded Systems

Throughout history, one can find an untold number of transdisciplinary systems involved in revolutionizing our daily lives. The very idea of designing systems or developing new mechanisms to solve problems is implicitly related to the notion of *system*. Since the 1940s, research contributions in the domain of system theory and Cybernetics have given rise to important formalizations of the concept of system. In particular [7][24] in the domain of Cybernetics, for the general theory of systems [15] and the complexity theory [29]. Based on these, my vision of a system is as follows:

Definition: ***A system*** *is designed for a specific purpose; the achievement of this purpose depends on the proper functioning of all its parts (sub-system) working individually or in interdependent manner. Each component of the system may not have knowledge of the overall system objective, while being undoubtedly a guarantee of its consistent running.*

In the literature, there are several classifications of systems depending on their objectives and their specificities e.g. transformational systems [71], and reactive systems

[71] [205]. In this thesis, we pay particular attention to reactive systems that are very close to embedded systems. For more information on transformational systems, [71] provides substantial material on the subject.

Reactive systems are described as being in perpetual interaction with their environment and respondent to the stimuli of the latter. In [13] and [10], reactive systems are divided into two sub-categories: conversational systems in which no constraint exists on the response time of the system to an external request. These systems are deterministic to the extent that they may choose to send or not in response to a stimulus and decide the moment of response (e.g. databases, network systems). On the other hand, purely reactive systems are forced to respond instantaneously to their environment.

Embedded systems are part of the category of reactive systems since they are designed to produce an instantaneous result with their environment.

2.2.1 Description of the concept of Embedded System

The family of embedded systems is a variant of reactive systems given that they are governed by the same computing rules. However, a peculiarity of embedded systems lies in the fact that they jointly incorporate software (SW) and hardware (HW) parts, which gives them a heterogeneous appearance.

Quote: *"An embedded system is an engineering artifact involving computation that is subject to physical constraints. The physical constraints arise through the two ways that computational processes interact with the physical world: reaction to a physical environment and execution on a physical platform"* [76].

From a platform point of view, this definition highlights the heterogeneous nature of the EmS defined as interacting HW/SW parts. The heterogeneity is accentuated by the diversity of the types of integrated SW and HW.

The software programs are mainly supported by Real Time Operating Systems (RTOS), Middleware, Internet Modules, Graphical User Interfaces (GUI) [192], and so on; the platforms are mainly composed of Processors, Memories, Mass storages, Sensors, Actuators, etc. However, even being composed of different modules, the objective of an embedded system is unique over time. They thus accomplish a repetitive task whenever they are accessed by their environment.

Quote: *"An Embedded System (EmS) is an electronic system with dedicated functionality built into its HW and SW. The HW is microprocessor-based, and uses some memory to keep the SW and data. It provides an interface to the world or system it is part of. In most cases, it is a part of a larger heterogeneous system where it plays a computing, measuring, controlling and monitoring role. "* [192]

Nowadays, embedded systems are present in several sectors e.g. avionics/ the automotive industry, mobile telephony, networks, etc. (Figure 2.1). The design of systems in each domain is based on design requirements of the domain (e.g. standards, consumption and space constraints). Each requirement is a realization constraint that may even have an impact on the choice of the HW for its design. Constraints are various i.e. functional, non-functional (HW constraints, performance, power consumption), etc.

Based on the business domains, sub-classes of EmS are identified such as Real Time EmS (RTES) and critical EmS [208].

An RTES [184] [119] imposes functioning or reaction constraints on a fixed time basis. Such a system is doubly constrained by the behavior it must achieve and the

Figure 2.1: Example of Embedded Systems Integration

time of realization of the behavior. For example, the braking system of a car integrates EmS with real time constraints e.g. a braking from environmental action should provoke a reaction taking into account appropriate braking delays (required by the system). A braking action is performed from the environment; accordingly the system must be designed to respond to such a request in real time.

With these strong critical constraints, the description of the design flow of such a system must not neglect any aspect having an impact on the reaction time of the final system to be implemented. Therefore, the modules produced in the chain must be precise as regards to the functionality they perform and their interaction must be handled throughout the design process.

For the critical systems, any misinterpretation of constraints can lead to tragic consequences [187] (physical damage or serious economic damage). For example the railway signal systems are critical systems.

2.2.2 Evolution of Embedded System

Between the 1960s and the 1990s, the use of embedded systems experienced unprecedented growth. Indeed, we saw the emergence of new electronic components to facilitate the design of systems and master their complexity. For example, in the 1980's, DSP processors (first single chip 1980, NEC μ PD7720 and AT & T DSP1) and FPGAS were designed with component integration capabilities such as microprograms, specialized and reconfigurable circuits. More broadly, to meet space and portability constraints (while maintaining the capacity of the EmS); we witnessed the emergence of System-on-Chips (SoCs) [192]. SoCs allow the integration of whole system in a single chip of reduced size. These chips can integrate (DSP, FPGA, Converters Analog/Digital and Digital/Analog, control modules (micro-controllers), etc) otherwise known as Intellectual Property (IP). Figure 2.2 proposes a description of the abstraction of a system on chip SoC based on [89]. As a result, several features are now integrated and designed in the form of a package or IP. This modular component interconnection approach creates interfacing requirements [192]. For interfacing to components, during the last decades, several efforts to standardize HW parts have led to the realization of communication modules such as (VMEBus [122], PCIbus [28], Avalon Bus [4] of Altera) for the board-based systems. At the time, this enabled more complex systems to be developed. Such evolution was predicted by Moore's law in 1965 on the possibility of doubling the number of transistors in chips every 18 months in the years to come. The complexity of the system design depends strongly on the diversity and size of the SW and HW. The

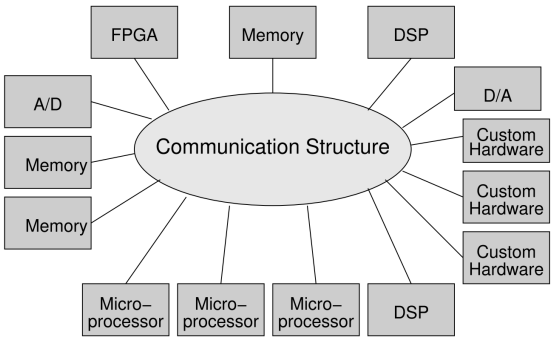

Figure 2.2: Some elements integrated within a SoC [89]

larger and more heterogeneous the system, the more the technologies required for their design are sophisticated and complex (DSPs, FPGAs, Shared Memory, etc). Other factors such as space restrictions and Energy consumed also have an impact on the design choices.

The integration of these different technologies is also an important design issue since many formalisms and engineering domains interact under the imposed constraints. In this context, several design challenges are posed. We examine some of them in the next sections.

Quote: *"Modern cell phones may have four to eight processors, including one or more RISCs for user interfaces, protocol stack processing and other control functions: a DSP for voice encoding and decoding and radio interface; an audio processor for music playback; a picture processor for camera functions; and even a video processor for new video-on-phone capabilities."*[126].

Nowadays, most of the "Computing Systems" are EmS. In 2012, the EmS market represented a significant investment of about $ 56 billion (25 % annual growth). In this context, the competitiveness of the EmS design firms is measured on their ability to reduce costs and design time [54], while ensuring the production of more efficient systems. To achieve these objectives, design methodologies have an important role to play. Unfortunately, with the ever increasing level of performance and number of features offered, the methodologies, languages and development processes are less suitable for the design of systems. In the next section we are interested in EmS design techniques as well as future challenges to improve design techniques.

2.3 Embedded System Design and Implementation

With the old development methods HW and SW partitioning was very early in the design process. Each engineer used to work in isolation and independently, and then integrated the various modules in late stages. The use of this same process with the current complex systems is problematic because the addition of new features involves an increasing number of engineering domains. As a consequence, the systems integration phases are tedious and error prone. Searching for errors at this level is very time-consuming.

Quote: *"Designing embedded systems requires addressing concurrently different engineering domains, e.g. mechanics, sensors, actuators, analog/digital electronic*

hardware, and software." [51]

2.3.1 Design Methodologies and Design Flows

To compensate for these different breaches, the development processes were improved to allow, the earliest possible, consideration of the issues related to the heterogeneity of the systems as well as their verification and validation.

2.3.1.1 Improved Methodologies for improved Design Flows

The major trends for harmonized system design advocate raising the abstraction of system models to provide a common reasoning framework for the different stakeholders of the development process. This reasoning framework facilitates decision-making after consultation between engineers and the partitioning comes at very late design stages.

In fact, several levels of abstraction are defined, allowing successive model refinement steps to be included up to the implementation. This process also allows the design of systems that are correct-by-construction with the activities of analysis and validation of models at each level (Figure 2.3).

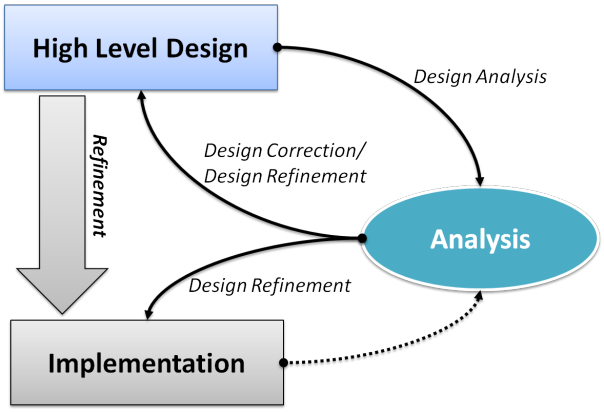

Figure 2.3: An simple Abstraction-Refinement Process

The refinement of the behavior occurs horizontally, vertically and across different tools. In both cases, the formalisms to express behavior play an important role. In fact, the high-level behaviors are increasingly expressed by high-level languages such as activity diagrams and state machines. However, successive refinements of behavioral models tend to make a vertical refinement using action languages, or very low level programming languages.

The Horizontal heterogeneity (decomposition) raises the problem of decomposition of behaviors and interactions. The models that are decomposed must remain consistent before and after their refinement. The structure modification implies changes on the interaction mechanisms. For this reason, the newly refined models often require an adaptation of the interaction mechanisms to preserve the behavioral logic.

Vertical refinement raises the problem of the hierarchical heterogeneity. In other words, considering one level of abstraction, all the system modules obey the behavioral logic induced by a given MoC; after hierarchical refinement of each module, the different parts might, in turn, obey different MoC rules. In such cases, the adaptation between the hierarchical levels that require different MoC semantics should be provided.

The functional behaviors of the different modules are the basis for the interactions. The Cometa approach does not aim to describe such behaviors. They are in fact

incorporated as "legacy codes" that can read (or receive) , transform, and write (or send) data. The functional blocks must not introduce side effects on the functioning. In addition, they do not handle global variables of the system. In the various system configurations, the data are to be stored in storage entities such as FIFO, LIFO, or transported by communication mechanisms such as Message Passing.

Besides, separation of concerns is encouraged because the design of applications still suffers from a lack of clear separation of aspects related to the description of the communication between components that are often built inside the description of application behavior which does not facilitate the understanding of the mechanisms of communication between modules as well as the reuse of application behaviors. Several solutions have been developed over the past decades to encourage these new design methodologies. For instance, the ESL community has made considerable efforts in terms of methodologies and tools [36]: several extensions of SystemC aim at raising the level of abstraction of the primitives e.g. TLM (Transaction-Level Modeling) [61], and also to allow heterogeneous modeling of the modules in accordance with different semantics of execution e.g. HetSC [80] or SystemC-AMS (Analog/Mixed-Signal) [196]. The improvement of development processes will significantly reduce the efforts required for verification of highly heterogeneous systems (that currently represents 70 % of the design efforts).

"Boeing 777 had $ 4 billion development costs, 40 to 50 % of integration and validation efforts."

2.3.1.1.a Raising the level of Abstraction

Higher levels of abstraction are recommended in all communities developing complex systems. They are parts of the techniques to abstract away the bulky properties. Thus, designers have the opportunity not only to focus on specific parts of the system, but also to opt for an incremental approach for design, where models are refined on several levels of abstraction. With abstraction, designers reason, analyze and validate a particular aspect of a system.

Moreover, development on several levels of abstraction greatly improves the "debugging" phases because it allows errors to be located more easily.

Unfortunately, one of the main issues is to define and fix the aspects related to the interaction or exchange of data between different modules at each abstraction level, which is our motivation in this thesis. To this end, the management of concurrency and communication on different levels of abstraction also allows the development of the mechanisms of execution control and synchronization of applications before their synthesis on a highly parallel architecture. In addition, the separation of the computation and communication allows a clearer description of the system modules and their interactions.

2.3.1.1.b Separation of Concerns

The separation of concerns is complementary with the abstraction/refinement approach.

Separation of concerns is a design principle that promotes the separation of the system into several autonomous entities each addressing a particular problem. The entities (or modules) are given to qualified engineers that implement solutions for each

problem. Besides, the integration phase of entities is a mandatory step because it can affect the overall system behavior.

In this approach, at each level of abstraction, designers use virtual models representing architectures and virtual structures for analysis of the models. While abstracting the description of architectures and behaviors, users also separate the communication rules.

2.3.1.1.c IP Reuse

This solution is based on the reuse of IP blocks (Intellectual Property) [56]. A core IP is defined as an electronic component implemented and offering guarantees for a given functionality.

IP-Blocks are used in very late stages of design because they correspond to final implementations to integrate in the HW (or interacting with the other HW components). In high-level design steps, the IP-Blocks under consideration can be summarized in terms of their functional properties and their interfacing characteristics with external modules. In a development process, the use of these components helps prevent their reimplementation and therefore offers a gain in precious time for design and verification steps, thus improving productivity.

2.3.1.1.d Analysis, Verification and Validation

These activities allow the detection of design errors for the SW and HW. For the SW part, it is mainly behavioral analysis on computation, synchronization mechanism and scheduling; on the HW part it comes to performance and energy consumption testing. Analysis activities should also oppose the HW/SW co-design with their global properties (behavioral, performance, space, etc.) at each level of abstraction. For example, sequential SW design and programming techniques currently cause many problems with the new parallel platforms because they do not integrate consistent concurrency [65]. Indeed, the various features of the systems were previously developed with a "sequential reasoning" i.e. succession of statements to make a computation. Nowadays, designers must also take into account the concurrency of the different modules and the properties of host platforms as soon as possible. These new constraints must be part of the process of analyzing the behavior of the system using platform virtualizations.

The above ideas were the basis of numerous initiatives of research, publications, standardization and frameworks [8] [60]. As a result, there are a large number of tools on the market that are used during the design phases [90] [199]. In this context, tools and framework designers are increasingly constrained to ensure compliance between tools.

2.3.1.2 Well-Known Foundations for Embedded System Design

Due to the engineering domain differences, SW and HW parts have long been designed separately (early partitioning between SW and HW). On the one hand, to design SW, languages such as C [100], Ada [111], or recently C++ [188] are used for behavioral modules of the systems. While the HW parts are implemented using languages such as VHDL [160] or Verilog [190]. The above languages very quickly became difficult to use for large systems. Therefore, we are witnessing the emergence of the first languages providing more abstraction in the way to describe the behavior of the systems, but above all, which allow the SW and HW to be described jointly. Most of these languages are

based on paradigms of design with very strong semantics. We can refer to the languages based on the purely synchronous approach such as LUSTRE (declarative) [70] and ESTEREL [14] and SIGNAL [11]. The LUSTRE language addresses the continuous real time treatment of data-flow synchronous systems. On the other hand, the arrival of the SystemC language has enabled the standard for the design of EmS. System models are described as processes interconnected via ports and channels that implement the properties of the system. The approaches around SystemC allow system analysis and Design Space Exploration [1]. The language is implemented on top of the standard C++ libraries and offers primitives for the description of properties related to data types specific to EmS, the types of communication supports, and the description of structures composed of interconnecting concurrent entities. In addition, the framework natively implements a simulation engine for the analysis of the system models. The execution engine has wrappers allowing the description of several execution semantics such as Kahn Process Network (KPN) [96], Synchronous Data Flow (SDF) [113] semantics. The SystemC language has currently become the reference for abstract descriptions of EmS and defines several levels of abstract representation of models and has more abstractions such as SystemC TLM or SystemC-AMS.

However, it still has many shortcomings for: heterogeneous semantic expression and accurate abstraction levels to describe the system models. In fact, the language remains very close to the implementation. The above languages are popular in EmS design communities, especially as they are integrated into several tools for modeling and analyzing heterogeneous systems. In particular, SystemC appears as the target of many high-level modeling frameworks because it provides adequate support to move towards code generation for the implementation. Somehow, this choice is motivated by the fact that formalization, standardization and tooling efforts were made around the language. To have a good vision of how embedded systems are designed, it is also important to clearly distinguish between the different constituting activities.

The specification phases allow system engineers to come to an agreement and analyze what the system must do. In this step, engineers focus on a very abstract description of the system by building representative high-level models of the structure of the system, its behavior and possibly its I/ O and expected performances. The aim is to agree on a first draft of the system and its behavior taking into account functional and non-functional properties. In [159], the authors describe such models as analysis models that are made using different notations and supports depending on the objectives. The description formalisms range from mathematical representations (on paper) to graphical descriptions of the structure and behavior analyzed by tools. There are several tools to manipulate these formalisms e.g. the Unified Modeling Language (UML) [140], MATLAB Simulink tools [199]. The specification phases are followed by so-called Electronic System Level (ESL) [127] Design activities.

2.3.1.2.a Electronic System Level

Previously known as System Level Design (SLD) [127], it brings together several methodologies and tools aimed at the synthesis of an implementation from abstract models of an electronic system. Mainly adopted by designers of SoCs, it aims to optimize and improve the performance of the systems on chips.

[1] Design Space Exploration activities always participate in the choice of architectures optimization by reducing the number of configuration of architectures possible for a type system based on the properties of virtual platforms.

Quote: *"Any level of abstraction above RTL, can be considered as part of the ESL"* [175]

The ESL design adopts the Y-chart [103] principles and promotes the separation of concern. Thus, the experts in EmS design tend to describe functional and architectural systems separately: functional design refers to the design of the applications that make up the system, and architectural design is separated into logical architecture design and physical architecture design. Logical architectural design refers to all the activities of architecture choices and deployment of the applications onto the logical architecture for analysis and refinement, while physical architecture design refers to activities of building the final physical platform. The functional description can be accomplished in several stages (depending on the stage of design). The functional descriptions are interested in the description of the structure, behavior and mechanisms for communication of the subsystems of the system. These activities enable a simple way (FSM, algorithms) to represent abstract and formal application model behaviors the specification of their execution properties. The abstract representations are based on formal models of computation (MoC) such as KPN, CSP [82]). These MoCs are defined in the section 3.4.2.

2.3.1.2.b Platform-Based Design

During architecture design phases, engineers choose different platforms to assemble in order to boost the performance and increase system efficiency once they are implemented. The HW choices take into account physical and spatial constraints (resources, communication, energy, size, etc). The analysis of application mapping onto HW architectures has an impact on the effectiveness of the final realization because it helps to explore the best architecture assembly to optimize the performance of the systems. Platform-Based Design (PBD) is a *Meet-In-The Middle* approach (Figure 2.4) that associates the efforts of functional design top-down with the efforts of abstraction for architectures "bottom-up" [201][101]. Each architecture abstraction hides details and

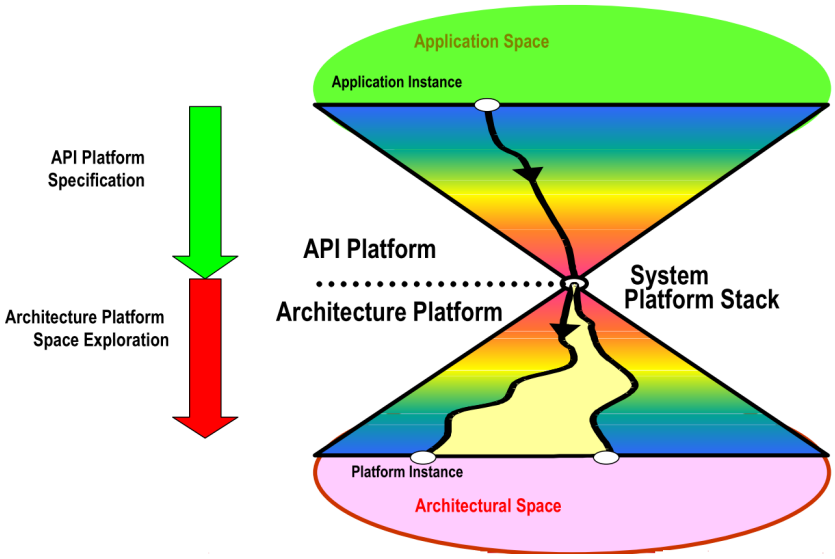

Figure 2.4: Platform-based Design Principles

properties of the lower architecture abstraction it depends on. Nevertheless, the ab-

stractions contain sufficient details to be easily refined from one upper architecture model to a lower model. A platform in the PBD approach is an abstraction layer of architecture that allows a number of properties of architecture to be displayed, and also offers opportunities to refine this architecture to provide detailed properties. For a given level of abstraction, the platform is defined by a library of components and rules (for component assembly). The concept of Platform Stack [201] is defined to represent all of the methods and tools to refine an abstract platform to its next refined platform. This means, all the relevant techniques for the refinement from one abstraction to another. During the analysis activities, the platform Application Program Interface (API) is used for the recognition of the interfaces between the SW and HW. Today, the difficulty of using the PBD techniques resides in a lack of design flows associated with this approach which weakens the proposed solutions and slow its evolution. Indeed, designers remain quite skeptical about changing their development processes to these approaches. Besides, the platforms are generally poorly characterized because their definition is hard to achieve for a given abstraction level.

2.3.1.3 Examples of Frameworks for RTES

It would be difficult to draw up an inventory of all the tools and languages dedicated to the design of EmS and based on the previous recommendations in section 2.3.1.1. The following references are also environments for EmS design [199] [18] [8] [158] [105], and yet the list is far from exhaustive. However, we will present two of the well-known environments for EmS design: one widely used in industry i.e. Simulink and an academic framework [45]. This choice allows us to briefly illustrate the solutions that exist in these communities to solve the problem of EmS design following the recommendations.

Simulink [97] is a tool from the Mathworks suite of tools for the specification, analysis and simulation of dynamic systems such as signal processing (control-flow, stateflow and data-flow). This framework is used to describe models as the interconnection of linear or non-linear components graphically represented by a “Block Diagram”. In a “Block Diagram”, the Hierarchical components are connected via ports and lines (branches). Thus, specification activities allow the graphic representation of complex models that can be simulated using the Simulink execution engine. The latter is able to interpret models according to the several execution semantics which it implements. Simulink has mainly 3 categories of execution semantics corresponding to the implementation of models of computation: Continuous-Time, Discrete-Time and Event-Triggered. For each of these types of semantics, Simulink not only defines specific types of Blocks (Block CT, Block DT, etc.) but also allows their internal combination (heterogeneous). As long as the semantics of models remain in these 3 categories, the runtime engine is able to interpret their combination and then to provide results of simulations based on signal (data) input of the system.

Regarding our approach, the major shortcoming of Simulink is related to its lack of openness to the description of the model semantic, in case of exchange between tools. There are no guarantees that the Simulink model will be correctly interpreted outside its own framework by another analysis tool. Similarly, the lack of explicit and formal description of the semantics of Simulink models causes difficulties in proposing external solutions (from external tools) to interpret correctly the output models of Simulink.

SESAME [45] is another approach based on the above design methodologies. Indeed, SESAME is a framework for modeling and simulation (for multimedia embedded systems). It is primarily aimed at Pruning on the designs of system architecture as

well as Design Space Exploration (DSE). The methods applied in the framework for a high-level design separate the description of behaviors (communicating processes and features KPN); the description of system architecture (constraints and topology). The high-level models (including Khan communicating processes) are associated with a code generator *KPNGen* for the production of C/C++ code for the simulation of the application. Apart from the separation of concerns, the framework defines an engine allowing the co-simulation of model behaviors and architecture models with an execution semantics based on execution trace handling (trace-driven) as shown in Figure 2.5.

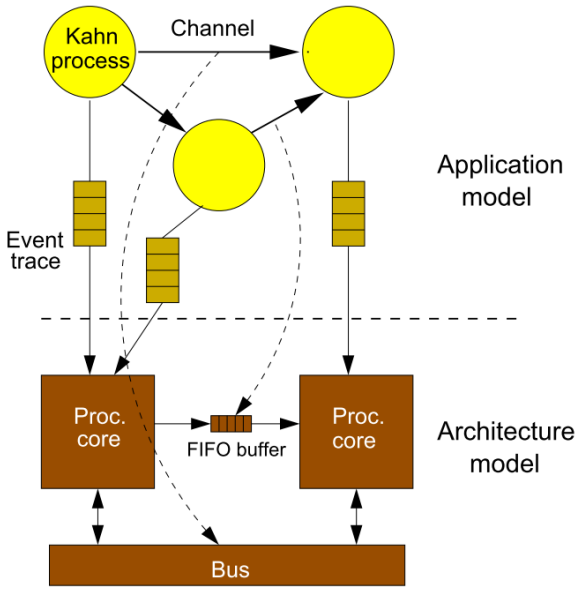

Figure 2.5: Allocation model from SESAME

Nowadays, significant efforts have been made to solve the problems of the ESL and offer innovative solutions with best abstractions [101] [90]. However, many of the solutions suffer from a lack of maturity of the semantic consistency of models and the preservation of the semantics of models between tools as stated by [175] [8]. Indeed, the design by refinement principle will be difficult to apply on several levels of abstraction when several independent design tools are used and they do not provide guarantees on the semantics and behavior preservation during the stages of refinement. Therefore, the models are not necessarily correct-by-construction. This observation is made by [207] stating that ESL methodologies such as the High-level Synthesis (HLS) while offering added value for the synthesis of implementation, the rapid prototyping, or the analysis of performance, they remain quite immature in so far as synthesizers (code generators) still suffer from the existing gap between: the so-called high-level languages promoting (separation of concerns and design by refinement) and the lower level implementations (i.e. Register Transfer Level (RTL)) associated with different tools using fine grain implementations. Moreover, most of the tools in this context are academic tools [45] [8][39] and only few of them are accepted by industry.

To cope with the complexity of the systems on the one hand, and the semantic gap between design tools on the other, it is necessary to use several intermediate tools during the development process to perform high-level analysis, refinement and incremental code generation. In other words, these tools must have compilers and interpreters at

each level of abstraction.

2.3.2 From Design Automation to building Tool Chains for RTES

In this section, we describe the relationship between the design automation and the tool chain descriptions while trying to make explicit the constraints related to this relationship.

With the multiplication of the number of engineering domains and the number of steps involved during the design phases, there is also an increase in the number of tools. In this context, there are many challenges that are posed, especially for the automation of the design phases. However, automation is strongly constrained by problems of tool interoperability.

Automation aims to enable easier data exchanges and transformations between tools without loss of information. It must allow automation of design activities while taking into account the translations of data between tools. These procedures usually contain phases of data handling which can cause loss of information or poor semantics translation leading to misinterpretations of the data. The interoperability targets the ability of specific frameworks to ensure a solid connection/integration of tools using reliable definitions of infrastructures (interfaces, communication protocols, data format, etc.). The problems of interoperability between tools exist in different communities including embedded systems. In the context of embedded systems, interoperability is a major concern regarding the heterogeneity of systems and their strong underlying semantics.

For EmS, design automation is often related to the analysis or RTL code synthesis from high-level models (e.g. SystemC, TLM, etc). The generated code artifacts are: tasks to run on processors; communication protocols connecting modules [207], etc. For this reason, high-level Synthesis (HLS) activities are considered as part of the process of automating the Design because their goal is to provide the tools and methodologies required to generate / synthesize (implementation) code for the HW. The early HLS tools failed to deliver satisfactory results for large scale systems because generators are ad-hoc and are often difficult to adapt to new components or architecture properties.

The new design methodologies and tools are opting for a more reasonable approach since they focus on a particular aspect of the system for which they generate adequate optimized code. In this context, tools such as Catapult C Synthesis [18], Bluespec System Verilog [138] focus on the synthesis of implementation for parts such as DSP, or control, etc [128], and so on.

In a context where the design approaches focus more on methodologies based on abstraction and refinement, we can also integrate design automation to ensure a good connection between the tools at each level of abstraction. Design Automation, thus, guarantees the proper exchange of data between the various tools while strengthening the interoperability between tools. We address this matter in the next sections.

2.3.3 Interoperability and Integration for Embedded System Design

The variety of modules that make up the systems imposes new design flow (or design flow refactoring) integrating several design tools. In the EmS domain, integration of these tools poses huge problems of interoperability that are managed by the definition of tightly coupled tool integration frameworks for which the set of tools is known in advance and integration efforts are made around these specific tools.

2.3.3.1 Tool Interoperability in the RTES

Tool interoperability refers to the ability of different tools to exchange and communicate (data, services, etc) in a consistent way. Process automation tools (e.g. BPMN [204]) allow the subsequent firing of activities based on a sequence of scheduled activities. The interoperability of data refers to the exchange of data between tools. The EmS world is relatively late in addressing these concerns, which is explained by the complex nature of the data that are produced by the various tools.

The problem of interoperability in this context is exacerbated by the lack of a common infrastructure serving as support for exchanges between tools in the domain of EmS. Moreover, the reluctance of the tool-providers to invest in the production of bond interfaces for different tools is an obstacle for consistent tool interoperability. However, there is hope with a few approaches and projects which are more interested in the interoperability aspect of tool chains [163][202][167].

Interoperability, generally speaking, is defined as the ability of several environments (e.g. tools, people, physical entities) to exchange information seamlessly clearly specified interfaces and protocols.

In this case, the specification of the information exchange techniques is mainly based on foundations such as the resolution of the syntactic and semantic interoperability.

On the one hand, syntactic interoperability is revealed as being the ability of several tools and environments to exchange data using common data formats , interfaces and formally defined protocols.

On the other hand, semantic interoperability provides various tools with the ability to exchange data in a consistent and efficient manner, thanks to the definition of upstream common interpretation mechanisms or via semantic transformations between tools .

Tool integration in a development process is the description of the common rules and techniques used as supports for communication between tools. These supports respond to integration issues such as the definition of the common data formats, the common interfaces and functions between tools.

There are several works in the literature along the lines of improving tool integration in the SW development processes [75] [169] [168]. However, the pioneering work of Wassermann [203] provides a structured vision of tool integration by providing dimensions to characterize the tools:

- The platform scale used to describe the services implemented in a framework of integration;
- Presentation dimension allows you to find solutions to improve the interaction between users through integration environment;
- The Data dimension allows the improvement of the use of data by the integrated tools. Data must in this case be handled in a consistent manner;
- The Control dimension improves environment handling functions and must take into account the design process of the environment;
- The Process dimension focuses on the role of tools in a development process. This dimension ensures that the interaction tools are effectively a chain.

Quote: *What does "integration" mean? Integration is a property of tool interrelationships. Understanding it will help us design better tools and integration mechanisms.* [191]

2.3.3.2 Tool Interoperability within Tool Integration

2.3.3.2.a Common Design Flow Infrastructure (CDFI)

Pimentel [163] defends the idea of the necessary definition of a common infrastructure to ensure good interoperability between tools in a design flow (System Level Design tools). Their approach would be based on the definition of a standardized "Meta-Tool" framework for designing System Level Design Flows (SLDF). In such case, the design stages are plug-ins. The framework presents descriptions of data and the file formats that are standard, thus facilitating exchanges with external tools (plug-ins). With CDFI, one can build some pre-packaged, standardized and customized design flows.

The data produced in design flows are saved in a structured manner within a common repository and with a format which is comprehensible to the tools connected to the repository. The tools in the flow play different roles from : system model design, refinement of models, to system synthesis.

As shown in Figure 2.6 tools manipulating the repository of CDFI models formally declare the type of data they input and those they produce on output. Moreover, each tool should produce the following information (pre-conditions, input requirements, semantics, post-conditions and output definition).

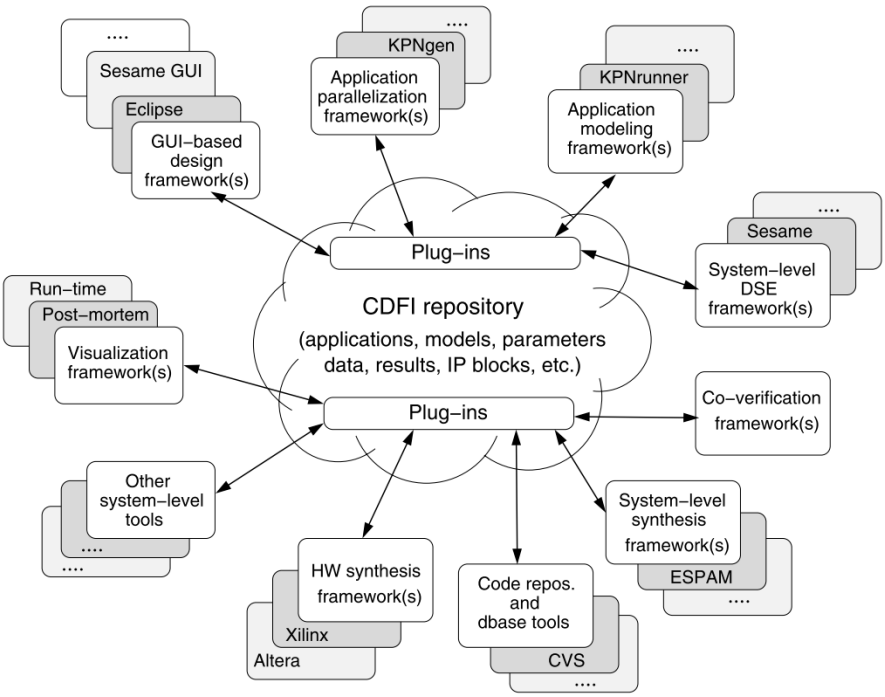

Figure 2.6: CDFI Model integration for different external Tools

For example, a tool such as SESAME must specify that the models it takes as input are the KPNs and that it is capable of generating output composed of performance metrics and multiple architecture instances for DSE.

2.3.3.2.b Levels of Conceptual Interoperability Model (LCIM)

The work of A. Tolk et al. [202] presents a more conceptual vision of the problem of interoperability. In these works dedicated to modeling and simulation, the authors define a conceptual interoperability called Levels of Conceptual Interoperability Model

(LCIM) that describes the different levels of interoperability not only for systems but also for the tools. In their approach (see Figure 2.7), we are mainly interested in the levels of interoperability from 2 to 6 that emphasize one major challenge i.e. semantic interoperability. For a detailed description of the different levels, the reader can refer to [202] [193].

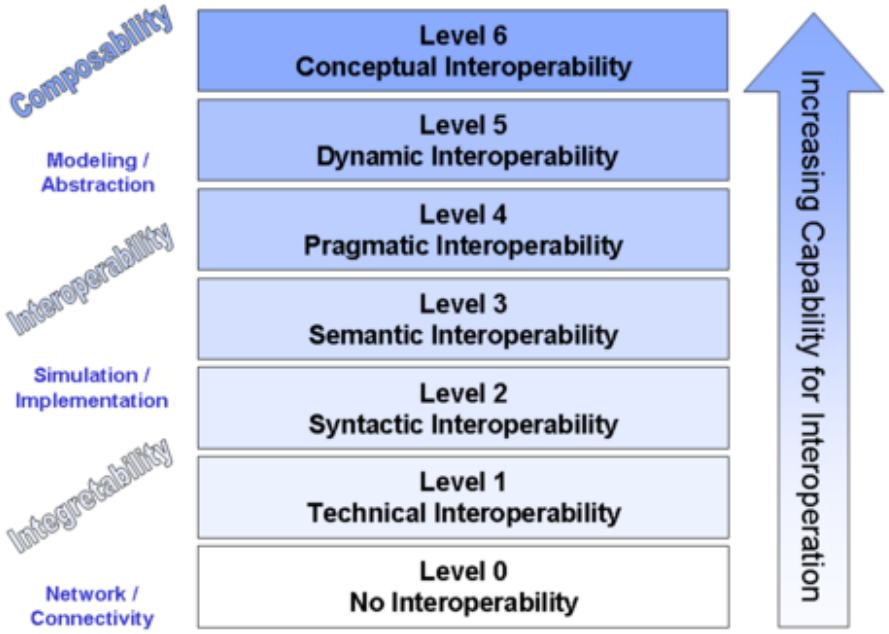

Figure 2.7: The Levels of the LCIM Model

Syntactic interoperability as outlined in Level 2 is reached once several systems (tools in our case) are capable of exchanging data the format of which is known to the various tools. Thus, the formal definition of a common format for data exchange is one of the first steps in a process of interoperability definition.

Level 3 defines interoperability at the semantic level (static) that focuses on the definition of the meaning of data between tools regardless of the manner in which they are represented.

Levels 4, 5, 6 define interoperability at the semantic level (dynamic). This is materialized by the description of the way in which the data are handled; the ability of tools to understand and exploit the changes related to the data over time; finally the ability of tools to have an alignment on the constraints and the assumptions which allow a correct and common interpretation of models.

2.3.3.3 Summary

There are a large number of standard data formats for syntactic interoperability between tools. Among other formalisms, we can cite XML [22], XMI [149] for MDE, RDF [98], etc. In the MDE community the metamodels are used to achieve syntactic interoperability and are serialized in XMI (Section 3.2 presents the MDE and the use of metamodels). Besides the ability to exchange information based on the same formalism, the second challenge raises the important issue of the semantics of models (static and dynamic) [193][163]. Static semantics defines the meaning of a concept in

a language or in a given context (domain). While the dynamic semantics clarify the dynamic evolution of the model elements based on formal rules (e.g. MoC described in 1.2). The two semantics are important to ensure automatic meaningful and accurate interpretation of data from one tool to another.

The above issues are addressed in several major projects targeting the interoperability of design tools. These projects include the following iFEST [86], CESAR[95], MBAT[129], etc. Our work in this thesis is part of the iFEST project. The iFEST project seeks to solve the problems related to the integration of tools for embedded systems. Specifically the project tries to provide an integration environment to troubleshoot data exchange between tools and offers a better management of the product life cycle. The idea is to define high-level interface and interaction services based on an exchange standard called OSLC [12]. From the services, engineers can define prototypes of well integrated tool chains. The services are grouped around the concept of "Tool Adaptor ". A Tool Adaptor allows the data generated by a tool to be transformed in conformity with the OSLC data representation standard (RDF) and uses the services defined to transmit data through the framework.

2.3.4 Remaining Issues

Automation of design activities is tedious mainly because of the number of tools involved during the development processes. Indeed, modern systems are composed of several heterogeneous components and levels of abstraction, for which we can imagine different tools for their realizations. This has the consequence of making the definition of generic design automation almost impossible since the new tools are weakly integrated. In fact, for various reasons (e.g. policy, complexity), most existing automation techniques are done ad-hoc inside a "closed" Framework as in Computer-Aided Software Design tools [168]. Indeed, for EmS, the tools of the same chain are efficiently and tightly integrated, and have generators that allow the data to be moved from one tool to another with minor losses of information. The Frameworks such as Daedalus [139], SESAME, and Metropolis [8] are based on this approach.

Nowadays technical infrastructures to ensure automation allow functionality and service definition to facilitate exchanges between the tools. However, the problem of automation is not just summed up in a question of "technical mechanisms" but also of exploitability of the data that are produced. Indeed, the heterogeneity of the engineering domains and data pose the problem of the consistency of exchanged data. Such consistency of data is limited by syntactic and semantic interoperability problems. While the syntactic interoperability is well addressed with the definition of several common formats to exchange data, the semantic aspect remains a challenge for the definition of automated and integrated design flows.

Moreover, the integration of new tools becomes difficult when it comes to the preservation of the semantics of data that are exchanged. So far there are no sufficiently mature solutions in the different proposed approaches that allow the semantic aspects to be consistently taken into account. Semantic interoperability is achieved when there is a common interpretation and unambiguous understanding of exchanged models, regardless of their representation. The negative consequences of the use of models in different tools come mainly from this breach which constitutes a particularly important handicap when it comes to exchange of executable models the behaviors and semantics of which must be preserved across different tools.

2.4 Conclusion

The important work in the domain of embedded systems has allowed the emergence of several new design approaches that advocate abstraction to reduce complexity of system realization. These efforts have enabled the rise of technologies, languages, methodologies and Frameworks including industrial environments as well as tool-providers.

In the previous sections we have seen some of these significant contributions which now constitute an important basis for the development of EmS.

However, there are several observations that are made regarding the current solutions:

- Most of the environments with an efficient design automation as those stated earlier are based on the use of a single framework that defines strong and fixed couplings of data between tools. Therefore, the tool connection mechanisms are defined ad-hoc and are not visible to users. Consequently, blind faith should not be made to tool providers regarding the semantics of the data being exchanged.

- These environments incorporating fixed sets of activities and (well-integrated) tools have difficulties in accepting the integration of new design tools with unsupported semantics. Indeed, the integration of the necessary semantic adaptations for new tools is a failure and not yet addressed in the context of tool integration. For ESL, the community still suffers very complex and costly manual integration phases where the intervention of experts capable of determining a good integration of heterogeneous environments is needed. This issue has been raised in other works on tool integration pointing out the lack of solutions to address the semantics of data exchanged between the tools of an integration environment [75] [107].

- Our last observation relates to the lack of tools offering higher levels of abstraction. For ESL, the abstract executable languages are at best the SystemC language and its abstractions [90], or libraries on top of existing programming languages such as C, C++ [100][188]. In the context of system engineering, such languages are still very low level.

In summary, ESL currently lacks solutions to offer more abstraction for the description of systems; automated, flexible and reliable environments to connect tools not belonging to the same environment; and environments allowing tool's replacements without weakening the design flow and the data produced.

In the rest of our approach, we present the successful solutions solving some of these shortcomings. Especially in the next chapter, MDE techniques are presented: for better abstraction languages for system design and better design automation definitions. In recent decades, communities such as the MDE have offered solutions dedicated to embedded systems with better abstractions for the specification, analysis and refinement of models of systems. These techniques are accompanied by tools that allow automation of exchanges between different environments to be defined more easily. Indeed, the transformation of modeling tools are often more intuitive and less complex than those handled in other communities, including ESL. In the next chapter, we will precisely study the theoretical basis of the modeling approaches. Such approaches will be used to compensate the failings of the ESL design languages for abstractions. In addition, we examined the failings of the MDE and MBE from the ESL perspective.

3

Towards Formal Semantics in MDE

Contents

3.1 Introduction

As we discussed in the previous chapter, the methodological aspects of systems design advocate raising the abstraction level of languages, the separation of concerns, etc. Moreover, for productivity increase and reduction of time-to-market, tool integration supporting design automation is an important aspect. Besides, we acknowledged that an ideal development framework must be based on the aspects mentioned while ensuring good tool interoperability in terms of data, process, control, etc. Specifically, data exchange must ensure a syntactic and semantic interoperability.

In this chapter, we look at current solutions improving the methodological aspects mentioned as well as the technical aspects (e.g. design automation). We especially describe these solutions in their ability to express abstract languages syntaxes and their ability to preserve the semantics of the data produced from these languages.

This chapter provides solutions based on Model-Driven Engineering and on the opening of this field to ensure a better preservation of model semantics. MDE is a serious candidate to overcome the shortcomings induced by the lack of sufficient abstractions and separation of concerns. MDE does so by promoting the use of Domain-specific modeling languages (DSML) [197] [123].

A Domain Specific Modeling language intends to describe a system focusing on its primer properties. It uses abstracted key concepts that define the system representation.

For embedded systems, the preservation of semantics focuses not only on the syntactic elements (i.e. meaning of exchanged items), but also on the operational aspect, which mainly provides means for parallelism control of system models. Currently, the parallelism control of exchanged models is done through the description of formal models of computation (MoC) that are implemented in tools. Therefore we present two other sections respectively dedicated to the description of MoC approaches and the analysis of the relationships between MoC approaches regarding the MDE features and requirements.

3.2 Model-Driven Engineering

This section aims at describing basic concepts that make the foundations for Model-Driven Engineering. Many of the approaches around MDE were and are motivated by the common ambition of different communities to reduce the complexity of system development processes. These attempts are more or less all based on the same principle i.e. the use of more abstracted descriptions to represent systems and make them understandable for different users.

Quote: *"Model engineering is the disciplined and rationalized production of models. Model-driven engineering is a subset of system engineering in which the process heavily relies on the use of models and model engineering. "*[48]

Model-Driven Engineering stands for the definition of the concepts and principles to offer better abstractions for development processes, it also encompasses process and analysis concerns.

MDE approach as defined by [49] is an integrative approach that aims to integrate different technical spaces (Grammarware, Dataware, etc). Technical space is defined by the set of formalisms, tools and theoretical foundations that helps to describe a model

within a domain. Although the technical spaces might be different there are similar key concepts in the different spaces, that emphasis the concept of abstraction. Indeed, concepts such as model, metamodel and processing exist in the different technical spaces even if they carry different names.

3.2.1 MDE principles and basic concepts

3.2.1.1 Models, Metamodels

A model is an abstract representation of "something" (system, house, car). This representation does not represent the concrete artifact, however, it has enough abstract details regarding this "something" to be understood and interpreted or viewed as that "something". The "Pipe" example in Figure 3.1 shows a picture of a "Pipe" which is actually just a picture of it, that is understood as a "Pipe"[16].

Figure 3.1: The Pipe example

For system design, a model is seen as a formal specification of the structure and behavior of a system reflecting its properties without being its final realization. Thus, at a given level of abstraction , a model exhibits enough details to be representative of a system and to address a particular aspect of a system.

After the definition of what a model stands for, it is convenient to think in terms of the means currently available for defining models.

Models are realized using modeling languages (DSMLs), also called metamodels. In the context of MDE, all metamodels are defined from the same set of concepts from the Meta Object Facility (MOF) standard [142].

According to [17], a metamodel is a set of concepts and the relationships between them.

Indeed, a Metamodel (UML, DSMLs) defines the boundaries of the model and the relationship between the different elements of the model. Therefore it is considered itself as a model describing a set of concepts and their relationship in order to be used to describe other model instances. At some point, a metamodel can be self descriptive, i.e. it can be used to describe itself using its own concepts; such a model is called meta-metamodel (e.g. MOF, Ecore).

3.2.1.2 Relations between Models, Metamodels and Meta-Metamodels

The layered description of the modeling approach is a result of the efforts from the MDE community to find solutions to federate the different emerging metamodeling facilities [140], [147], [144]. As a solution, the Meta Object Facility (MOF [142] was proposed to represent the metamodeling language on top of the other metamodeling languages i.e. (meta-meta modeling) language (self-defining) [17]. This is illustrated

by the pyramid classification exposing the relationships between the different layers of modeling languages (Figure 3.2).

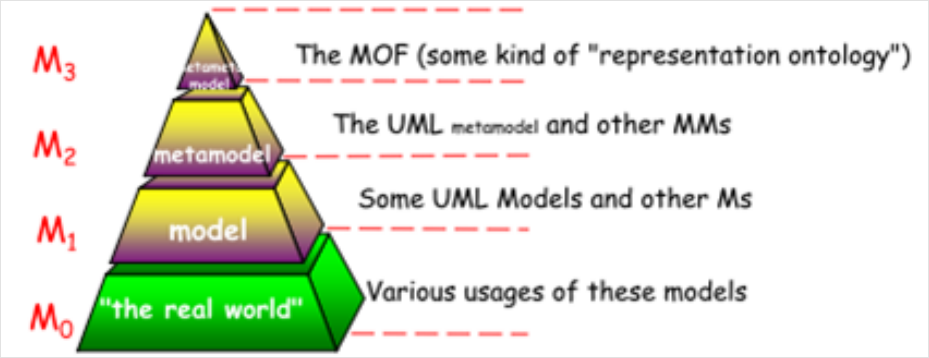

Figure 3.2: The Layers between Models, Metamodels and Meta-Metamodels

In the Pyramid, the lower level represents the real artifacts from the real world (M0), these concrete artifacts can be abstracted into models with various representations, the representations are defined at the (M1) level; the M1 models are defined using modeling languages i.e., meta-models defined at the (M2) level; and finally the (M3) level corresponds to the description of the MOF used to represent all meta-models.

However, this representation is very controversial since it does not give any formal definition of the relationships between the different layers i.e. the formal relationship between a model, metamodel and a meta-meta model. Therefore, such approaches face skepticism within several communities [48].

Quote: *"Is MDA about studying the Egyptology? "* [48]

Several attempts to formalize the different relationships have been made, and [48] is one of the prominent researchers [49] [124] proposing a formalization of the relationships.

As shown in Figure 3.3, the models, metamodels and meta-meta models highlight basic relations [47]. These relations are in most cases derived from conformance (e.g. conformsTo) and representation (e.g. representationOf) relations e.g. a model conforms to metamodel a while a model can be a representationOf a system. The conformance relationship is also reusable between the metamodel and the meta-meta models.

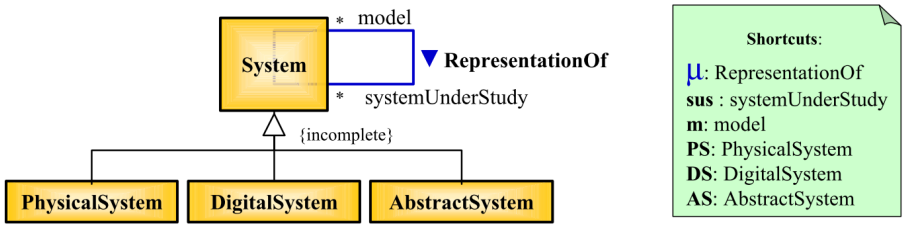

Figure 3.3: Example of relations between Models, Metamodels and Meta-metamodels

All these parts of the MDE framework participate into using strong abstractions as key elements giving good definition of uncoupled domain specific concerns fostering reuse and automatic generation of models for different targets (tools, platforms, etc).

In the next section, we address model transformation that basically controls how synthesis or binding between models are made using models and metamodels.

3.2.2 Model Transformation

The transformation rules are cornerstone design elements. It is through the transformations defined between different models, different views, and different levels of abstraction that designers provide a link between the artifacts they handle and communicate. Several works are currently interested in the definition and classification of different types of transformations. For instance, [194][104][34] have taxonomies describing different types of transformations. The transformation in this context addresses the refinement and translation of a given model into its corresponding output model both conforming to metamodel definitions (we previously defined the possible transformations between the PIM, and PSM PDM in the context of MDA.).

The objective of the processing steps is to provide one or several outputs from given input models. The output result can be a synthesized code in a given language (code generation); a different representation of the model (model's syntax and semantic translation); or documentation.

We can classify transformations as horizontal or vertical. Horizontal transformation (e.g. refactoring) do not necessarily include a refinement of the input model, it may materialize, for example, a change of view. The vertical transformation allows refinement of the models. For instance, in the MDA context, all transformations are potentially vertical due to the refinement of behavior models and architecture models. The transformations can also be horizontal except the $PIM \rightarrow PSM$ that is a refinement in all cases.

The description of transformations in MDA relies on the description of rule patterns executed by the transformation engine. The rule syntaxes conform to a transformation metamodel to describe them (see Figure 3.4-a). Transformation metamodels are defined from MOF thus providing a uniform format complying with the format of the candidate models for transformation.

The transformation rules are divided into exogenous or endogenous transformation depending on the models and metamodels involved for the transformation. As shown in Figure 3.4-b: an exogenous transformation involves at least two modeling languages. The transformation goes from a source model (with a given abstract syntax) to a target model representation (conforming to a different abstract syntax). For this particular case, the problem of semantic translation is a recurrent difficulty. In the case of the endogenous transformation, source and target models are defined using the same modeling language (same abstract syntax).

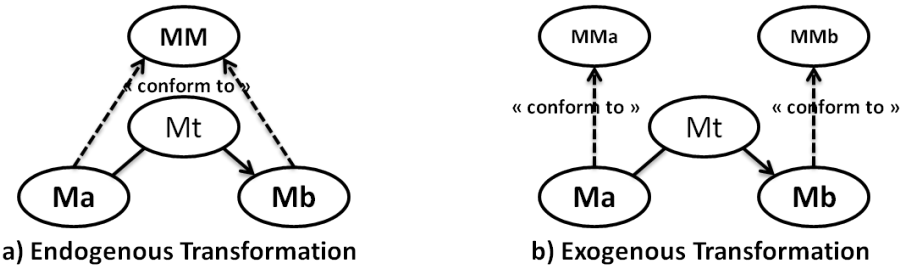

Figure 3.4: The Types of Model Transformation

The rules defined for the transformations mainly rely on techniques, or standards such as QVT (Query, Views, Transformations) [156], ATL (Atlas Transformation Language) [94], etc. Depending on the technique/standard used, the rules are written differently. For example, the QVT standard uses declarative types of rules, unlike Ker-

meta [91] which lays down imperative rules. ATL defines both types of rules. The first solutions for the transformation of models were integrated within CASE environments [104]. Nowadays, Eclipse [41] offers an integration environment used for different types of activities, including modeling and transformation. The project Eclipse Modeling Foundation (EMF) [27] has been an opportunity to integrate functionality to make tooling for modeling and the transformation of models based on the Ecore[1] formalism. The EMF API in Eclipse allows models to be manipulated directly (transform, integrate and refine) directly models. However, several tools can be developed above the API. For instance, besides the above transformation languages (i.e. ATL, QVT, etc), there are more recent environments that tend to propose new solutions for model transformation: the Acceleo [137] of Obeo or MDWorkbench [183] of sodius transformation tools.

3.2.3 MDE methodologies and Standards

Figure 3.5 shows some of the approaches within the MDE domain.

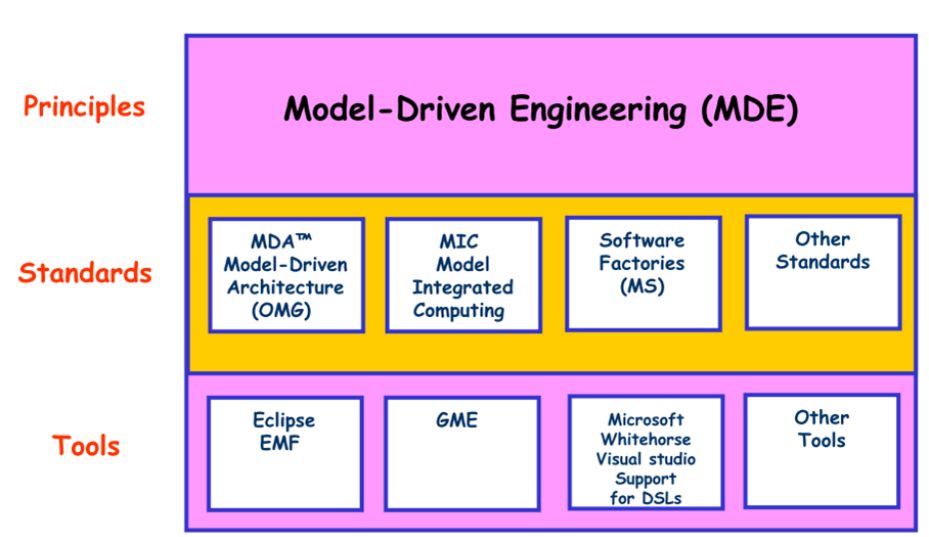

Figure 3.5: The different parts of MDE

3.2.3.1 MDA

The Object Management Group (OMG) [146] is presented as an institution that aims to provide solutions to facilitate the integration and interoperability of different environments.

Model-Driven Architecture (MDA) [194] [104] [166]is an OMG standard encouraging the separation of concerns between the system's functional models and their potential target platforms. This separation emphasizes the reuse and refinement of the functional models into different platforms. Without the platform details, the models are also more easily analyzed. To this end, the MDA specification defines concepts that highlight the separation of concerns for system models. The definition of viewpoints is one of the approaches that provides the separation of concerns that aims to provide several representations of a system based on specific criteria. The definition of separate

[1]The Ecore formalism is a standard for the description of metamodels in the same way as the MOF

viewpoints is an abstraction, because users are limited to a set of elements to study and to analyze (e.g. behavior, performance, dependability, etc) [194]. Today, a great difficulty of the MDE is to ensure consistency of the views and more specifically, the consistency of information exchanged between these views. Since each view is likely to possess its own models with their syntax and specific semantics, model transformation from one view to another poses the problem of the preservation of the models' semantics during pre / post transformation of models.

The general view in Figure 3.6 illustrates the concerns' decomposition adopted by MDA: the Platform Independent Model (PIM) represents the models for which the platform specific information has been abstracted away. Therefore, any PIM highlights only concerns related to the structure and functionality of the system model. The Platform Description Model (PDM) has the purpose of abstracting the characteristics of agiven platform at one level of abstraction while giving details on its properties and topology. The model resulting from the mapping of a PIM on a PDM is called the Platform Specification Model (PSM). The PSM is also an abstract model that is the refinement of a PIM with details related to its target platform. However, the final implementation is derived from the PSM by binding mechanisms to generate the implementation code specific to the environment (e.g. C, VHDL, etc). In some cases of MDA the designers only consider the two model layers PIM and PSM where PSM will include the target platform information [104].

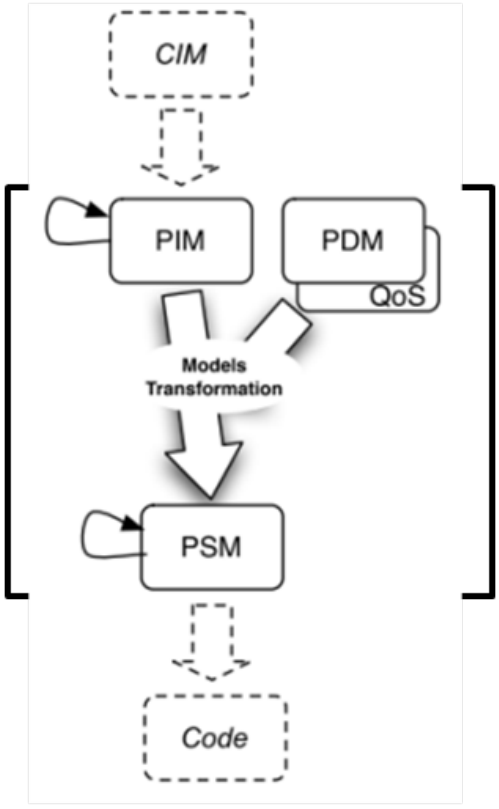

Figure 3.6: MDA Separation of concerns

With these basic definitions, several transformation schemes are defined between the different types of models: $PIM \rightarrow PIM$ is rather a refinement process where abstract system models are refined into more precise models; $PDM \rightarrow PDM$ and $PSM \rightarrow PSM$ are both refinements of the model platform; while $PIM \rightarrow PSM$

result from the mapping process that combines PDM with PIM. In order to integrate the different approaches willing to apply the above methodology, the MDA also encouraged the use of a common modeling language for the communication between the different models. We notice that the separation of concerns as described in the MDA is quite close to the separation of concerns as defined in the ESL community. These similarities are due to the fact that they are both inspired from the Y-chart methodology [103].

The use of models is central to the MDE approach, therefore we give a few important definitions on the notions of models, metamodels and transformation of models in MDE context.

3.2.3.2 Other Standards

There are a large number of standards and tools based on the MDE approach. Among the standards are: the Unified Modeling Language (UML), the Meta-Object Facility (MOF), the XML Metadata Interchange (XMI), the Software Process Engineering Metamodel (SPEM), the MOF Model-to-Text language (MOFM2T) [150] and the (QVT). A part from the standards, EMF framework is an eclipse project for model-based design. Many tools are currently developed around the EMF framework. Ecore is the metamodel description format implemented in EMF as a metamodel (Ecore defines itself). The use of this formalism allows all Modelers based on Ecore to use the tooling proposed by EMF. The serialization format associated to Ecore models is the XMI.

3.2.4 Challenges

The use of models is based primarily on the use of a certain abstract syntax. In this case the transformations between models consist of translations from one abstract syntax to another. Today one of the major problems for the interoperability is due to the difficulty in preserving the meaning of models in different contexts. Indeed, the semantics of models is useful to set a correct interpretation of the models in their theoretical and technical context of definition. Semantics also ensures that the models remain valid in other technical spaces for analysis or validation. Several contributions for the description of the semantics of models were conducted in the MDE community [72] [136]. In the next section, we propose to study the major semantic categories and their relationship to the abstract syntaxes (metamodel).

3.3 Semantics in General and Semantics in MDE

The semantics clarifies the meaning of the language constructs allowing models with unambiguous meaning to be built. The definition of the semantics of models is often obtained through knowledge of the domain it is supposed to represent. It corresponds to the identification of what represents each abstract concept in the domain where it has a meaning.

3.3.1 The types of Semantics

Approaches such as [72] [92] (see Figure 3.7) consider such description as a mapping between language (concrete syntax / abstract syntax) and the semantic domain that gives the meaning. This semantic domain is also a set of formal definition that can be related to any notation. Thus, the semantic mapping must take into account the nature of the semantics that is mapped. Several types of semantics have been studied

and classified in the literature. Figure 3.8 shows this classification. There are mainly two types of semantics: the static and dynamic semantics.

Figure 3.7: Mapping of syntax to semantic domains

Static semantics is very close to axiomatic semantics since it can have the same representation but is only interested in the statically properties. Such properties can be analyzed at compile time. Consequently, static semantics gives meaning to model elements without any assumption on the way they are executed.

In this thesis we examine more closely dynamic semantics also referred as the execution semantics [206]. Indeed, a consistent analysis (for simulation or model-checking) is not possible without a formal definition of the execution semantics of models that are handled by the tools.

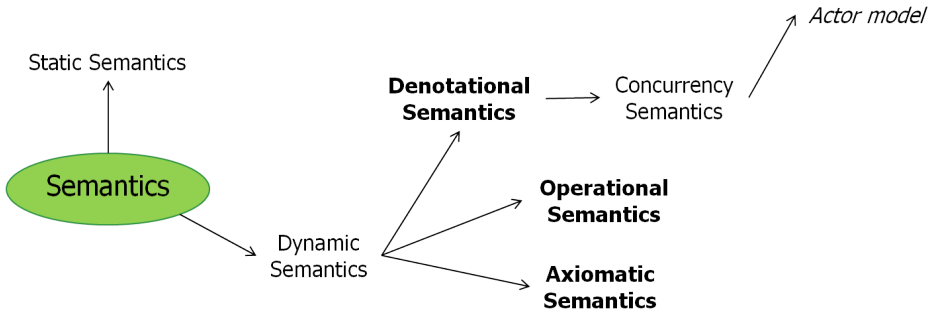

Figure 3.8: Example of Semantics Classification

The execution semantics as shown in Figure 3.8 can be divided into several semantics: operational semantics, denotational and axiomatic, etc.

Quote: *"The different styles of semantics are highly dependent on each other. For example, showing that the proof rules of an axiomatic semantics are correct and are related on an underlying denotational or operational semantics. To show an correct implementation, as judged against a denotational semantics, requires proof that the operational and denotational semantics approved. Moreover, in arguing about an operational semantics it can be an enormous help to use a denotational semantics, which often has the advantage of abstracting away from unimportant, implementation details, as well as providing higher-level concepts with which to understand computational behavior. "*[206]

3.3.1.1 Axiomatic Semantic

Axiomatic semantics [32] is an abstraction of the denotational semantics (described mathematically) which aims to define the meaning of a program based on the logical properties that define the semantics. The idea is based on verification of a set of predicates that specify the meaning of a program (pre and post) execution on the machine's memory. The semantics is often associated with the work of C.A.R Hoare

[81] and R.W. Floyd [53] for proof of program correctness properties. For example, for proof by Hoare's method, the axiomatic semantics are described as follows: The properties are expressed in the form of a triplet {a} E {b}: a is a precondition in the logic of predicates that the memory of the machine must check before executing the action/command E. After execution, memory checks for b only if a have been checked earlier.

- {a} E {b} means: If a is satisfied and E ends, then, after execution, b is satisfied after execution. *Partial Correction*;
- and, [a] E [b] means: If a is satisfied, then E ends, and the memory status satisfies b after execution. *Total correction.*

3.3.1.2 Denotational Semantics

Denotational semantics [52] gives formal mathematical descriptions of the concepts from programming languages, or modeling languages. The descriptions give the basis that facilitates the proof activities [179]. Denotational semantics [69] [178] applies to the domain theory from which it extracts foundations to provide a complete and formal description of the meaning of programs. This is materialized by the use of the partial order, continuous functions and "least fixed point" theories to provide the formal description of the semantics. Languages such as Haskell [125] are based on denotational semantics. The denotational semantics are clearly written and restricted to the set of properties that describe how to run a program. Thus, the description of the denotational semantics of a language is often associated with translation semantics consisting of the projection of the language to a formal domain e.g. projection of a language to Petri networks [162] [161] that has a denotational semantics. Petri networks are typically used for the description of parallelism and for solving the problems of synchronization between components (programs).

3.3.1.3 Operational Semantics

The operational semantics addresses the description of the dynamic behavior of languages and explains how an abstract machine runs a program (model) from the language. Thus, it allows the properties of computer programs to be analyzed (correctness or safety) in an operational manner. The preliminary work on [178] led to the clear separation between the notion of denotational semantics and operational semantics. Furthermore, [164] [165] clearly established the idea behind operational semantics and described its constituents. During this period, several experiments also contributed to facilitating the understanding of what an operational semantics should be e.g. the use of lambda-calculus [9] to clarify the operational semantics of LISP [185]. The founding principles of the operational semantics consider three key elements for the description of the operational semantics of a language: syntax, semantic properties, and the computation.

- The syntax stands for the syntax of the programming language (or modeling) required to clarify semantics. The modeling languages must describe the behavioral model and the operational semantics associated with it using the same language. To fulfill such a property, the DSMLs need to natively have mechanisms to describe the execution of the concepts step by step, or integrate mechanisms of weaving as defined in Kermeta which offer the ability to add an action language to describe the operational semantics on the abstract syntax [91].

- The semantic properties describe the evolution of the program step by step. The idea is to specify and build the set of rules defining the succession of memory state changes during execution. Memory changes take into account a number of (pre/-post) conditions. The rules are inductive and represent individual conditions to move from a memory execution state to another. [164] has identified two types of operational semantics: natural (big-step/NOS) and structural (small-step/ SOS). A NOS [21] has less detail and defines the way in which the execution results of a system as a whole are obtained. Thus, natural semantics only considers the initial execution state and the final execution state. The SOS is recommended in most cases because it allows non-determinisms often induced by the NOS to be avoided. Moreover for SOS, the behavior of a program is described through the behavior of all its parts. This definition implies that one can have several intermediate memory execution states and each state is likely to return a perceptible and analyzable value. The SOS makes sense in this context since it means a finer description of the succession of memory states during execution of the program. The SOS defines several inference rules defining transition relationships from one state (stage of execution) to another. The mechanism is quite close to the definition of a transition system.

- A computation corresponds to the processing that transforms the values in the memory state. It affects the passage from one execution state to another since the changed values have an impact on the conditions of state changes (evaluation of the state successor).

In summary, the definition of the successive memory state changes (in regard to some conditions) and gives a meaning to a program or a model by relating its execution. For the objectives that we described earlier (i.e. analysis, implementation, generation of tools), this last semantic offers more guarantees because it provides more relevant details for analysis or code generation; it also provide details for the integration between tools with different execution semantics. In the remainder of this thesis we especially focus on the use of operational semantics to describe the execution of embedded systems models. As [206] argues, the different semantics are complementary. As such, to formally prove operational semantics, it is not uncommon to find their corresponding denotational semantics. If we take the example of the UML StateChart that describes behaviors in a operational way, the work of [109] offers a formalization via denotational semantics of the meaning of the language.

3.3.2 Semantic expression in MDE

With the proliferation of tools in different domains of engineering, the design phases (particularly the analysis phases) should benefit from the description of the tools' formal execution semantics, which are tool specific in most cases and based on some computation paradigm. In this particular context, a correct semantic mapping (static semantic mapping) cannot guarantee an efficient analysis of the model from one tool to another.

The implementations of execution semantics are strongly coupled with their implementation language or platform. The abstraction of the execution semantics of the models would help the MDE to reach a new milestone by allowing the separation of execution semantics with the implementation related to the constraints of the languages and platforms, keeping only the execution logic. The use of execution logic will have several objectives:

- It allows the lack of preservation of the semantics of the models to be addressed during the analysis which is detrimental to consistent analysis and behavioral consistency;
- It allows the determination of the level of compatibility between tools based on the compliance of their underlying execution semantics at the earliest possible moment;
- It allows the consistent refinement of models until the fine grain code generation for a given platform, thus constituting the mapping of the behavior of models on the platform execution semantics.

Several (ongoing) works began a few years back to address the problem of the description of the models' execution semantics [136][46] [132] [1].

3.3.2.1 Defining OCL constraints for models

The expression of the semantics of modeling languages is a problem that has long been managed by the description of constraints on abstract syntaxes, in order to restrict their underlying models. Constraints are expressed in many ways and start, for example, with the description of the multiplicity relationships linking the concepts of the abstract syntax. Besides, the OMG has also defined a standard called Object Constraint Language (OCL) [153] [172] for the description of constraints on the abstract syntax of the models to clarify their meaning (static semantics). OCL was initially defined for the UML modeling language to express declarative constraints on these structural elements. The constraints are described in the form of pseudo-code defining invariance rules of the model in the form of pre and post conditions; expression for model navigation; or Boolean expression. However, the language does not express the behavior of models at runtime. The use of the language has been recently extended to MOF [142] approaches allowing the integration of constraints on the modeling languages derived from this standard. Thus, several other languages such as QVT [156] transformation languages are based on OCL for the description of the semantics, navigation (object query) and the transformation of models. Examples of OCL use are provided in [3] [64]. More recently, environments such as Kermeta allow the semantics of the models to be expressed and their behaviors at runtime.

3.3.2.2 Kermeta

Kermeta [91] is an environment which has been developed since 2005 and dedicated to the implementation of several activities around the manipulation of models (e.g. modeling, description of constraints on metamodels, model transformation and model execution). In regard to the activities allowed by the tool, Kermeta uses the concepts from MOF, QVT, OCL and aspect-oriented programming [102] and the metamodel implemented in the tool is Essential MOF (EMOF) [151]. The Kermeta framework proposes an aspect-oriented approach for the weaving of control behaviors (operational semantics) on the concepts of a language conforming to EMOF. The action language to describe the operational semantics is imperative and close to the OCL and Java syntax as shown in Figure 3.9. The Figure illustrates an attempt to express a simple operational semantics on a metamodel concept called *MoCComponent* of Cometa. The description and the weaving of the complete operational semantics of a DSML allow models not only to be run, but also to be navigated. However, for any model described

in Kermeta, the proposed execution is purely sequential and the description of the formal operational semantics is generally defined implicitly.

```
aspect abstract class MoCComponent inherits NamedElement
{
    attribute ownedPorts : MoCPort[0..*]

    attribute runnable : ecore::EBoolean

    operation fire() : Void raises MoCComponentException is
    do
            stdio.writeln( self.name + " fired ...")
            if(self.getMetaClass.name=="BasicComponent")
            then
                var basic :  BasicComponent
                basic ?= self
                result := basic.behavior.runBehavior("token")
            end
    end

}
aspect class BasicComponent inherits MoCComponent
{
    attribute behavior : FSM

}
```

Figure 3.9: Excerpt of operational semantics expressed in Kermeta

3.3.2.3 fUML

fUML (Foundational UML) [155, 33] is an OMG standard to describe the semantics (static semantics and dynamic semantics) for a subset of UML accurately and formally. The chosen subset which is "UML2 Superstructure metamodel" is used to describe generic concurrent entities or physically distributed systems. In its fundamental form, any model definition based on fUML must strictly be defined with this subset of concepts; any semantics described must comply with the execution semantics proposed for this subset of elements. Standardized Action Language For Foundational UML (ALF) [154] has been developed to serve as an action language for fUML. Some execution engines for fUML have been developed in environments such as Papyrus [108] [59].

We have chosen to present these three approaches (i.e. OCL, Kermeta, fUML) expressing semantics in an MDE context, however there is an large number of interesting works addressing this problem [37] [30] [198].

3.3.3 Challenges

MDE approaches contributed significantly to the design of embedded systems, in terms of abstraction of system design languages (models, metamodels: MARTE, SysML, etc); in terms of tooling for automatic code generation and transformation of models between

tools (code generation between DSMLs and from DSML to SystemC, C++, C, etc.); and finally in terms of simulation and model-checking of models for analysis at multiple levels of abstraction.

Unfortunately, on this last point MDE approaches are today suffering from a lack of consistency and formalization of executable models. The problem of the executability of models cannot be solved as long as the formal execution semantics of the models are not sufficiently taken into account in the design process and during exchange between tools. Specifically, in the domain of ESL, the abstraction of the behavior of the different modules is done using the abstract descriptions of computation models (MoC) [60]. The MoC formally determine the model's execution semantics. For embedded system design and implementation, they are used to describe the control of parallelism. Therefore, the lack of MoC description *intra* and *inter* tools has an impact on their interpretation. Given their importance, the next section is dedicated to the description of the MoC Theory.

3.4 The MoC Theory

There are several types of computation models that can be bound to different engineering domains (e.g. signal processing, control-command). The concept of Model of Computation is part of a larger domain which is the theory of computation that has been intensively studied during many decades [181] [117]. The theory of computation gave birth to most of the current programming languages by providing the basis for what they should do and how they should operate. Early stages of computation description include for instance the automata theory [83], the Turing machines [195], later followed by other theories such as lamba-calculus [9] or process networks such as [96] [114] for parallel processing's. There are many definitions of the concept of Model of Computation, even including decompositions of the term Model of Computation to Model of Concurrency and Communication. As suggested by last term, it is all about defining the way in which the different parts composing a program or a system (subcomponents) interact to produce the behavior of the system. Thus, the formalism aims to express the concurrency of the different parts and the way they communicate. It addresses the way in which an abstract machine interprets and evaluates the evolution of computing systems over time. The notion of "evolution over time" is dependent on the nature of the system. For instance, "evaluation over time" can be measured in terms of the interaction of the system with its environment, responding to stimuli from the environment with zero delay, these systems are so called perfectly synchronous [70] [13]. There are different categories of computations based on the properties of the systems. In the following sub-sections, we will see different groups of formally defined models of computation and the classifications that have been made over the decades to identify MoCs according to the properties of the systems. We can note that each category is close to the engineering domains for which it defines the theoretical foundations for the realization of components.

3.4.1 Data-Flow Oriented MoCs

The data-flow oriented MoCs relate to the categories of computational models focusing on the processing of streams of data. The processing of big data streams imposes requirements on how the streams can be read, processed and written into memories without causing tremendous memory use, overwriting, or inter-blockings. Even if most

of the data-flow oriented systems are not critical (except for medical), it is rather annoying to obtain inconsistent processing results after computation. Therefore, the MoC theory helps to define the computation rules to avoid such issues, or at least to detect them at analysis steps.

The data-flow oriented MoCs are based on the description of data-flow graphs. Data-flow graphs clarify the interdependence relations between the different data processing nodes using the size of data that are handled in I/O by each processing node.

In some cases, the connections between nodes (edges) bear a weight highlighting the size of data that are exchanged between two nodes. In such models, the availability of the data determines the execution of modules. There are several examples of data-flow oriented semantics in the literature including Data Flow (DF) [93], Synchronous Data Flow (SDF) [113]. There are numerous theoretical models of computation for data-flow and we do not intend to define them all, however we give two examples of such MoCs and their underlying rules.

- Synchronous Data Flow: most signal processing algorithms can be modeled by the SDF MoC (Synchronous Data Flow). It is a Data Flow graph for which the production and consumption data rates are known before execution. Execution policy is given by static mechanisms searching possible (ordered or partially ordered) execution sequences for the model. The static Scheduling requires searching for eligible periodic executions called PA (S/P) S (Periodic Admissible Sequential/Parallel Scheduling). The following techniques are frequently used when searching for a periodic scheduling: solving equation balances (in Ptolemy) [26]; building and solving the topological matrix [113]. If no PA (S/P) S is found the model is flawed and would be impractical.

- Synchronous Data Flow with Multi-Dimensional Data Arrays [20] [31]: is used for intensive signal/ data processing. The system combines parallelization of tasks (computation) and a parallelization of the data (data parallel reading/writing). It is described as a blend of functional blocks (n subcomponents) that produce / consume data. Each component has a *RepetitionSpace* (Vector). The execution of an application based on the Array-OL semantics depends on the following information: the expression of data parallelism (i.e. the dependencies between data arrays allowing a minimum order of execution of the components to be set); the topology of the application (it is obtained by constructing the directed acyclic graph which gives the relations of dependencies between the components of the system); the number of times that each component should be executed to produce or consume an array (is given by the product of the values defined in the *RepetitionSpace* for each component). Having the different information, any scheduling to solve the equations of data dependencies is valid for the system.

3.4.2 Control-Flow Oriented MoCs

Control-Flow oriented MoCs are more frequent for the description of the evolution of systems and are clearly older than the other MoC families. The first control MoCs include the automata theories, Turing machines and more recently approaches based on parallel process networks. The MoC formalisms provide a description of how the various sub-parts communicate taking into account their concurrent nature. The control-flow MoC have more impact on critical systems and are regularly used to solve and reason about the properties of models of systems implemented (rail control systems, landing gears, etc.).

The control-oriented MoCs are generally used to describe sequential algorithms where the different states of execution can be captured. They are represented in the form of a control graph (Petri Nets), or Finite State Machine (FSM). The evolution of a control system is thus conditioned by the successive states of the system. The description of the dependency links between the various states and the verification of the state change conditions help to describe the sequencing of the computations between modules.

We present some examples:

- Kahn Process Network [96]: The KPN MoC suggests that write requests are "non-blocking" and read requests are "blocking" if the memory is empty (empty FIFO). From this property, we deduce that the implementation of the MoC *Scheduler* must guarantee an instantaneous *Ack* response to each module that enables a "non-blocking" *Write* request. For the reception of a "blocking" *Read*, an *Ack* must be sent to retrieve the requested data if available in the FIFO, otherwise the request is blocked.

- CSP: The underlying operational semantics of the CSP MoC suggests that write and read requests are synchronized, therefore they are blocking. There is no need here for a FIFO to store data. Usually, shared variables are sufficient for the communication. According to this property, the implementation of the MoC scheduler must ensure that *Ack* are sent to the reader and the writer if the requests simultaneously reach a synchronization point in time.

For system design, the overall behavior of system modules is given by the combination of the behavior of each module and how they communicate (message passing, shared variable, etc) according to the MoC. The ensuing interactions define the evolution of the system over time. Such evolution is measured by the transformations applied to models (change of states) or the processing of the data.

There are several other MoC families that highlight formal properties that are not necessarily included in the two groups previously cited e.g. the continuous MoC formalisms like (CT [120], ODE [110]). In the next section, based on the state of the art we present the relationships between the main MoC families.

3.4.3 MoC Classification

In this section, we show some examples of taxonomies that have been proposed to group the computation models and study their compliance. [176] proposes a classification of these different models of computation showing the relationships that can exist between them (Figure 3.10). In the following figure, we can see a possible classification of some MoCs as well as their relationships.

Part of the computation models cited in the Figure was implemented in the Ptolemy II [26] tool. Some of Ptolemy II's modeling domains [44] are FSM [112], DE [135], Continuous Time (CT) [120], Communicating Sequential Processes (CSP) [182], Process Network (PN) [63], etc. A second classification was proposed by [88] based on the abstraction of time.

According to [87], from specification to implementation, the complexity of a system can be managed through several levels of specifications based mainly on four MoC concerns: *computation*; *communication*; *data* and *time*. These different concepts determine the level of concurrency in the system model and must be taken into account during all the development steps (especially the refinement and the analysis).

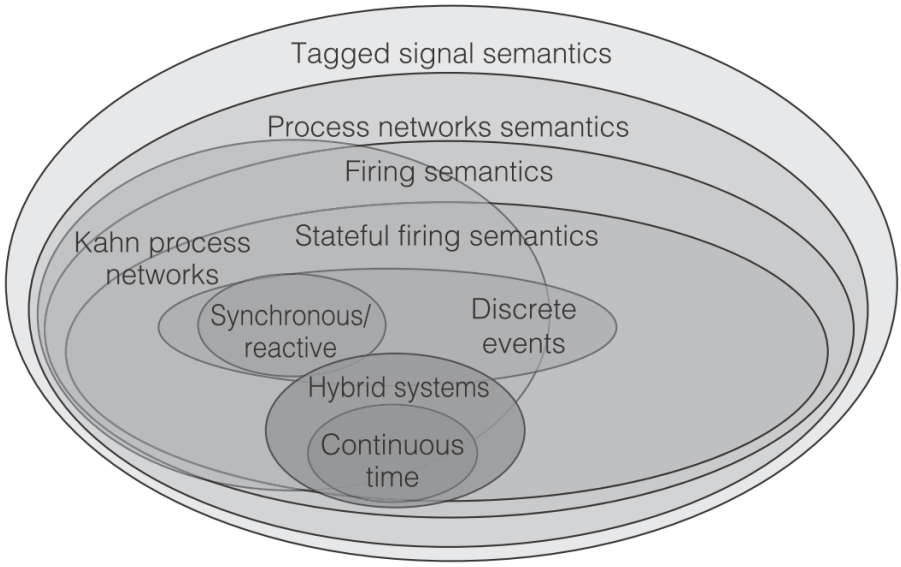

Figure 3.10: Example of MoC Semantic Classification

The level of concurrency determines the parts of a system that can be executed concurrently and the parts that require synchronization or sequential execution.

For example, the high-level models properties can be described without any reference to the concept of time as a first step: focusing strictly on the representation of data and dependency relationships between components.

However, to reach an implementation on a platform, the time properties of the system's components must be taken into account. These new properties are integrated in the system model at some point in the refinement process, opening possibilities for new types of analyses.

In this context, one can classify the properties of models taking into account the following criteria: Untimed models, Timed models, Asynchronous models, Synchronous models: The classification presents 3 categories:

- UMoC (Untimed MoC): Processes communicate and synchronize based on the order of events in the absence of time.
- SMoC (Synchronous MoC): The timeline is abstracted into uniform intervals. Every computation within an interval occurs at the same time, but the intervals are completely ordered along the timeline, and the evaluation cycle of processes lasts exactly one time interval. This category is further separated into two, which is based on whether the output event of a process occurs in the same time interval as the corresponding input event (perfectly synchronous MoC) or whether every process undergoes a delay from an input event to an output event (clocked synchronous MoC).
- TMoC (Timed MoC): This MoC is a generalization of SMoC. Timing information is conveyed on the signals by transmitting absent events at regular time intervals. In this way, processes always know when a particular event has occurred and when no event has occurred. TMoC differs from the synchronous MoC on two points, the granularity of the timing structure is much finer and a process can consume and emit any number of events during one evaluation cycle.

3.4.4 MoC Composition

Each previously presented MoCs belongs to a computing paradigm. Nevertheless, the heterogeneous nature of the systems requires interaction of the different paradigms for a global solution. The heterogeneous composition of the MoCs must also be taken into account in the phases of abstraction at high-level. The work on the classification already gives answers. Indeed, the classifications provide a basis for the study of *compliance* using the existing relationships between different MoC. There are mainly two types of composition: the hierarchical and non-hierarchical composition.

The hierarchical composition as implemented in Ptolemy exploits the flexibility of certain MoC implementations such as PN) to define hierarchical composition. In other words, the composition is defined first by the compatibility of the MoCs, then by the level of expressiveness and restriction of each MoC. The MoCs with the most restrictive properties are hardly usable to express those with less restriction. Conversely, the less restrictive MoCs may serve as a basis for representing other restricted MoCs through mechanisms adding restriction properties to constrain the flexible MoC. For example, it is easier to represent SDF from PN than the opposite. In [62], works around the composition of semantics in Ptolemy helped to define basic composability rules between the MoCs. The relations are categorized from the most restrictive MoC to the more expressive thereby defining possible compositions between them.

The non-hierarchical composition is mainly based on the use of interface components to adapt components that rely on different MoCs. The MoC adaptation is provided inside the interface component that manipulates the inputs and outputs of the other modules to which it provides an adaptation. This approach is used in several works such as [133] with the Heterogeneous Interface Component (HIC); and the work of [174] with the use of adaptation components called *Domain Interface*.

In Figure 3.11, the upper part shows how composition between components is realized with Ptolemy. The top-level model is presented on the left. It has two components A1 and A2 and a *Director* D1 that implements the rules of a given D1 MoC. A1 and A2 follow the execution rules induced by D1. A1 is an atomic actor while A2 is a composite actor that contains a sub-system constrained by another *Director* D2. The hierarchical decomposition of A2 is the model on the right, composed of two atomic actors A3 and A4 (and the D2 *Director*). The composition of D1 and D2 is not explicitly presented and is hidden in the implementation framework. However, the composition here is hierarchical. The lower model extracted from ForSyDe, presents several processes: P1, P2 and P3 are based on the MoC A, while P4 and P5 are based on the MoC B. Because of the connection between (P2 and P4) or (P3 and P5), two *Domain Interface*s are used to translate the semantics of MoC A to the semantics of MoC B. In this example, the *Domain Interface*s are at the same level as the other processes which highlight non-hierarchical composition.

The two types of compositions have their advantages and disadvantages. The hierarchical composition does not explode the complexity because it offers a better abstraction approach since each hierarchical level can be seen as an abstraction that hides the modeling details of the lower modeling level. However the semantic adaptation layers are not visible for users because they have fled into the kernel. The lack of visibility of the semantics of the transition is clearly problematic for analysis and model transformation. On the other hand, the non-hierarchical approach offers more visibility on how adaptation has been accomplished. Unfortunately, the problem lies in the multiplication of the number of adaptation components when the systems are highly heterogeneous.

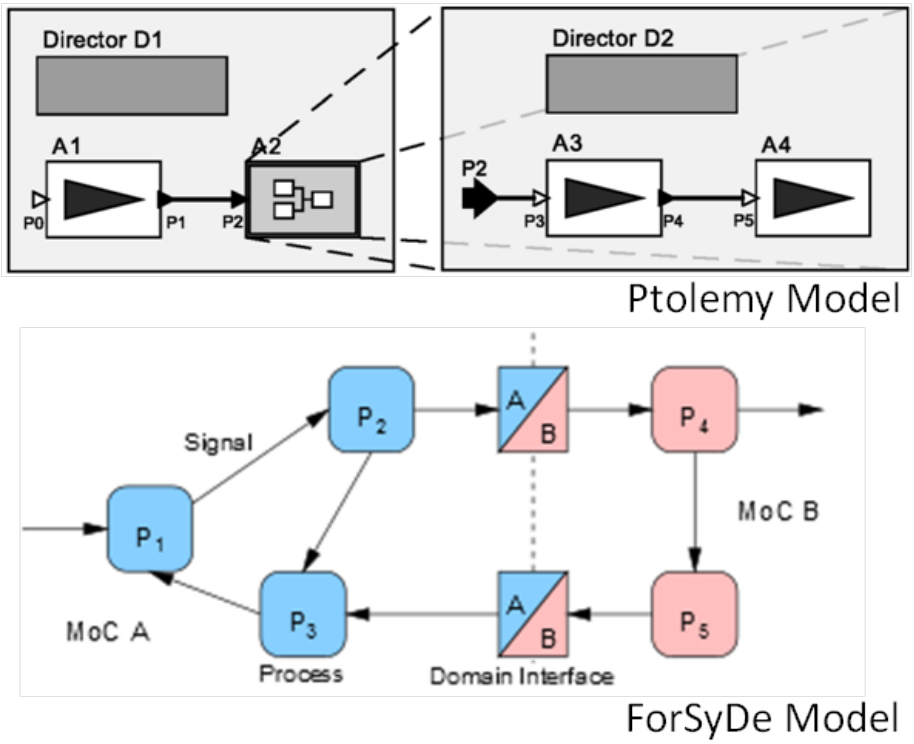

Figure 3.11: Models from Ptolemy and ForSyDe

The diversity of properties and formalisms to express MoC has also led to the emergence of works to find an abstract and formal description of a common representation of MoCs. For instance, [115] proposes the description of the "Tagged Signal" language offering enough abstract concepts to describe different MoC semantics. "Tagged Signal" theory focuses on the description of the systems as components (Behavior) exchanging signals. A Signal contains a sequence of partially ordered events. An event has tags to represent the carried data, and other tags to represent information. Depending on how the second tag is used, the sequence of events can be totally ordered. In fact, the meaning of the tags is crucial to the representation of different semantics e.g. discrete (second tag takes value in the natural set), continuous semantics (tag takes value in the real set), both implying total ordering of events.

The power of MoCs is their formal nature and their mathematical description. The mathematical description provides an abstract representation which is easier to use for Reasoning on execution properties. For instance, if one wishes to abstract DSP components, the associated MoC abstraction (e.g. SDF) can be used to analyze properties and find scheduling. However, to dynamically use MoCs for analysis, their implementation must be provided. In the context of ESL design, tools are strongly based on such MoC theory [77, 80, 79] [8] for the description of the abstract SW models as well as the HW models. The MoCs are described in the form of libraries of execution semantics on top off runtime engines. Within the MDE context, the explicit definition of MoC-based execution semantics is missing. However, we will try in the next section to describe some MDE technologies that refer to MoC descriptions.

3.4.5 Frameworks for System Design based on MoCs

There are many well-known frameworks and tools that propose solutions for EmS design, where the abstraction and use of MoCs is central. However, it is important to describe them since most of them are the current best MoC-Based tools on the market of EmS design and analysis.

- Mescal [66]: aims at heterogeneous, application-specific, programmable multiprocessor design. It is based on an architecture description language. On the application side, the program should be able to use a combination of MoCs which is best suited for the application domain, whereas on the architecture side, an efficient mapping between application and architecture is to be achieved by making use of a correct-by-construction design path.

- Ptolemy [26]: provides an environment for defining and modeling of communicating systems based on hierarchical components. They have the two main concepts that are the actors and directors. The *actor* (concurrent entity) can be seen as a component that communicates with other components through MoC rules well-defined by the *Director* which describes the communication. Ptolemy defines its own execution engine that defines how components are built and more specifically how they communicate and execute at their borders. However, the way MoCs are implemented is unique and MoC definitions cannot be used outside the context of Ptolemy.

- ModHel'X [19]: The ModHel'X framework has many similarities with Ptolemy. However, the author defines the concept of hierarchical blocks and point interface for communication and a system based on snapshot (triggering updates of data passing among components) to simulate the system. This is a major difference with Ptolemy which improves the explicit definition of semantic adaptation between heterogeneous hierarchical levels (using different MoCs).

- ForSyde [174]: addresses the design of heterogeneous embedded systems supporting several models of computation (MoCs). In ForSyDe, systems are modeled as concurrent process networks that communicate with each other via signals. It uses the concept of process constructors, which leads to a formal and structured model, where communication is separated from computation. Processes belonging to different MoCs communicate via well-defined domain interfaces. With ForSyDe, it is possible to model and simulate complex electronic systems, where some parts are modeled with the continuous time MoC, while digital HW is modeled with the synchronous MoC, and SW with the untimed MoC.

The tools presented above benefit from capitalizing many years of experience in the specification and the formal analysis of embedded heterogeneous systems. They allow the consistency of the overall semantics of heterogeneous models within their own framework to be analyzed.

However, in relation to the previously established criteria i.e. intake to raise the abstraction level of languages, to assume syntactic and semantic interoperability of models, to improve the design automation in tool chains, these tools do not significantly contribut to those aspects because they are not intended to address them. Moreover, such tools are not actually considered as MDE based, hence the need to consider the MDE solutions and their relations to the MoC approaches in the next sections.

3.5 MoC in the context of MDE

The MDE has experienced important progress in system modeling topics, which has attracted the attention of the community specialized in heterogeneous system design (ESL, PBD). Consequently, several approaches have been launched to provide solutions to EmS design [118] [58] [170] [74] [180] [40]. The solutions not only target description of heterogeneous models, but also target their execution at high-level. Despite the considerable efforts, the MoC issues are recurring and not well addressed by MDE tools. As such, the MDE tools must adjust to the height by providing tools usable in an ESL context or sufficiently effective to ensure the development of a system flow on several levels of abstraction with mature MoC descriptions. Waiting for this day to come, there are at least some hopeful solutions for EmS design in the MDE context such as [8][189]. In the next sub-sections, some of the MDE-based languages and frameworks for EmS design are presented.

3.5.1 MARTE

The Modeling and Analysis of Real Time Embedded Systems (MARTE) [148] is a UML profile for EmS design and analysis. This profile introduces new concepts that allow the real time aspects to be taken into account, thus filling the gaps in the standard profile SPT [152] (Scheduling, Performance and Time). MARTE profile allows the concurrency among different components to be specified by defining the concept of "RTUnit", and defining the communication among components with the concept of "RTConnector". The profile also allow the reuse and simplification of the concepts in other EmS profiles. For example, the concept of Quality of Service and Fault Tolerance in MARTE is derived from the standard profile QoS & FT (Quality of Service and Fault Tolerance). The MARTE profile is built around 3 packages promoting the separation of concerns:

- the *foundation* package provides the basics of language dealing with non-functional property modeling, time modeling, generic resources and allocations;
- the *design* package allows execution platforms for HW and SW to be modeled on several levels of abstraction;
- the *analysis* package provides mechanisms to annotate models for analytical purposes. It provides generic concepts to cover several types of analysis such as the possible scheduling or performance.

Advances were made on execution semantics' aspects (synchronous, asynchronous, models of time). Even if an important step has been crossed with this profile (by the explicit description of information related to the execution semantics), the language does not offer the description of operational semantic models that can be used for model's effective execution, or further to adapt the semantic differences between several modules of a system. Indeed, the final description of the execution semantics is provided by formalisms external to the profile. For example, the execution semantics associated to the MARTE time model are provided by the (CCSL) [5, 35] framework. Similarly, the runtime properties (i.e. synchronous, asynchronous, etc) attached to the communication and scheduling elements are implemented with other formalisms.

3.5.2 SysML

At the structural level, SysML [73] defines an architecture that is based on Blocks or hierarchical components that are derived from UML components e.g. Composite Structure diagram. One of the main strengths of SysML is the fact that it describes new types of diagrams allowing the description of requirements and the parameterization of system models. One of its disadvantages is that it does not address aspects related to the execution semantics of the models it describes, thus leaving flexibility for their interpretation. Besides, even if it allows through mechanisms of profiles computations to be accessed such as the PN models, DE [135], SR [42]; it does not mention if these various models of computation will be interoperable in its environment.

3.5.3 MoPCoM

The MoPCoM [200, 106] design process is an MDE oriented methodology dedicated to the description of systems by abstraction/refinement techniques, including design space exploration, and PBD (Platform-Based Design). The process defines several levels of abstraction in the development process, where design and analysis can be performed. As shown in Figure 3.12, the MOPCOM process applies an MDA (PIM, PDM, PSM) approach and defines three levels of representation of the system modules.

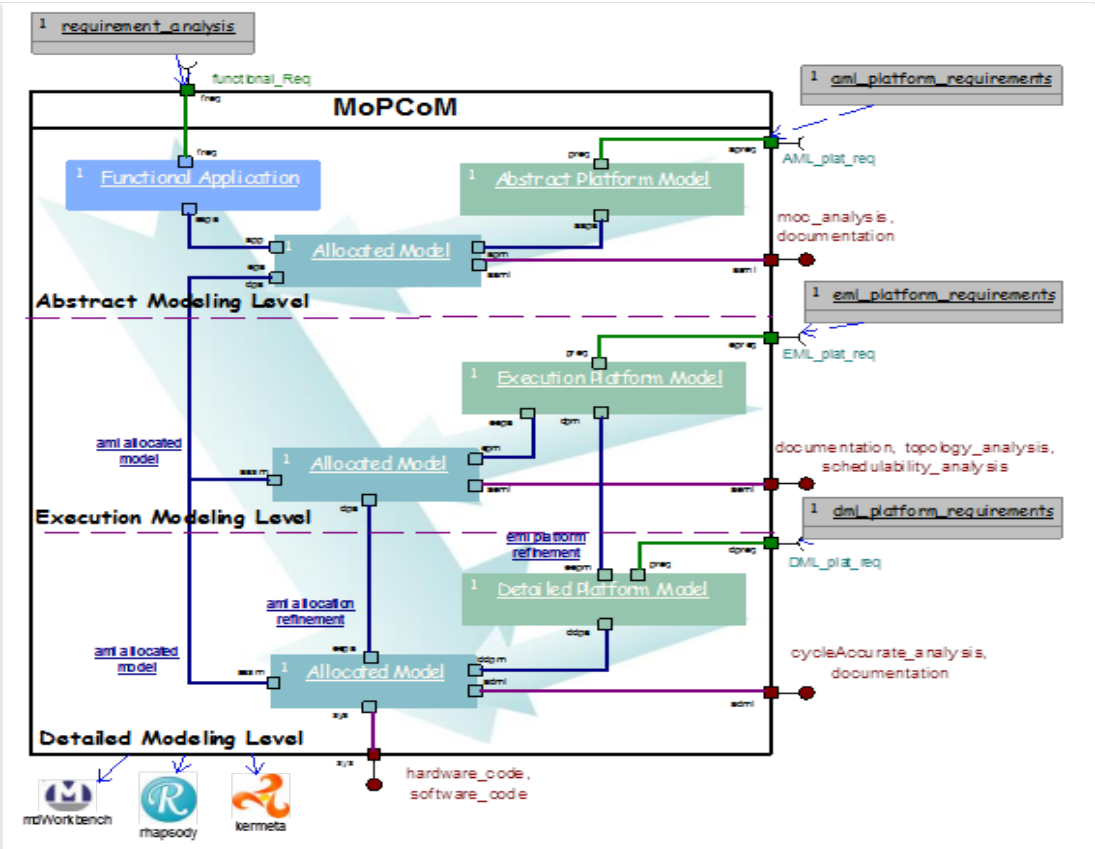

Figure 3.12: MoPCoM methodology

- Abstract Modeling Level (AML) targets the expression and analysis of concurrency and communication of the system modules. It allows the operational spec-

ification of the interactions between concurrent modules without assumptions or constraints on the platform resources.

- Execution Modeling Level (EML) targets the expression and analysis of the models (from the AML level) while they are allocated on physical structure models that provide coarse-grained details on the nature of the final platform.
- Detailed Modeling Level (DML) targets the description of the fine-grained platform (e.g. by refinement of the physical models from EML) enabling finer analysis of the allocated system models, which will provide a final implementation of the system.

3.5.4 Metropilis

Metropolis [8] is an electronic system design environment that provides formal analysis, simulation and synthesis capabilities. It allows the difficulties from the interconnection of tools to be reduced including interoperability from a semantic point of view. The tool relies on a formal metamodel with an unambiguous semantics reusable for the specification of the application functionality, for their analysis and the description of the architectures and mappings. In Metropolis, the systems are represented as communicating concurrent entities (networks of behaviors). Behaviors produce events controlled and transmitted by the Metropolis execution engine to the different tools with respect to some chosen MoCs. Analysis steps result in formal verification and simulation activities. The approach offers the abstraction of generic concepts to capture and translate execution semantics into the concepts of the Metropolis metamodel (execution semantics model). These concepts separately address aspects related to computations, communication, synchronization, etc.

3.5.5 Challenges

For the analysis of behavioral models in MDE, the previously presented approaches (i.e. Kermeta [91] or [37]) propose facilities for the description of the dynamic semantics of models. However, for specific engineering domain semantics, the current dynamic semantics description approaches are poorly oriented in the capture of these semantics. For instance, a description of the execution semantics for a specific EmS domain (e.g. design of a DSP modeling), the abstraction of the structure is easily provided by DSMLs. But, to describe the dynamic semantics of such models, developers must provide the semantics properties underlying the different components using MoCs such as SDF, MDSDF (that need to be captured with metamodels). MARTE proposes concepts to tackle scheduling issues for (synchronous, asynchronous, timed) models; unfortunately the communication semantics (synchronous, asynchronous, etc) are left as semantic variation points. As a consequence, communication between different engineering domain experts still poses problems and impacts the interoperability of models with different execution semantics. Using the MoC theory and classification also including composition rules, would ease communication between experts and give a formal foundation for the integration of different semantics.

3.6 Conclusion

The previous chapter allowed us to identify ESL concerns to improve the design of embedded systems. The major challenges of the future design methodologies include

raising the level of abstraction from the programming languages; the definition of consistent and relevant model analysis phases; the automation of the transformations for model refinements within the tools and between tools; Finally, the above points must ensure a preservation of the semantics of models during the transformation steps and analysis stages.

The MDE has demonstrated abilities to respond to some of the above concerns. Metamodels allow higher levels of abstraction and a common formalism of communication between various engineering domains and tools. They enable interoperability at the syntactic level and the transformation tools enable automation of refinement activities. Furthermore, modeling tools such as UML offer profile mechanisms to capture static properties that are specific to the engineering domains, thus allowing analysis.

Unfortunately, ESL tools can hardly take advantage of abstractions and automation tools offered by the MDE tools because they do not offer guarantees in terms of formalizing and description of heterogeneous formal execution semantics. Indeed, the correctness of a model's behavioral analyses is hardly ensured towards different tools, especially when the tools rely on different execution semantics that are not explicitly described. In fact, tools offering execution engines rarely provide information finely describing the execution semantics behind the engine (synchronous, asynchronous, timed, continuous, discrete, etc). The lack of such information can be a drag for the analysis and communication of models, since there are no analyses of their compliance or composability with other environments. For these reasons, the MDE tools are not suitable for defining a complete design flow stemming the complex analysis issues related to semantics preservation during all the design steps.

The MoC theory seems to provide answers for the above execution semantics issue. The MoC controls the consistency of any analysis activity and describes the semantic properties. For instance, abstracting the implementation of a DSP processor will use specific implemented mathematical models giving details on how such a model should be executed. In other types of implementations, event-based paradigms are used as the abstraction to describe how models are executed.

We believe that the MDE should take into consideration the theory of MoCs for the description of correct analyses and for the preservation of the semantics of models throughout the development process. MoCs give bases for the explicit definition of formal semantics for implementation purposes. They also offer bases for the composition of the execution semantics.

In our approach, we try to have a merging approach between MoC and MDE:

- to improve MDE models with formal execution semantics. The MDE must then provide DSMLs to capture reusable MoCs for the definition of the adaptations between executable exchanged models.
- to improve MDE-based design flows. The captured MoC models are used to ensure semantics preservation between tools (i.e. models from the tools).
- to improve the opening and flexibility of tool chains, thus moving from integrated design flows to opened design flow with replaceable tools.

The following chapter 4 presents our work in the context of IFEST project to address the first two points mentioned i.e. redefining and formalization of the Cometa DSML allowing to express reusable semantic models for the preservation of semantics. Then we will see in section 5.3 the application of the Cometa semantics models through an experimentation on a industrial use case.

4

The Cometa Concepts, Models and Validation

Contents

4.1 Introduction

The Cometa DSML aims at providing MoC models that are used to ensure the consistency and reliability of the system models for the analysis and refinement steps between tools. The MoC models capture the missing MoC properties (static properties and operational properties).

MoC-Based operational semantics in Cometa are intermediate layers between the application models and the execution engines. They allow the behavior of models to be adapted according to the execution rules of given MoCs. The description of the layers follows two criteria which are on the one hand, the structure of layers that define interaction and synchronization topologies; on the other hand, execution control behaviors that are used by the structure, allowing the synchronization of modules. Particular concepts and their relations were defined in Cometa to specify the layers.

Accordingly, each model produced by a tool in a tool chain can be woven with Cometa MoC models providing the missing static and dynamic properties to correctly interpret the models within different tools. Enriched and woven models have enough properties to be transformed in a transparent manner while preserving their semantics.

The methodology associated with Cometa combines preliminary steps including: the identification of MoC semantics and the study of their compositionality that are presented in Section 4.3.2.2.

In the following sections, we formally describe the concepts and relations in Cometa, while showing excerpts of their implementation in the form of a metamodel.

4.2 Foundations of the Cometa Approach

The layers describing the execution control define several concepts that we associate with different concerns e.g. structure and behavior. The abstract concepts in the DSML can be grouped into two main categories for the description of the structure of so-called *Semantic Layer*s (SL) and the description of the behaviors for execution control of the requests issued by the application models. The category for the description of the execution control is divided into *Communication*, *Time* and *Data* concerns. The *Behavior* concern as defined by [87] is not in the scope of our DSML since Cometa does not aim at the programming of application function blocks. The concepts of these categories define several relationships that we will discuss in the sections below. The models produced are defined in the form of libraries.

4.2.1 *Semantic Layer* Definition

The semantic layer's structures are inspired by the ADLs [57] [130]. They emphasize the capture of the hierarchical structure with the possibility of extending each structural element with a behavior or MoC specification to control the execution of the system modules. The modularization induced by structures provides better readability of the models and their reusability. We propose the following definition for SL's.

Definition: A Semantic Layer combines a virtual structure description and several execution control behaviors that are MoC related. The virtual structure decomposition reproduces an existing structure model to add the missing information

which is related to its MoC properties. It allows several layers of semantics adaptations to be captured.

As shown in Figure 4.1, there is an large number of architecture description languages. This excerpt is a small proportion of all the existing industrial or academic formalisms. Consequently, a single definition of the notion of system architecture is hard to provide. In fact the definition varies according to application domains. Nevertheless, significant formalization efforts have been made to identify common semantic grounds for ADLs.

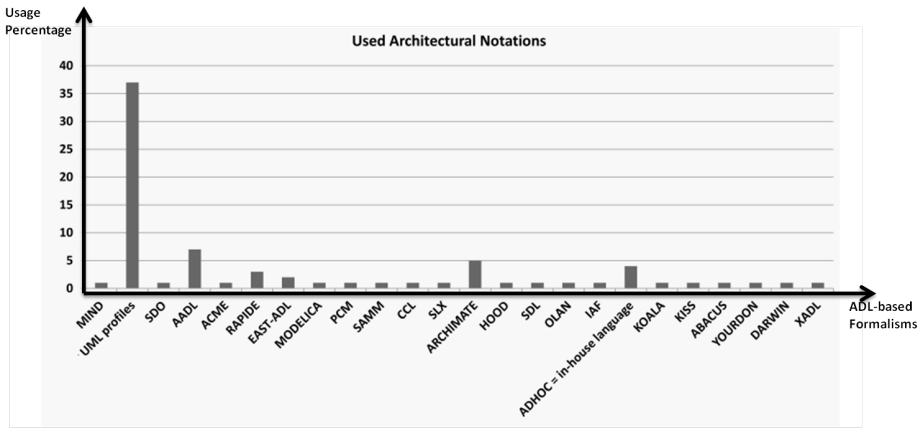

Figure 4.1: Some ADL notations

4.2.2 The system's MoC characterization

Cometa criteria for the description of the MoC properties (static and dynamic) include the four following concerns: *Communication concern*, *Behavior concern*, *Data concern* and *Time concern*. The major contributions of Cometa are at the *Communication concern* and *Behavior concern* levels.

- the *Communication concern*: this concern can explicitly define the high-level communication mediums and enrich them, at each level, with the MoC semantic properties related to their engineering domains. The semantic properties are exploitable by dynamic behavioral mechanisms that define the evolution of the system and the firing of the modules.

- the *Behavior concern*: In Cometa, the description of behaviors makes 2 paradigms of execution explicit: *untimed* or *timed*.

  The description of untimed control behaviors is based on the use of finite state machines (FSM) to regulate the requests from the application blocks. Cometa state machines allow a set of control states to be defined. The machines allow the definition of (as restriction) a processing order of the requests that are sent to the machine. The state changes represent the different steps of control underlying a type of communication formally defined by MoC rules. The logic implemented in the FSMs is abstract and operational. The choice of the FSMs is justified by several criteria which include the following:

  – For consistent system analysis, it is important that their interpretation space is bounded. The execution control behaviors should also be bounded by

using a formalism that allows a finite state space for the interpretation of the system to be kept.

- The FSMs are variants of the Abstract State Machine (ASM). These formalisms are known for their solid mathematical foundation facilitating formal validation.
- The FSMs have been used with success in several areas such as sequential, parallel, distributed, or real-time systems. According to [134], such formalism has formed the basis for the description of the semantics of languages such as C, C++, SDL [25], and VHDL [160] which are widely used for the implementation of systems.
- FSMs are executable and easier to understand compared to other fine grain formalisms such as programming languages. They, thereby, facilitate readability, understanding and expression of algorithms.
- Finally, state machines can be used on several levels of abstraction to represent, the implementations of a system incrementally (successive refinements of behavior).

Thus due to various arguments, we have chosen this formalism to represent the control mechanisms regulating the exchanges between the application blocks and participating in the monitoring of the requests.

- the *Data concern*: the types, sizes and structures of data have an important role to play in defining the executability and the model's execution control behaviors. In engineering domains such as multimedia design, the different processing modules do not necessarily exploit the same amount of data each time they are fired. In fact, certain types of modules need to read the streams of data several times to fully process them. It is important to abstract the sizes and types of data to allow optimal coordination of the different processing modules avoiding data loss.

- the *Time concern*: The timed paradigm allows the description of control mechanisms; the evolution of which is based on the description of clocks specifying the evolution of time. To specify this aspect, we propose the abstraction of concepts allowing the modeling of clocks attached to execution control mechanisms. The modeling of time opens up the possibility to express broader MoC semantic varieties (e.g. DT or CT). The abstracted concepts are very similar to those described in the MARTE model of time. However, our approach does not yet offer fine operational semantics' descriptions for the time models. The idea of using the FSMs to describe such time evolution and constraint is currently being explored but the results are not sufficiently mature for presentation in this chapter. We are also exploring the possibility of coupling our approach to the CCSL formalism which already provides operational and formal semantics for the interpretation of time models for MARTE. The time abstraction gives the ability to reason about the ordering induced by the order relationships and is a first step to the definition of correct scheduling policies. Depending on the granularity of the time representation, time abstraction may be logical or physical. Logical representation uses the notion of time "tick" which is a triggering event that reflects the evolution of time [6][141]. The discrete time representation defines an execution instant that can be clearly identified in the time space. As defined by [6][141] [120] such instants can be represented using the natural numbers set. The continuous time description can be represented with the real numbers set.

The current work on the time aspect in Cometa is presented in 4.3.4.

In the next section, we firstly provide the relations between the concerns we are addressing, and we progressively introduce the formalization of the concerns and concepts of Cometa.

4.2.3 Formal Description of the Cometa concepts

A Cometa execution control layer is mainly composed of a host structure for control behaviors and the control behaviors themselves. The structure sets up a topology of interconnected components via ports and connectors. Control behaviors are local or global: local MoC behaviors are captured as communication protocols and global MoC behaviors are global scheduling behaviors associated to scheduling components.

In Cometa, the concepts of the metamodel are not specific to pre-existing languages or tools. These are common generic concepts defined in the literature for the design of systems that we assemble in a certain way to face the execution control issue. As a result, the models produced from this language can be reused by different transformation, analysis or specification tools. The DSML is described in the form of a metamodel "Ecore".

Although metamodels and other so-called high-level modeling languages are effective for abstraction of system models, many of them lack formal definitions to limit the interpretation of a language. Limiting the interpretation context of a language consists of providing the set of mathematical rules facilitating the unambiguous analysis of system models designed from such language.

Indeed, the denotational semantics allow reasoning about the properties and interpretation of models through the definition of formal semantics and execution rules that describe the evolution of a model (e.g. the transition systems). The formalization of the Cometa DSML below forms the basis for the description of the semantics of Cometa models.

A $\text{Cometa}_{specification}$ is given by the association of an abstract syntax $\mathcal{L}_{Cspec}$; the semantic domain $\mathcal{S}_{Cspec}$ of concepts that are described in the abstract syntax; and the mapping rules associating each element of $\mathcal{L}_{Cspec}$ with an element of $\mathcal{S}_{Cspec}$:

$$Cometa_{specification} : \langle \mathcal{L}_{Cspec}, \mathcal{S}_{Cspec}, \mathcal{M}_{Cspec} : \mathcal{L}_{Cspec} \rightarrow \mathcal{S}_{Cspec} \rangle$$

$\mathcal{L}_{Cspec}$ defines a 4-tuple making the modeling concerns of the abstract syntax explicit. In the same way, semantic domains and mapping rules associated with the language are divided into as many semantic spaces and rules showing the separation of the modeling concerns (in our case Structure (*Str*), MoC (*MoC*), Data (*Data*) and Interface (*Int*)). For convenience in the formalization, we separate the *Data* concern from the description of other *MoC* concerns (e.g. time, communication).

$$\begin{cases} \mathcal{L}_{Cspec} : \langle \mathcal{L}_{Str}, \mathcal{L}_{MoC}, \mathcal{L}_{Data}, \mathcal{L}_{Int} \rangle \\ \mathcal{S}_{Cspec} : \langle \mathcal{S}_{Str}, \mathcal{S}_{MoC}, \mathcal{S}_{Data}, \mathcal{S}_{Int} \rangle \\ \mathcal{M}_{Cspec} : \langle \mathcal{M}_{Str}, \mathcal{M}_{MoC}, \mathcal{M}_{Data}, \mathcal{M}_{Int} \rangle \end{cases} \tag{4.1}$$

The mappings between $\mathcal{L}_{Cspec}$ and $\mathcal{S}_{Cspec}$ that take into account the separation of concerns are formalized and presented as follows:

- $\forall\, \mathcal{L}_{Str}, \mathcal{S}_{Str}, \mathcal{M}_{Str}$ then $\mathcal{M}_{Str} : \mathcal{L}_{Str} \rightarrow \mathcal{S}_{Str}$

- $\forall\, \mathcal{L}_{MoC}, \mathcal{S}_{MoC}, \mathcal{M}_{MoC}$ then $\mathcal{M}_{MoC} : \mathcal{L}_{MoC} \rightarrow \mathcal{S}_{MoC}$
- $\forall\, \mathcal{L}_{Data}, \mathcal{S}_{Data}, \mathcal{M}_{Data}$ then $\mathcal{M}_{Data} : \mathcal{L}_{Data} \rightarrow \mathcal{S}_{Data}$
- $\forall\, \mathcal{L}_{Int}, \mathcal{S}_{Int}, \mathcal{M}_{Int}$ then $\mathcal{M}_{Int} : \mathcal{L}_{Int} \rightarrow \mathcal{S}_{Int}$

$\mathcal{S}_{Str}$: the semantic domain of the structure in Cometa is given by the formalized Abstract Description language (ADL) domain that describes all of the elements to design a topology of components.

$\mathcal{S}_{MoC}$: represents the semantic domain of a MoC. MoC-Based semantic domains represent all the properties that are specific to a MoC and which allow to describe the execution rules of a program or a model based on this MoC.

$\mathcal{S}_{Data}$: matches the set of primitives that are known for the specification of data in general (integer, real, booleans, etc). The semantic domain of the data may also be considered depending on the data that are handled in a specific engineering domain (e.g. Multi-dimensional arrays).

$\mathcal{S}_{Int}$: represents the set of concepts formally defined in the literature and allowing the interfacing of communicating components. For instance, this semantic domain makes references to the IDL formalism types of services (e.g. *Send*, *Receive*).

The mapping functions $\langle \mathcal{M}_{Data}, \mathcal{M}_{Str}, \mathcal{M}_{MoC}, \mathcal{M}_{Int} \rangle$ respectively associate semantics with the concepts defined in the abstract syntax sets $\langle \mathcal{L}_{Data}, \mathcal{L}_{Str}, \mathcal{L}_{MoC}, \mathcal{L}_{Int} \rangle$.

The metamodel excerpt presented in Figure 4.2 shows the separation of concerns highlighting the description of the Structure library (*StructureLibrary* concept), Data Library (*DataTypeLibrary* concept) and the MoC Library (*MoCLibrary* concept).

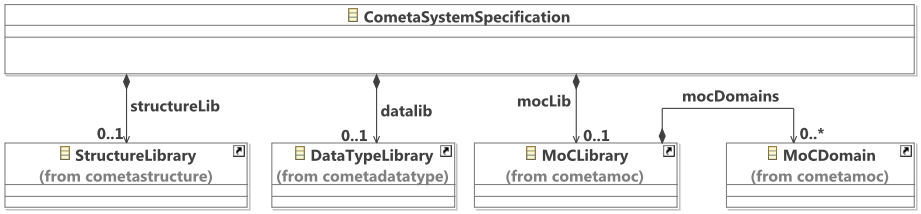

Figure 4.2: Overview of the concerns related to Structure, Data and MoC-behavior

In the next sections, we begin the detailed study of the various concerns of the language and the formal description of the language's concepts. We introduce the following useful rules:

- **a:A** means an element a of type A;
- **{A}** means a none empty set of elements of type A;
- **a.b** means the element b of a;
- **a.{Y}** means the set of elements of type Y from a;
- **x::=a;F** means x is defined either by the element a, or the sets of elements of type F, or a combination of these different elements;
- $f : \mathcal{A} \rightarrow \mathcal{B}$ means that for all elements a:$\mathcal{A}$ then f(a) is an element of $\mathcal{B}$.

The metamodel's *reference*s are defined as *function*s specifying associations (with multiplicities) between the concepts (i.e. meta-classes of the metamodel).

4.2.3.1 Structure Description

The specification of a structure highlights the level of parallelism in an interconnection of communicating components. By level of parallelism we mean the identification of the parts in the system structure that remain parallel once the execution control behaviors have been applied in the model. Indeed, the execution control mechanism allows the introduction of sequential execution between some parts (entities) of the system. This structure description also facilitates the mapping of control machines by clearly separating the control behaviors that are for protocols and those for scheduling.

The description of the semantic layer's is provided by $\mathcal{L}_{Str}$. The language offers concepts for the modeling of libraries of interconnected virtual components representing the topology of a semantic layer. The $\mathcal{L}_{Str}$ is formalized as follows:

$$\mathcal{L}_{Str} : \langle \mathcal{S}tr_{Lib}, \mathcal{S}tr_C, \mathcal{M}oC_{Cn}, \mathcal{M}oC_{Cp}, \mathcal{C}_c, \mathcal{B}_c, \mathcal{M}oC_P, \mathcal{C}, \mathcal{P}_t \rangle$$

The elements contained in the different sets below are formally defined after the presentation of the actual various sets.

- $\mathcal{S}tr_{Lib}$: is a library defined to contain the semantic layers. The networks of interconnected components form the semantic layers. The library allows the description not only of several containers of topologies (i.e. *StructureContainer* and *CompositeComponent*), but also atomic components (i.e. *BasicComponent*s).
- $\mathcal{S}tr_C$: In Cometa, each *StructureContainer* represents a semantic layer.
- $\mathcal{M}oC_{Cn}$: represents the set of connectors that can be defined from the $\mathcal{L}_{Str}$ abstract syntax. The connectors allow the components of the topology to be linked and are named *MoCConnector*s in Cometa.
- $\mathcal{M}oC_{Cp}$: this set contains various components which we define as *MoCComponent*. The $\mathcal{M}oC_{Cp}$ set is divided into two sub-sets $\mathcal{C}_c$ and $\mathcal{B}_c$ with $\mathcal{M}oC_{Cp} = \mathcal{C}_c \bigcup \mathcal{B}_c$.
- $\mathcal{C}_c$: represents the set of composite components named *CompositeComponent*s in Cometa.
- $\mathcal{B}_c$: refers to the set of atomic components of a Cometa structure. Unlike the composite components, the *BasiCComponent*s are the entities directly connected with the application blocks.
- $\mathcal{M}oC_P$: represents the set of *MoCPort*s defined from Cometa. The *MoCPort*s are the interconnection points for the components. The *MoCPort*s are connected between them by the Cometa *MoCConnector*s.
- $\mathcal{C}$: represents the set of describable containers from $\mathcal{L}_{Str}$. By definition, the *StructureContainer* and *CompositeComponent*.
- $\mathcal{P}_t$: finally, the *Part*s are containers insofar as they are able to contain semantic layers. Therefore:

$$\mathcal{S}tr_C \subseteq \mathcal{C}$$

$$\mathcal{P}_t \subseteq \mathcal{C}$$

$$\mathcal{C}_c \subseteq \mathcal{C}$$

The description of the sets is a first step in the description of the $\mathcal{L}_{Str}$ language. The defined sets do not exist independently of each other. Indeed, elements of some sets (e.g. $\mathcal{M}oC_P$) come from the ability of other elements of sets (e.g. $\mathcal{M}oC_{Cp}$) to define them. For instance, the ports are described when instantiating components.

From this observation, we study the properties that bind the various elements of the sets. The described properties make it possible to design topologies in the form of interconnected components since they clarify the manner in which these topologies are built.

4.2.3.1.a Cometa *StructureLibrary*

For Cometa, a *StructureLibrary* (4.2) contains all the topology containers (*StructureContainer* or *CompositeComponent*). The specification of atomic components (*BasicComponent*s) is also accomplished at this level. Consequently, we consider that *StructureContainer*s and *MoCComponent*s are specified at this level. The formalization for a *StructureLibrary* is defined as follows:

$$\forall\ stl \in \mathcal{S}tr_{Lib} \text{ then } \begin{cases} stl ::= \{StructureContainer\}; \{MoCComponent\}; \{Interface\} \\ \{StructureContainer\} \subseteq \mathcal{S}tr_C \\ \{MoCComponent\} \subseteq \mathcal{M}oC_{Cp} \\ \{Interface\} \subseteq \mathcal{L}_{Int} \end{cases} \tag{4.2}$$

In the language, a function (i.e. reference) *mocStructures* is associated to the creation of instances of *StructureContainer*. The same type of reference exists for the creation of instances of *MoCComponent* (i.e. *BasicComponent* and *CompositeComponent*.

The concept *Interface*, further specialized to *MoCInterface* and *RTInterface* is used to define the different interfaces between the languages for computations and the Cometa layers. The definition of the interfaces is not relevant in this section and will be presented in Section 4.2.3.5.

4.2.3.1.b Cometa *StructureContainer*

A *StructureContainer* (4.3) represents a possible configuration of interconnected components. It allows the description of new components, connectors and the mechanisms for component's interconnections. The components are Cometa *MoCComponent*s and *Part*s as shown in Figure 4.3. These two concepts are defined and formalized later in this section.

We can formally define *StructureContainer* as:

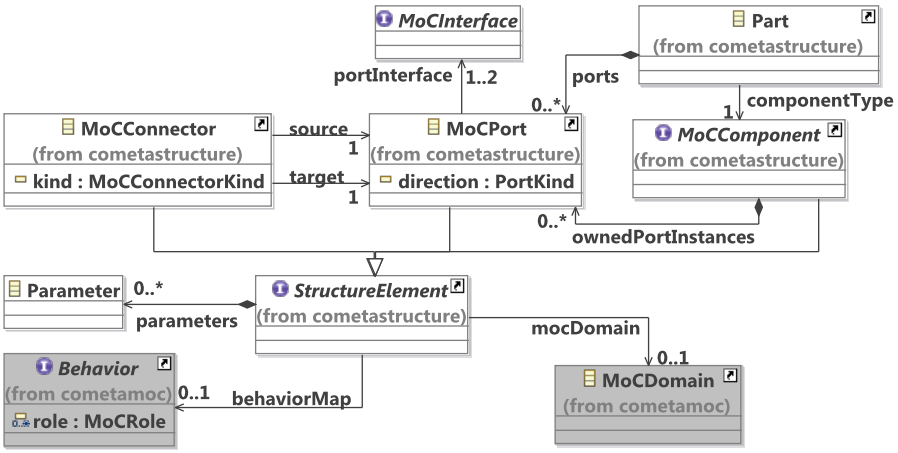

Figure 4.3: Excerpt of the structure description in Cometa

$$\forall\ c \in \mathcal{S}tr_C \text{ then } \begin{cases} c ::= \{Part\}; \{MoCConnector\}; \{MoCComponent\} \\ \{Part\} \subseteq \mathcal{P}_t, \\ \{MoCComponent\} \subseteq \mathcal{M}oC_{Cp}, \\ \{\ MoCConnector\ \} \subseteq \mathcal{M}oC_{Cn} \end{cases} \tag{4.3}$$

In Figure 4.3, the common properties of *MoCComponent*, *MoCConnector* and *MoCPort* (described below) are specified through the concept of *StructureElement*. For instance, the *StructureElement* provides associations to *Behavior*, *MoCDomain* and *Parameter*.

4.2.3.1.c Cometa *MoCConnector*

A Cometa *MoCConnector* (4.4 and Figure 4.3) is a structural concept to bind the ports between the various components (*Part*s or *MoCComponent*s). Links to ports are defined by the $< source, target >$ functions that combine each *MoCConnector* to its specific input and output ports. The connectors mainly contribute to the definition of the execution control mechanisms because they carry behaviors to synchronize components ($< bhv >$ function in 4.4, also called $< behaviorMap >$ in Figure 4.3). Such *Behavior* is defined in 4.2.3.2. A connector is also associated with a type $< ckind >$.

The $< ckind >$ relationship is defined as a function associating a type to a connector. For example, in the incoming data exchange for a *CompositeComponent*, an input port of a *CompositeComponent* is associated to an input port of its internal component by a *delegation* connector. In the outgoing exchanges, a *delegation* connector connects an output port of an internal component with an output port of its *CompositeComponent* container. For other cases, they are identified as being of type *assembly*. We present the connectors by the following formal definition:

$$\forall\ cn \in \mathcal{M}oC_{Cn} \text{ then } \begin{cases} cn ::= source; target; ckind; bhv; \{Parameter\} \\ source\ : \mathcal{M}oC_{Cn} \rightarrow \mathcal{M}oC_P, \\ target\ : \mathcal{M}oC_{Cn} \rightarrow \mathcal{M}oC_P, \\ ckind\ : \mathcal{M}oC_{Cn} \rightarrow \{delegation, assembly\}, \\ bhv\ : \mathcal{M}oC_{Cn} \rightarrow \mathcal{B}ehavior \end{cases} \tag{4.4}$$

Parameter is defined in 4.2.3.4.b.

4.2.3.1.d Cometa *MoCComponent*

The *MoCComponent* (4.5) of $\mathcal{M}oC_{Cp}$ are virtual parallel entities used not only for the hierarchical description of topologies, but also as support for the mechanisms of execution control. The *MoCComponent*s represent atomic and composite components. These two types of components have similar properties such as the ability to define sets of ports and to carry behaviors used for execution control (i.e. $< bhv >$ function in 4.5, also called $< behaviorMap >$ in Figure 4.3). However, when they are defined separately, atomic and composite components provide different properties. Descriptions of the distinct properties are proposed by the 4.6 and 4.7 formal definitions. We propose the following formalization for the common property description:

$$\forall\ cp \in \mathcal{M}oC_{Cp} \text{ then } \begin{cases} cp ::= \{MoCPort\}; \{Parameter\}; bhv; mocdomain \\ \{MoCPort\} \subseteq \mathcal{M}oC_P, \\ bhv\ : \mathcal{M}oC_{Cp} \rightarrow \mathcal{B}ehavior, \\ mocdomain\ : \mathcal{M}oC_{Cp} \rightarrow \mathcal{M}oCDomain \end{cases} \tag{4.5}$$

Each *MoCComponent* has an associated *MoCDomain* defined by the function $< mocdomain >$ in 4.5, or called $< mocDomain >$ in Figure 4.3. The concept *MoCDomain* is defined in Section 4.2.3.2.

The behaviors described in Section 4.2.3.2 participate in the definition of the execution control mechanisms of the components. In the case of composite components, the execution control mechanisms consist in the specification of global *Scheduling* mechanisms. For atomic components, they consist in the description of local control behaviors to associate with other control behaviors defined in the ports, connectors, etc.

4.2.3.1.e Cometa *BasicComponent*

Basic components (4.6) are the atomic concurrent entities directly interacting with the application blocks. In our description, their connection to the application blocks implies the definition of a function $< comp >$ associating the atomic components with the application blocks. The concept of application block (i.e. *Block*) is not part of $\mathcal{L}_{Str}$. The *Block* notion is defined as part of the interfacing description. By definition, we consider a *BasicComponent* as an entity capable of interfacing with such blocks via its interfaces. In this approach, we assume that the function blocks are executed sequentially. An atomic component is formalized as follows:

$$\forall\ bc \in \mathcal{B}_c \text{ then } \begin{cases} bc ::= comp \\ comp\ : \mathcal{B}_c \rightarrow \{Block\} \end{cases} \tag{4.6}$$

Block is defined in Section 4.2.3.2.

4.2.3.1.f Cometa *CompositeComponent*

A *CompositeComponent* (4.7) objectifies the notion of hierarchy. The concept has the same properties as a *StructureContainer* with the difference that it possibly has ports and behavior when interconnected with other components of the same hierarchical level. In this case, the control mechanisms carried by the component serve as "semantic glue" between the sub-topologies and the external components connected to the composite component, thus ensuring overall consistency of the execution policies. Composite components are part of the *MoCComponent*s of Cometa. We formalize the concept in the following manner:

$$\forall\ cc \in \mathcal{C}_c \text{ then } \begin{cases} cc ::= \{Part\}; \{MoCConnector\}; \{MoCComponent\} \\ \{Part\} \subseteq \mathcal{P}_t, \\ \{MoCConnector\} \subseteq \mathcal{M}oC_{Cn}, \\ \{MoCComponent\} \subseteq \mathcal{M}oC_{Cp} \end{cases} \tag{4.7}$$

4.2.3.1.g Cometa *Part*

The *Part* (4.8) participates in the description of topologies by offering the possibility of describing component reusability. A *Part* reproduces the behavior of a composite or atomic execution control module. Elements of type *Part* have ports connecting them with the rest of the structure. To declare the component they reproduce, the $< reusedf >$ function connects each *Part* to the component to which it reproduces the behavior ($< reusedf >$ is called $< componentType >$ in Figure 4.3). Parts may themselves contain other parts which are interconnected via ports. The concept *Part* is formalized as follows:

$$\forall\ p \in \mathcal{P}_t \text{ then } \begin{cases} p ::= \{Part\}; \{MoCPort\}; reusedf; linkedct \\ \{Part\} \subseteq \mathcal{P}_t, \\ \{MoCPort\} \subseteq \mathcal{M}oC_P \\ reusedf\ : \{Part\} \rightarrow \mathcal{M}oC_{Cp}, \\ linkedct\ : \{Part\} \rightarrow \mathcal{C} \end{cases} \tag{4.8}$$

4.2.3.1.h Cometa *MoCPort*

A *MoCPort* (4.9) is used for the communication between components. It is either the entry or exit point for data from one component to another. The *MoCPort* can be enriched with MoC based behaviors. When applying a MoC behavior to a *MoCPort*, this behavior participates in the specification of the communication protocol. The events (generated) from the application requests are sent to the execution control behaviors via *MoCPort*. Such events define interfaces *MoCInterface* that are contracts to guarantee access to the internal *MoCPort* control behavior (protocol) or to guarantee access to the behavior of the component to which the *MoCPort* is attached e.g. *MoCConnector* or *MoCComponent*. The relation to *MoCInterface* is defined by the $< int >$ function in 4.9, also called $< portInterface >$ in Figure 4.3. A *MoCPort* can

be directly connected to another access point (*MoCPort*) of another component. The *MoCPort* can be oriented IN/OUT to specify the direction of the communication (see $< dir >$ function in 4.9, also called $< direction >$ in Figure 4.3). The *Behavior* and interfaces of *MoCPort* constitute its specification. The Cometa *MoCPort* represent the interconnection points of the various entities that make up an interconnection of components. The description of a *MoCPort* brings out a set of properties mainly focused on the description of the mechanisms of execution control distributed between ports, connectors and components. The *MoCPort* is formalized as follows:

$$\forall\ pr \in \mathcal{M}oC_P \text{ then } \begin{cases} pr ::= \{Parameter\}; bhv; int; kindf; dir; target \\ bhv\ : \mathcal{M}oC_P \rightarrow \{Behavior\}, \\ int\ : \mathcal{M}oC_P \rightarrow \{Interface\}, \\ kindf\ : \mathcal{M}oC_P \rightarrow \{end,\ relay\}, \\ dir\ : \mathcal{M}oC_P \rightarrow \{IN,\ OUT\}, \\ target\ : \mathcal{M}oC_P \rightarrow \mathcal{M}oC_P \end{cases} \tag{4.9}$$

The distinct concepts presented in this section are involved in the description of semantic layers, especially in their structural aspect. As noticed in the description of the *MoCComponent* and *MoCConnector*, the associations of behaviors (*Behavior*s) and interfaces (*Interface*s) with the structural element respectively relate to the description of the execution control mechanisms and interfacing with the application blocks.

The behaviors for execution control are provided by operational MoC models that we describe in the next section. The metamodel excerpt shown in Figure 4.3 and 4.2 are parts of the DSML, highlighting the concepts of the $\mathcal{L}_{Str}$.

Cometa allows the description of intermediate execution control layers for concurrent system modules. The idea is to control the access to shared memory in order to ensure proper synchronization between the modules. This is further explained in section 4.2.3.5.

Before proceeding to the description of execution control mechanisms, we show the use of $\mathcal{L}_{Str}$ to describe the possible topological configurations.

4.2.3.1.i Semantic Layer Configurations

There are several ways to define the synchronization of interconnected concurrent components: global synchronization (e.g. synchronous *execution*); synchronization using communication (e.g. synchronous *communication*). The synchronous execution is related to the theory of "perfect synchrony"; while synchronous communication is more about defining communication mechanisms that put in place handshakes, semaphores, etc. Both are related to the theory of computation (MoC). In Cometa, we focus on defining mechanisms for synchronous or asynchronous communication.

In this context, there may be several topologies with Cometa as presented in the following Figure 4.4.

- In the first configuration ①, we consider the direct connection between ports without connectors. For example, the components A, B emit requests through their respective *MoCPort* interfaces. Such a semantic layer configuration will have the execution control behaviors split in two FSMs and each behavior is

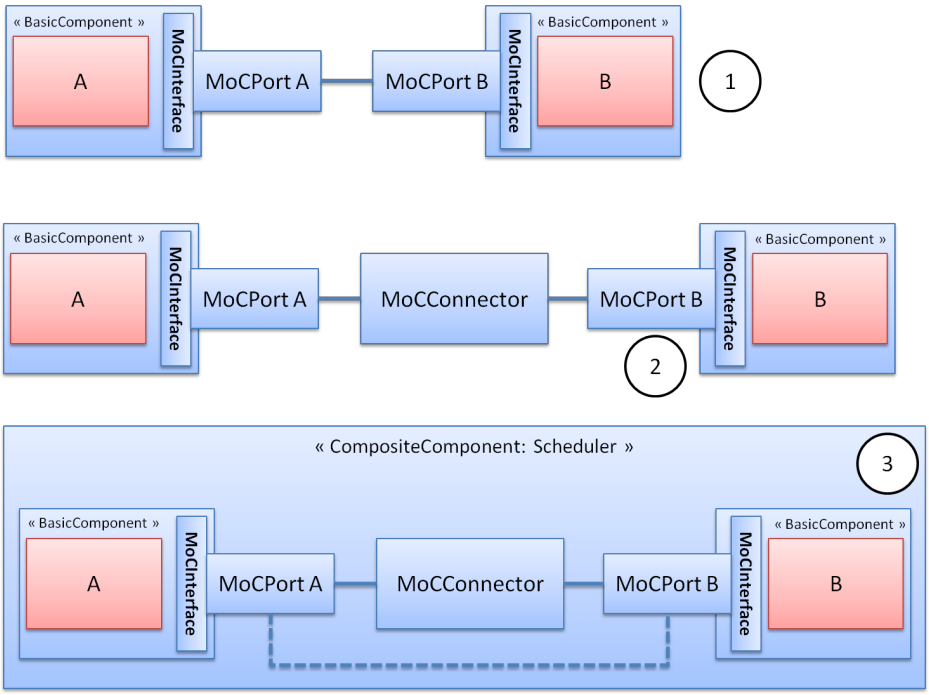

Figure 4.4: Topologies described using Cometa

placed in a *MoCPort* (access point of requests). The control behaviors defined in the two ports protocols handle the synchronization according to MoC rules. The MoC-based synchronization is described in Section 4.3.1.

- In the second configuration ②, there are two possibilities for dispatching the control FSMs. We can consider the case where the control of the execution is divided into three interconnected FSMs (one on each communication element *MoCPort* IN/OUT and *MoCConnector*); or we can also consider the case where there is only one FSM for the execution control placed on a *MoCConnector*.

- In the same configurations as in one and two, the third configuration ③ put in place a hierarchical component that plays the role of a global scheduler for all the different elements it includes (*MoCComponent*, *MoCPort*, *MoCConnector*). Such Scheduler is required if one wishes to make the execution of a composite component totally sequential.

The communication between FSMs is ensured by using specific events (e.g. *MoCEvent*s described in Section 4.2.3.2).

4.2.3.2 Execution Control Description

To solve the parallelism problem within systems composed of concurrent entities, MDE must take into account the concerns of controlling the execution of modules at each level of abstraction considering the synchronization issues that may arise.

Execution control refers to how the constituent modules in the systems interact and synchronize in time. This should be well controlled to ensure proper execution of the system on parallel architectures. Unfortunately, MDE is still struggling to describe this aspect formally and explicitly.

The execution control behaviors associated with the various elements of the semantic layers are designed to regulate the flow of exchanges between application blocks. The behaviors implement MoC-based rules for such control. In this section, we formally describe the $\mathcal{L}_{MoC}$ language to define these rules and execution properties. The concepts abstracted in $\mathcal{L}_{MoC}$ enable the production of executable and reusable behaviors to associate with semantic layers.

A $\mathcal{L}_{MoC}$ is defined by the 6-tuple $: \langle \mathcal{M}_{Lib}, \mathcal{D}_{MoC}, \mathcal{B}_{MoC}, \mathcal{B}_{FSM}, \mathcal{B}_{Op}, \mathcal{T}_{mod} \rangle$. $\mathcal{L}_{MoC}$ defines a library of MoC domains. The MoC domains are divided into two separate spaces for the description of FSM behaviors and the description of time models. We formalize these considerations in the following manner:

- $\mathcal{M}_{Lib}$: represents the unique library of MoC domains from Cometa.
- $\mathcal{D}_{MoC}$: represents the set of definable *MoCDomain*s based on the concepts of Cometa.
- $\mathcal{B}_{MoC}$: represents the set of execution control behaviors. This set is divided into two subsets $\mathcal{B}_{FSM}$ and $\mathcal{B}_{Op}$, with $\mathcal{B}_{MoC} = \mathcal{B}_{FSM} \bigcup \mathcal{B}_{Op}$ and $\mathcal{B}_{FSM} \bigcap \mathcal{B}_{Op} = \emptyset$.
- $\mathcal{B}_{FSM}$: represents the set of behaviors that can be described from the FSMs. Each behavior of this set is defined by the following 3-tuple:

  $\mathcal{B}_{FSM} : \langle \mathcal{S}, \mathcal{T}, \mathcal{E}_{MoC} \rangle$.

  - $\mathcal{S}$: is the set of states of a control machine (or FSM). The set of states includes: an initial state, several final states and several intermediate states. All the states of a FSM represent the state space of the control behavior.
  - $\mathcal{T}$: is the set of transitions to bind states of an FSM.
  - $\mathcal{E}_{MoC}$: is the set of (internal) events for the communication between FSMs in Cometa (i.e. *MoCEvent*).
- $\mathcal{B}_{Op}$ represents the set of opaque control behaviors definable from a programming language, or formalisms sufficiently expressive to specify execution control mechanisms based on the MoC theory (e.g. Petri nets).
- $\mathcal{T}_{mod}$ is the set of time models that can be modeled from the abstract concepts in Cometa within specific MoC domains. The formalization of its concepts is provided in Section 4.3.4.

4.2.3.2.a Cometa *MoCLibrary*

The models belonging to each of these sets are interlinked so as to ensure the description of executable and reusable behaviors. The relationships are presented as follows:

$$\exists!\ m \in \mathcal{M}_{Lib} \text{ such that } \begin{cases} m ::= \{MoCDomain\}; \{MoCEvent\} \\ \{\ MoCDomain\ \} \subseteq \mathcal{D}_{MoC}, \\ \{\ MoCEvent\ \} \subseteq \mathcal{E}_{MoC} \end{cases} \tag{4.10}$$

Figure 4.5 introduces an description of the part in the Cometa metamodel related to the MoC description. The concept of *MoCLibrary* associated with $\mathcal{M}_{Lib}$ is composed of several *MoCDomain* and several *MoCEvent*s.

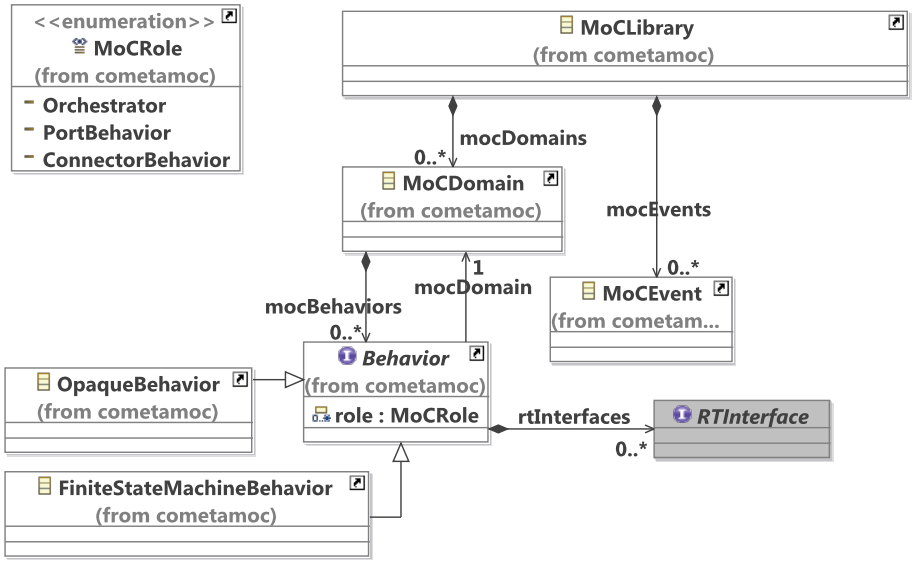

Figure 4.5: Content of a MoCLibrary

4.2.3.2.b Cometa *MoCDomain*

A *MoCDomain* (4.11 and Figure 4.5) is a concept reifying the concept of MoC modeling space. A MoC modeling space captures all of the rules, properties and semantics specific to a MoC. The captured rules include operational behaviors defined through the concept of *Behavior*. As a result, a *MoCDomain* is formalized as follows:

$$\forall\ dm \in \mathcal{D}_{MoC} \text{ then } \begin{cases} dm ::= \{Behavior\} \\ \{\ Behavior\ \} \subseteq \mathcal{B}_{MoC} \text{ which means ,} \\ \forall b \in \{Behavior\} \text{ then } b \in \mathcal{B}_{FSM} \text{ or } b \in \mathcal{B}_{Op} \end{cases} \tag{4.11}$$

4.2.3.2.c Cometa *MoCEvent*

The concept of *MoCEvent* (4.12) is used to describe the communication events between FSMs of a semantic layer. For example, the distribution of control over ports and connectors involves the use of these events as triggers for state changes in ports and connectors. These events are dissociated with interface events that are directly related to the application blocks. The formalization of the *MoCEvent* is as follows:

$$\forall\ ev \in \mathcal{E}_{MoC} \text{ then }, ev ::= \{Parameter\} \tag{4.12}$$

4.2.3.2.d Cometa *Behavior*

The behaviors of the $\mathcal{B}_{FSM}$ (4.13) and $\mathcal{B}_{Op}$ sets define common properties associated with all elements of $\mathcal{B}_{MoC}$. This means that the described behaviors have one or more roles associated with them; several parameters for the description of state variables; a referenced MoC domain to identify the type of MoC rules they implement. In Figure

4.5, *FiniteSateMachineBehavior* refers to elements of the set $\mathcal{B}_{FSM}$ and *OpaqueBehavior* refers to elements of the set $\mathcal{B}_{Op}$. Our focus is more on the description of the formalization of elements of type *FiniteSateMachineBehavior*:

$$\forall\ fsm \in \mathcal{B}_{FSM} \text{ then } \begin{cases} fsm ::= \{finalS : State\}; initS : State; \{State\}; \{Transition\}; \\ role; domain; \{Parameter\} \\ \{State\} \subseteq \mathcal{S}, \\ \{Transition\} \subseteq \mathcal{T}, \\ role\ \ : \mathcal{B}_{FSM} \rightarrow \{MoCRole\}, \\ mocdomain\ \ : \mathcal{B}_{FSM} \rightarrow \mathcal{D}_{MoC}, \\ \{MoCRole\} = \{Orchestrator, PortBehavior, ConnectorBehavior\} \end{cases} \tag{4.13}$$

The $< role >$ function is defined to associate the following roles (*SchedulingBehavior*, *PortBehavior*, *ConnectorBehavior*) to the different behaviors (e.g. *FiniteSateMachineBehavior*), making the elements they are related to explicit i.e. (*MoCComponent*, *MoCPort*, *MoCConnector*). One can associate several roles with the same execution control behavior if the mechanism which it describes is reusable in different contexts (e.g. scheduling, protocol). The function $< mocdomain >$ defines association to a specific MoC domain the MoC behavior belongs to.

Cometa defines event-based FSMs. The event-based FSMs are machines for which the variables or symbols that allow the changes of state are events from the system. As stated earlier, in the context of communicating concurrent entities, such events are requests such as *Send*, *Receive* or *Ack*.

With the abstracted concepts, one can describe e.g. Moore, or Mealey FSMs. The choice of one or the other machines to describe a control behavior is left to the discretion of the user. However, the concepts required to define both of them are present in the language. We will see the description of the FSM concepts in the following paragraphs.

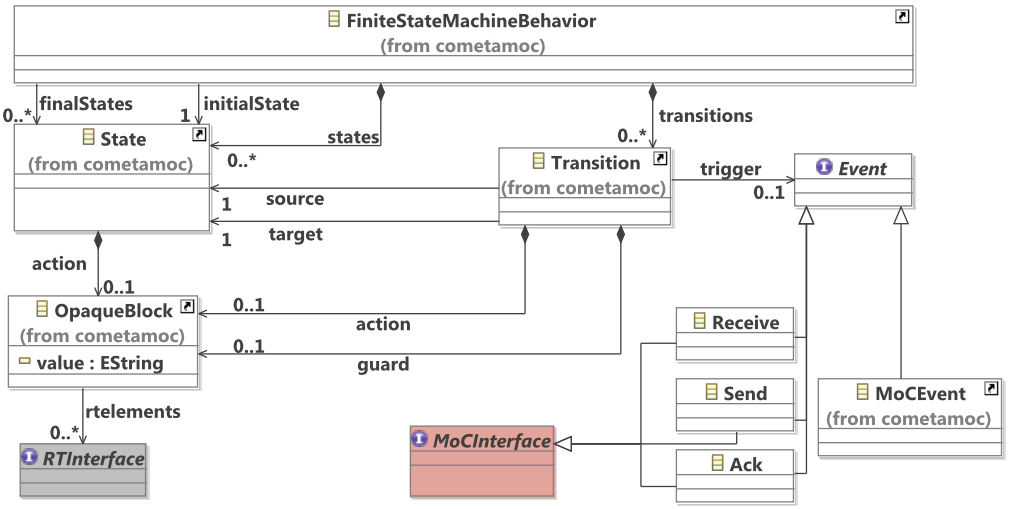

Figure 4.6: Excerpt of the Cometa FSM Behavior concern

The concept of *State* (4.14 and Figure 4.6) in Cometa is a description of a control state. A control state contributes in regulating incoming and outgoing requests of various concurrent entities. It allows the definition of the order in which the various requests must precede or succeed. This is done by describing in each state the authorized events generated from the application's requests and by defining the successors and predecessors states (of each state) that are part of the synchronization process and based on MoC execution logic.

The set of states of a control machine, and the set of transitions define the interaction patterns. An interaction pattern shows a possible way of communication and exchange between the entities of the system. In a Cometa FSM, the state changes can imply the execution of a sequence of instructions, either on a given state or on a given transition using the concept of $< action >$ (see Figure 4.6) that references a *Block* (*OpaqueBlock*. In our formalization, the $< action >$ function is represented differently for the *State* (i.e. $< action_S >$) and *Transition* (i.e. $< action_T >$). In the notation of $action_a$, $< a >$ stands for *State* or *Transition*. The referenced blocks define the sequences of instructions for internal and external Cometa communication. The internal communication is the communication between Cometa's control FSMs; in such case only the control events defined inside Cometa are considered. The external communication takes into account the events that are external to Cometa (e.g. requests from a concurrent entity). The description of the contents of *Block* is detailed in the Section 4.2.3.5.b. We formalize the *State* relations as follows:

$$\forall\ s \in \mathcal{S} \text{ then } \begin{cases} s ::= action_S \\ action_S\ : \mathcal{S} \rightarrow \{Block\} \end{cases} \tag{4.14}$$

At the most, one b:*Block* is associated to a s:*State*.

The concept of *Transition* (4.15) is used to describe the relationships between the various states of an FSM. They define relations of succession and precedence between the control states. The relations are implemented in Cometa using functions to denote the $< source >$ (i.e. predecessor) state and $< target >$ (i.e. successor) state for a given transition. There is one source and one target for each *Transition*.

Thus, starting from the set of transitions between control states, one can deduce all relations of type predecessor, successor between states. Transitions have $< trigger >$ functions that are of type *Event*. The occurrence of an event called the $< trigger >$ may involve state change during the control process if all additional conditions on the transition are verified. The conditions on transitions are referenced as the $< guard >$. A $< guard >$ is placed on the transition expression and its evaluation is mandatory for the state change.

- *Trigger*: This function is used to determine the type of event occurrences that may cause a change of state. In the context of Cometa, occurrences of events that are $< trigger >$ are *MoCInterface* (*Send*, *Receive*, *Ack*) and *MoCEvent*. The objective is to provide an order of processing requests by the synchronization rules.

- Guard: The $< guard >$ are functions to the expressions to evaluate. The expressions are additional conditions of transition from one state to another. In the context of the Cometa $< guard >$ references *Block*. The return of the evaluation of the *Block* content potentially authorizes the change of state.

We formalize *Transition* as follows:

$$\forall\ t \in \mathcal{T} \text{ then } \begin{cases} t ::= source; target; guard; trigger; action_T \\ source\ : \mathcal{T} \rightarrow \mathcal{S}, \\ target\ : \mathcal{T} \rightarrow \mathcal{S}, \\ guard\ : \mathcal{T} \rightarrow \{Block\}, \\ trigger\ : \mathcal{T} \rightarrow \{Event\}, \\ action_T\ : \mathcal{T} \rightarrow \{Block\} \end{cases} \tag{4.15}$$

Event is a set of communication events such as *MoCInterface*s (for external communication), *MoCEvent*s (for internal communication).

A *Block* in Cometa helps to define a sequence of instructions. The defined instructions allow communication between the control FSMs and the access to shared memory through the primitives (services) defined in *RTInterface*. A *Block* has a body attribute that is a string representing the sequence of instructions. In a sequence of instructions one can call services to gain access to the shared memory (e.g. add, remove on a FIFO); or one can call functions that convey *MoCEvent*, and *MoCInterface*.

The MoC communication strategies are specified within separate libraries of MoC-based behaviors. The behaviors are responsible for the arbitration of the communication between the components, and are represented as, e.g. communication protocol's behaviors, or *Scheduler*'s behaviors.

4.2.3.3 Data Description

The Cometa approach provides bases for the description of abstract data types which are categorized into 3 subsets: Primitives, Pre-defined and User-Defined.

The following formal definition is proposed: $\mathcal{L}_{Data}\ : \langle \mathcal{P}rim_{Data}, \mathcal{P}red_{Data}, \mathcal{U}sr_{Data} \rangle$

Primitive types refer to data types such as the Integer, Boolean, Real, etc.

The predefined types are domain specific, and allow the representation of types suitable for data flow semantics. For example, we abstract the concepts of *Array*, *Vector*, *MultiDimensionalArray*, or *Matrix* to capture specific information related to the size of the vectors, arrays, etc. This information is then exploitable by the execution control behaviors manipulating production and consumption rates.

Users also have the opportunity to describe their own data structure. The description of these new structures is provided by the abstraction of concepts for the definition of sequence of typed elements.

$\mathcal{P}rim_{Data}$ and $\mathcal{P}red_{Data}$ represent sets of abstract elements predefined in the metamodel of Cometa (see the above data type examples). While *UsrData* is based on the definition of concepts such as:

$$\forall\ d \in \mathcal{U}sr_{Data} \text{ then } \Big\{ d ::= \{DefinedType\} \tag{4.16}$$

DefinedType has a shaping (*Shaping*) to specify a sequence of elements. A *Shaping* can contain a typed set of *Element*s.

4.2.3.4 Relationships between Structure, Execution Control and Data

There are several inter-concern relationships. These relations allow the association of a topology with behaviors, behaviors to data structures, and so on. An overall semantic layer is thus composed of the combination of several concerns related by these relationships.

4.2.3.4.a Between Structure and Behavior

The structural elements (*MoCComponent*s, *MoCPort*s, *MoCConnector*s) are associated to execution control behaviors defined in the *MoCDomain*. Every structural entity is able to share a control *Behavior* with the other elements of the topology thanks to the defined $< bhv >$ function. For the purpose of formalization, we separate this function into 3 types of functions $< bhv1 >$, $< bhv2 >$, $< bhv3 >$ (4.17) on the structural elements mentioned above. In Figure 4.3, there is no separation between these types of functions which are represented by the same relationship *behaviorMap*.

$$\begin{cases} bhv ::= \{bhv1, bhv2, bhv3\} \\ bhv1 \ : \mathcal{M}oC_{Cp} \rightarrow \{Behavior\}.\ See formula\ 4.5, \\ bhv2 \ : \mathcal{M}oC_{Cn} \rightarrow \{Behavior\}.\ See formula\ 4.4, \\ bhv3 \ : \mathcal{M}oC_{P} \rightarrow \{Behavior\}.\ See formula\ 4.9 \end{cases} \tag{4.17}$$

Besides, as shown in Figure 4.3, attaching behaviors to structural elements is done through the concept of *StructureElement* for which we define a function (*behaviorMap*) to the concept of *Behavior*.

4.2.3.4.b Between Structure and Data

The relationship between the structural elements and the data is not direct. It is ensured by the use of the *Parameter* concept.

A *Parameter* (4.18) is an abstract concept of the metamodel that aims to capture and store variables and parameters. The elements of the Structure library (*MoCComponent*s, *MoCConnector*s, *MoCPort*s) and those of the MoC-based behavior library (*Behavior*), are related to concepts of the data library (*Type*) through the concept of *Parameter*. The following definition provides the formalization:

$$\forall\ p : Parameter \text{ then } \left\{ p ::= b : Block; t : Type \right. \tag{4.18}$$

A *Parameter* has a value (*Block*) and a type (*Type*). The value represents the data stored by the variable. The type represents the type of the stored value.

4.2.3.5 Description of the Cometa Interfaces to other DSML

As depicted in Figure 4.7, the Cometa models are layers receiving requests issued by their associated application blocks. An application block is interfaced with the layer

components via *MoCInterface*. A *MoCInterface* can trigger the execution control behavior defined in Cometa. The behaviors guarantee the consistent transmission of requests to avoid conflicts between the requests. For example, *Send* and *Receive* requests are taken into account only when all the synchronization requirements (defined by the MoC) are met.

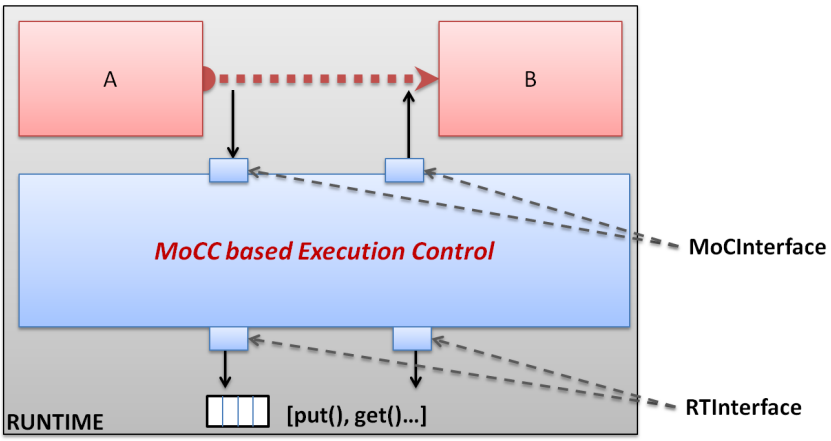

Figure 4.7: Highlighting the Interfaces

The Cometa behaviors allow the *Send*, *Receive* or *Ack* requests to gain access to the shared memories using services of the runtime interfaces *RTInterface* of Cometa. The *RTInterface* are services that highlight the functions to be implemented in order to manipulate a shared memory (e.g. add, remove). The *RTInterface* are also used to store the component requests in an event queue. The communication primitives are the set of implemented services understood by a given execution engine to be the requests between distinct entities. They help to establish communication between entities or provide access (read/write) to shared/distributed memories. There are several mechanisms of communication. However, Message Passing is the common standard for the communication between concurrent entities. The Message Passing Interface (MPI) [67] standard defines mainly three primitives for communication *Send*, *Receive*, *Send_Receive*. For example, the functional programming language Scala defines an actor based model; each actor has a "mailbox" in which the data conveyed by the requests (*Send*/*Receive*) are stored.

The interfaces that we address in this section have a dual purpose. They serve as a communication contract between the layer of MoC-FSM and the concurrent entities; they also serve as access to the memories shared between the different communicating entities. We define two types of interfaces, *MoCInterface* to meet the first objective and *RTInterface* to meet the latter objective.

4.2.3.5.a Specification of *MoCInterface*

The concept of *MoCInterface* captures the subset of events that represent requests produced between the MoC layer and the concurrent entities. We have restricted this subset to three types of exchanges (i.e. interfaces): *Send* and *Receive* requests (from entities models to the MoC models), and *Ack* (from the MoC models to entity models). On the one hand, the occurrence of events *SendEvent*, *ReceiveEvent* are used to trigger state changes in the control mechanism. On the other hand, the *Ack* release the components to continue their computation.

- *Send*: This concept allows the sending events to be described. The sending events are the result of data sending requests emitted by an entity behavior. These events can be parameterized. The parameters are used to specify the receiver of the request, the communication connector, or the data to be transmitted, etc.

- *Receive*: This concept allows the reception events to be described. The reception events are the result of the reception requests issued by an entity behavior. These events can be parameterized. The parameters are used to specify the communication connector.

- *Ack*: The *Ack* events are sent back to the behavioral entity so they can continue their execution. The *Ack* are needed in all running control scenarios with Cometa. They reify the concept of permission to run. The FSM captures the control behavior and governs the sending of *Ack* according to the MoC rules. For example, depending on MoC rules, the sending of *Ack* can be unconditional on reception of a *SendEvent* or *ReceiveEvent*; elsewhere it can be done with some delay when a request must be blocked. These events can be parameterized. The parameters are used to specify the receiver of the request, the communication connector, or the data to be transmitted, etc.

4.2.3.5.b Specification of *RTInterface*

The concept of *RTInterface* captures the events (*MoCEvent*s), as well as services and parameters that are used within the control behavior to interface with the communication media (shared memory such as FIFO and LIFO). Elsewhere, the services are used in the same way to store new events in the event queues that are operated by the schedulers.

The services are captured as annotations. Their implementation is provided as a library of functions external to the Cometa DSML. The captured services are used in the *Block*s of the control behaviors as a means to call their real implementation in the runtime.

- *Service*: The concept of service is used in the context of Cometa to abstract functions interfacing with the runtime, as well as to abstract functions for communication between machines of control. In our experiments, we have identified three types of primitives for the above intentions. The primitive *sendOfEvent* is used for sending *MoCEvent*s between control machines; the primitives (e.g. add, remove and check) are used to operate on queues of data or queues of events.

- *Queue*: The analysis of communicating behavioral entities must take into account the sizes of the shared memories used because in reality such sizes cannot be infinite. The concept of *Queue* is a concept for the virtual representation of the shared memories such as FIFO, LIFO. The purpose of this virtualization is to highlight their sizes and the required services to operate on the Queues.

4.2.4 Operational Semantics: FSM-Based Control

In this section, we describe the operational semantics of the semantic layer models described from Cometa using a Labelled Transition System ($\mathcal{LTS}$). The $\mathcal{LTS}$ helps to study the feasibility of the behaviors represented in the form of states and transitions.

The description of the operational semantics requires several preliminary definition steps, from the firing of a request by an application block to its definitive processing via the semantic layers.

A transition system consists of a set of possible states and a set of transitions involving changes of states. In our particular case, states and transitions in the $\mathcal{LTS}$ are provided by the control FSMs which are described by Cometa. The operational semantics of the MoC-based control layers is defined through a projection of Cometa semantic layers on an $\mathcal{LTS}$.

In such an $\mathcal{LTS}$, each semantic layer represents a configuration which is given by all the states of the semantic layer. In addition to the configurations, the transition relations describe rules making the possible state changes explicit i.e. the transitions to move from a *source* to a *target* configuration. Each of the transition rules clarifies the necessary conditions for state changes. State changes are the consequences of external events from the application blocks, or internal events produced by actions in the states and transitions of the MoC-based FSMs.

A Structural Operational Semantics (SOS) [164] is defined with the $\mathcal{LTS}$ which allows the definition of the transition rules on the $\mathcal{LTS}$. SOS provide a framework for the description of the operational semantics and complement the $\mathcal{LTS}$.

The SOS based transition rules are of the form $\frac{premises}{conclusion}(condition)$ [2]. In [164], the validity of the premises of a transition rule (under certain conditions) induces the validity of the conclusion of the transition rule. In this part, we will define some transition rules in the form of SOS associated with our $\mathcal{LTS}$, thus providing the operational semantics of Cometa.

4.2.4.1 Operation Semantics of the *Block* requests

By definition, the requests issued by the application *Block* are translated into noticeable events for the semantic layers and execution control behaviors.

The *Block* concept is viewed as a sequence of instructions accomplishing internal processing (i.e. computations), and expressing external requests to other elements. When the *Block*s are used within the FSM, the requests produce specific inter-FSM communication events *MoCEvent*s. When blocks are defined at the level of the *BasicComponent*, requests produce noticeable *MoCInterface* events. Formally, if we consider:

- $\mathcal{R}$: the set of requests that can be issued from *Block*, in Cometa we limit these types of requests to {*Send*, *Receive*}. Among the requests from different blocks (application, related to the MoC-based control machines), we are interested in send and receive requests. The computations made on these data are not considered.

- ϵ_{Event} : the set of events generated from queries above. In this set we have two categories of events. The first category includes the events that are generated from the requests of the application blocks. These events are then associated with those that are defined in *MoCInterface*. Besides, as previously described, the states and transitions of control machines are potentially associated to action blocks respectively defined by the $action_S$ and $action_T$ functions. The events generated from the requests in these blocks correspond to events of *MoCEvent*.

This definition provides the following rule:

$\epsilon_{Event} \subseteq MoCInterface \bigcup MoCEvent$

In the rules below, we consider that executing the requests from $\mathcal{R}$ causes the creation of noticeable events for the semantic layers of Cometa (external communication). As a result, ϕ_{Block} is the function associating a *Block*'s requests to the events (*MoCInterface*) that are generated by the requests. By definition, we restrict the types of events *MoCInterface* to 3 subtypes <*Send*, *Receive*, *Ack*>.

Since the states and transitions are associated with blocks by functions $action_T$ and $action_S$, the second rule allows the definition for each state and transition, the potential events which are produced when their internal actions are executed. The selected premises predict that if there are action blocks defined by ($action_S$ and $action_T$) for the states and transitions, there exists a θ_{Block} function that associates each request in these blocks to the events they generate. We note in this context that the events are of type *MoCEvent* (internal communication).

$$\begin{cases} \dfrac{}{\mathcal{R} \xrightarrow{\phi_{Block}} \epsilon_{Event}} \text{ Rule. ⓪} \\ \dfrac{action_S : \mathcal{S} \rightarrow \{Block\} \text{ or } action_T : \mathcal{T} \rightarrow \{Block\}}{\mathcal{R} \xrightarrow{\theta_{Block}} \epsilon_{Event}} \text{ Rule.①} \end{cases}$$

In summary of the above definitions, we assume that each request issued by a *Block* causes the creation of *MoCInterface* events or *MoCEvent* exploitable by the semantic layers.

4.2.4.2 Labelled Transition System for Cometa

The Cometa models aim at describing operational execution control layers based on the use of the FSMs and the theory of computation (MoC). The FSMs induce the operational aspect of the execution semantics and the MoC specifies the formal execution rules.

The FSMs are based on using events as a *Trigger* for the state changes. The operational semantics of these machines can be described in the form of a transition system, formally presenting the links between the different states. In fact, the $\mathcal{LTS}$ allow the Cometa models to be described in terms of finite-state space with the links triggering state changes. In the Cometa $\mathcal{LTS}$, states and transitions of the $\mathcal{LTS}$ assets are equivalent to those from the Cometa layers. The projection phase is described below by a mechanism to bring the components of a Cometa model to a set of state and transitions labeled by the events.

We formalize the $\mathcal{LTS}$ corresponding to the semantic layers in the following way: $\langle \mathcal{S}_T, \mathcal{I}, \mathcal{R}_T, \mathcal{L}, \mathcal{E}_t \rangle$.

- $\mathcal{S}_T$ is the set of states of a structural model. The states of this model come from the FSM that are associated with the various entities within the topology description i.e. (*MoCComponent*, *MoCConnector*, *MoCPort*). The following formal definitions represent the projection of the states according to their position in a structure of Cometa components:

$$\forall s \in \mathcal{S}_T \text{ then } \begin{cases} s ::= [StructureContainer].[\mathcal{X}]^*.[FiniteStateMachineBehavior].s, \\ \mathcal{X} \equiv Part, \text{ or } \mathcal{X} \equiv MoCComponent, \text{ or } \mathcal{X} \equiv MoCPort, \text{ or} \\ \mathcal{X} \equiv MoCConnector, \\ \mathcal{S}_T \subseteq \mathcal{S} \end{cases} \tag{4.19}$$

Each state in $\mathcal{S}_T$ is identified by the traces of its containing elements i.e. the traces from the lower hierarchical level container to the top level container. The containers are separated by brackets

.

The [*StructureContainer*] specifies the top level container; if the state belongs to a *Part*, *MoCComponent* or *MoCConnector* inside the top container, then [*StructureContainer*].[$\mathcal{X}$].[*FiniteStateMachineBehavior*].[s:State] identifies the state (with $\mathcal{X}$ = *MoCComponent* or *MoCConnector*, etc). For several hierarchical levels, the same notation is used with:

[*StructureContainer*].[$\mathcal{X}1$].[$\mathcal{X}2$]...[$\mathcal{X}n$].[*FiniteStateMachineBehavior*].[s:State].

This latter representation is simplified with the notation [*StructureContainer*].[$\mathcal{X}$]*. [*FSMBehaviorContainer*].[s:State].

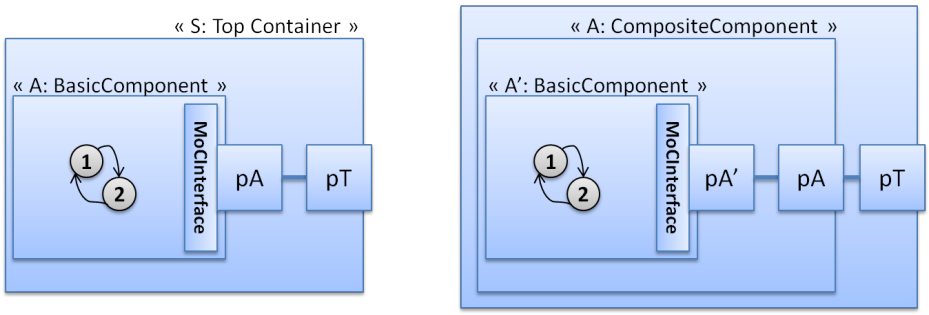

Figure 4.8: Simple example of Layer representation

In the Figure 4.8-left and the Figure 4.8-right, the States 1, 2 of the structural model are identified by (S.A.1, S.A.2). For the left example; states are identified by (S.A.A'.1, S.A.A'.2) for the right example.

- $\mathcal{I}$ represents the set of initial states and adheres to the same description rules as S_T States.
- $\mathcal{R}_T \subseteq \mathcal{S}_T \times \mathcal{S}_T \times \mathcal{L}$ and describes the set of transition rules to move from a given state to its possible successor states. The transitions in this specific case are conditioned by a label from the set of events $\mathcal{L}$ defined below. A transition of $\mathcal{R}$ is then given by the coupling of (current State, successor State, Label). The labels for the event-based FSM are events capable of causing the activation of a transition. The FSM being deterministic, one transition condition cannot help reach two different successors. Each transition reaches only one state with a single given condition.

- $\mathcal{L}$ represents the set of labels or events that may cause state changes. These already identified events are part of the sets of events *MoCInterfaces*, or *MoCEvents*. In the rest of this chapter, $\mathcal{L}$ and ϵ_{Event} are used in an interchangeable way. Therefore, we propose the following definition:

$$\forall\ ev \in \mathcal{L} \text{ then } \Big\{ ev \in \epsilon_{Event} \tag{4.20}$$

A transition can be described as the coupling between the *source* state, the *target* state and the event producing the state change. In other words, when a transition connecting two states is sensitive to an event causing a change of state, the transition is possible when the event occurs. Based on the fact that the events belong to ϵ_{Event}, we can define a function γ combining the occurrences of events (labels) to the transitions they trigger:

$\gamma\ : \mathcal{L} \rightarrow \mathcal{R}_T$

The $\mathcal{LTS}$ describes the system evolution at every stage of execution, thus, explicitly giving the overall status of the system. Given a system configuration (application blocks + Semantic Layer), one can identify the set of states, transitions, events (labels) between transitions.

The labeling functions are defined at the same level as the various transition rules. The FMSs are bound together by the communication events that occur between them. In this case, an event occurring in a state of an FSM_A can trigger a transition (change of State) in an FSM_B which is sensitive to this event.

Based on the rules defined for the block's computations, we can generalize the operational semantics of the $\mathcal{LTS}$ by identifying the conditions that enable the generation of events capable of stimulating the semantic layer's control machines. The following formula specifies these conditions.

$$\begin{cases} \dfrac{(\mathcal{S}_T \xrightarrow{action_S} \{Block\}\ ,\ \mathcal{R}_T \xrightarrow{\theta_{Block}} \epsilon_{Event})}{(\mathcal{S}_T \times \epsilon_{Event}) \xrightarrow{\theta_{ST}} \epsilon_{Event}} \ | \\ \dfrac{(\mathcal{T} \xrightarrow{action_T} \{Block\}\ ,\ \mathcal{R}_T \xrightarrow{\theta_{Block}} \epsilon_{Event})}{(\mathcal{T} \times \epsilon_{Event}) \xrightarrow{\theta_T} \epsilon_{Event}} \text{ Rule.②} \\ \\ \dfrac{(\mathcal{R}_T \xrightarrow{\phi_{Block}} \epsilon_{Event}\ ,\ (\mathcal{S}_T \times \epsilon_{Event}) \xrightarrow{\theta_{ST}} \epsilon_{Event})}{(\mathcal{R}_T \xrightarrow{\phi_{Block}} \epsilon_{Event}) \xrightarrow{\theta_{S_T}} \epsilon_{Event}} \ | \\ \dfrac{(\mathcal{R}_T \xrightarrow{\phi_{Block}} \epsilon_{Event}\ ,\ (\mathcal{T} \times \epsilon_{Event}) \xrightarrow{\theta_T} \epsilon_{Event})}{(\mathcal{R}_T \xrightarrow{\phi_{Block}} \epsilon_{Event}) \xrightarrow{\theta_T} \epsilon_{Event}} \text{ Rule.③} \end{cases}$$

There are several rule patterns to make the events explicit that are generated from the states and transitions. In order to facilitate the writing and reading of such sequences of rules, we regroup them consistently as described by the first rule above. The objective of this rule is to present the sequence of premises which allows to define, for each state or transition, the events that are generated during the execution of their action blocks.

Firstly, when an event of ϵ_{Event} allows a transition between states to be triggered, one can apply the premises of the first formula to determine how the actions of the *target* state generate new events. The core idea argues that at each entrance of a *target* state (after transition), there is an $action_S$ (and or an $action_T$) function that determines the block to run; there is also a θ_{Block} function that associates the block's request sequence to the events that are generated at their execution. Consequently, an input event that triggers state change, potentially, further induces the generation of new events which is simply written in the bottom part of the formula by $(\mathcal{S}_T \times \epsilon_{Event}) \xrightarrow{\theta_{ST}} \epsilon_{Event}$ (or $(\mathcal{T} \times \epsilon_{Event}) \xrightarrow{\theta_T} \epsilon_{Event}$ for $action_T$), etc.

The second formula's premises are: The execution of requests by ϕ is capable of producing stimulation events of the states; The second condition predicts that for a given state of an FSM, the event leading to enter such state potentially causes new requests within the state and therefore the production of new events that will be used for communication between the MoC-Based FSMs (related to Rule ②). Rule ② is a direct consequence of the combination of Rule ⓪ and ①. Similarly, Rule ③ is induced by the rules ⓪ and ②.

Under these conditions, it can be concluded that block requests cause the generation of control events between FSMs which enables the control of the requests from application blocks. Consequently, the state changes are made where transitions are possible. With the definitions provided, when a transition is possible, one can associate *source* and *target* states, as well as the event causing the state change to define the transition. To simplify the identification of current states, we introduce an alternative representation : $next_{FSM} : \mathcal{S}_T \times \epsilon_{Event} \rightarrow \mathcal{S}_T$. Consequently, the following formula defines such semantics:

$$\forall\ (ev{\in}\mathcal{L};\ s, s'{\in}\mathcal{S}_T) \text{ then } \left\{ \frac{\mathcal{L} \xrightarrow{\gamma} \mathcal{R}_T\ ,\ s \xrightarrow{\mathrm{t}} s'\ ,\ \mathrm{t}\ : (s, s', ev)}{s \times ev \xrightarrow{next_{FSM}} s'} \right. \text{Rule.④}$$

The formula above can be interpreted as follows: if we have premises on the occurrence of events that can trigger transitions by γ, and the transition rules specifying the *source* and *target* states $\mathcal{S}_T \times \mathcal{S}_T \times \mathcal{L}$, as a conclusion we can define a transition from the *source* state s to the *target* s' by the transition $t : (s, s', ev)$ with $ev \in \mathcal{L}$.

The sequence of requests in a system creates as well the sequence of events which has the consequence of the triggering of transitions when possible. The succession of transitions between states is formalized in the following manner:

$$\left\{ \frac{s \xrightarrow{\mathrm{t}} s'\ ,\ s' \xrightarrow{t'} s''}{(s \xrightarrow{\mathrm{t}} s') \xrightarrow{t'} s''} \exists(t, t') \text{ such that t and t' } {\in}\mathcal{R}_T \right. \qquad \text{Rule.⑤}$$

If a transition t allows state changes between s' and s, and a second transition t' also allows state change between s' and s'', then we can conclude that the succession of t and t' induces state changes from s to s''. Generalizing to n, we obtain the operational semantics below when the transitions are for the same FSM (Rule 6), or when we have different communicating FSMs (Rule 7).

$$\begin{cases} \dfrac{s_0 \xrightarrow{t_0} s_1, \ldots, s_{n-1} \xrightarrow{t_{n-1}} s_n}{\Big(\ldots\Big(\ldots\Big(\Big(s_0 \xrightarrow{t_0} s_1\Big) \xrightarrow{t_1} s_2\Big)\ldots\Big)\ldots\Big) \xrightarrow{t_{n-1}} s_n} \\ \exists(t_0, \ldots, t_n) \text{ such that } t_0, \ldots, t_n \in \mathcal{R}_T \text{ Rule.⑥} \\ \\ \dfrac{(s_0 \xrightarrow{t_0:(s_0,s_1,ev_0)} s_1, ev_0 \xrightarrow{\theta_{s_1}} ev_1), \ldots, (s_{n-1} \xrightarrow{t_{n-1}:(s_{n-1},s_n,ev_{n-1})} s_n, ev_{n-1} \xrightarrow{\theta_{s_n}} ev_n)}{\Big(\ldots\Big(\ldots\Big(\Big(s_0 \xrightarrow{t_0} s_1, ev_0 \xrightarrow{\theta_{s_1}} ev_1\Big) \xrightarrow{t_1} s_2, ev_1 \xrightarrow{\theta_{s_2}} ev_2\Big)\ldots\Big)\ldots\Big) \xrightarrow{t_{n-1}} s_n, ev_{n-1} \xrightarrow{\theta_{s_n}} ev_n} \\ (t_0 : (s_0, s_1, ev_0), t_1 : (s_1, s_2, ev_1), \ldots, t_{n-1} : (s_{n-1}, s_n, ev_{n-1})) \text{ Rule.⑦} \end{cases}$$

The semantic definition allows the current state of each FSM to be traced after the interception of an event through its premises. The value of the current state is derived from the transition rules defined by the $\mathcal{LTS}$.

The above definition allows the current state of each FSM to be traced after the interception of an event. The value of the current state is derived from the transition rules defined by the $\mathcal{LTS}$.

4.3 Execution Control Mechanisms description

There are several ways to define execution control mechanisms and they can also be managed by different types of components. A particular entity to implement execution control rules is the *Scheduler*. The global schedulers hold all the rules to trigger the execution of components, thus controlling their synchronization. Besides, communication entities can also be used to control the executions. They allow the various components to synchronize based on their properties. One can imagine cases mixing the use of schedulers and the communication mechanisms for execution control. In this case, the elements of communication are seen as components by the schedulers. Based on the above information, we can classify these mechanisms as follows: communication-based (asynchronous, synchronous, timed, untimed); or scheduler-based (also asynchronous, synchronous, timed, and untimed).

4.3.1 Scheduling in Cometa

In the HW/SW co-design domain, scheduling is an important aspect given the parallel nature of platforms on which the different components and tasks are mapped. This aspect addresses the management of execution tasks and the communication transactions. Such activities allow the control of the execution of the individual modules making up a system.

Depending on the level of abstraction and depending on the design activity performed, a type of scheduling can be selected instead of another. For example, centralized scheduling is usually recommended for analysis and simulation activities. However, for parallel architectures, the use of centralized scheduling does not always benefit (in terms of performance) from the advantages induced by parallel architectures. In this case, distributed scheduling is favored.

For executable model exchanges between tools, it is necessary to address the description of the scheduler when the semantics are heterogeneous, or when scheduling

formalisms require different underlying semantics (i.e. different MoCs) in each tool.

In this context, the MoCs define how a set of interconnected components should interact, which is implemented in the *Scheduler*s. The operating of a MoC is often provided by an operational semantics based on the MoC rules.

With Cometa, all the *Scheduler*s are defined in the form of FSM interconnected via a structure that we call "layer". The layer is able to interact with the external computations to monitor their execution. Both centralized and distributed scheduling is described in Cometa. The FSMs of the *Scheduler*s are the operational versions of the MoC underlying the *Scheduler*s. The global MoC of the system is provided by the composition of all the MoC-based FSMs.

In the following sections, we look at both types of scheduling which are the centralized scheduling and the distributed scheduling.

4.3.1.1 Centralized scheduling in Cometa

The centralization of the *Scheduler* makes sense when there is a significant amount of data exchanged and very frequently requested by the modules composing the system. In this case, a consistent data update is essential for the proper functioning of the system. Indeed, such scheduling controls the overall execution order of all computations on data, the communication transactions, as well as the updates on data. Unfortunately, for large systems, they easily become complex compared to the distributed scheduling. For example, the management of the set of requests coming from multiple and heterogeneous processors can be a bottleneck reducing performance.

As shown in Figure 4.9, the centralized *Scheduler* implementation in Cometa is based on: the definition of the execution control (FSM) representing the operational semantics of the MoC rules; the definition of synchronization events between the modules issuing requests; the definition of the interfaces between the *Scheduler* and the external modules (or computations).

The concepts required for the description of centralized *Scheduler* are available through: the concepts of *StructureContainer* and *CompositeComponent* (e.g. *S:Top Container* in Figure 4.9) that allow the retrieval of the number of sub-components (e.g. *MoCPort A*, *A: BasicComponent*, *MoCConnector*) on its hierarchical level, as well as supporting the execution control FSM; the concept of *MoCPort* defines the interfacing points and the synchronization events are provided by the *MoCInterface*; finally, the link between components is determined through the *MoCConnector*.

The MoC implemented in a generic *Scheduler* must be able to keep the same operational mode regardless of the number of sub-components that are at stake. In this context, the difficulty comes from the dynamic increase in the number of components (potentially heterogeneous) in the same hierarchical level. The heterogeneity of the MoCs comes with the heterogeneity of the components.

It is difficult to combine several MoCs at the same level of abstraction using centralized scheduling. This is due to the fact that the complexity and size of the *Scheduler* increasingly grows depending on the number of MoCs in the hierarchical level. Additionally, the *Scheduler* should be able to separate the different semantic groups, provide their implementation, and ensure the consistency of their composition. Besides, to communicate with the new components, the *Scheduler* must include information for interfacing the heterogeneous new components. Such a generic *Scheduler* is hard to specify and implement considering the MoC constraints. Therefore, the centralized *Scheduler* usually addresses only a single MoC semantics by hierarchical level i.e. all

components of this level are managed by the same MoC; which allows a glimpse of the generic *Scheduler* implementation based on a MoC.

Under these conditions, the heterogeneous hierarchical levels provide implicit MoC-based execution semantics' adaptations from one level to another. For instance, a *Scheduler* from a higher hierarchical level has a global vision of the requests at the borders in terms of I/O. Then, such a *Scheduler* receives all requests from modules and decides, according to its implemented MoC, those that have priority and must be running. The *Scheduler* is then set as a Cometa component and has to be a container for a set of sub-components on the same hierarchical level.

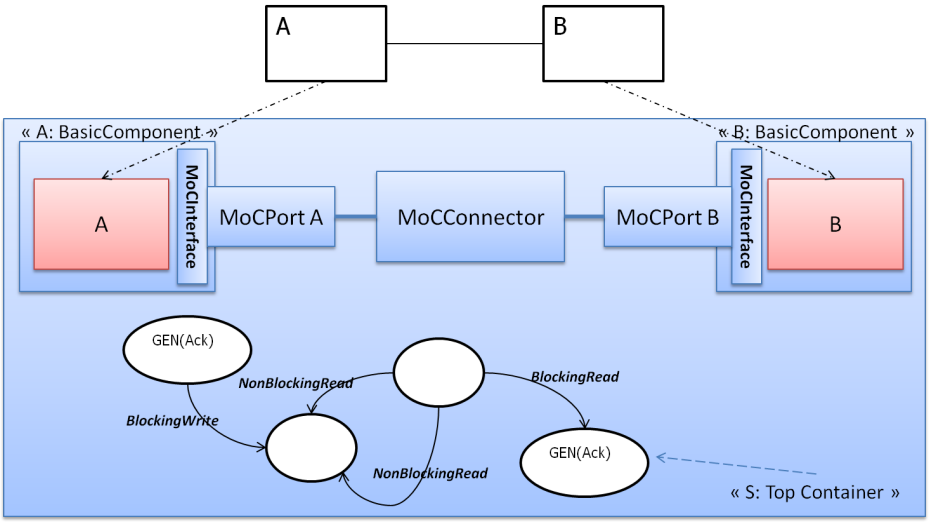

Figure 4.9: Example of container (*Top*) with an MoC-based FSM for centralized Scheduling

On this basis, and on the formal basis of the MoC rules, it is possible to determine a *Scheduler* associated with e.g. the *StructureContainer* or *CompositeComponent* to orchestrate the different components.

4.3.1.2 Distributed Scheduling in Cometa

The distributed scheduling is another execution control mechanism realizable in Cometa. This solution stands out from the centralized scheduling by its methods of implementation. Indeed, several local *Scheduler*s are put together to give scheduling decisions on the set of components they are related to.

One of the main advantages of the approach is the possibility to represent heterogeneous execution control mechanisms at the same hierarchical level and further, each local *Scheduler* does not need to known its other unrelated components. These two points are fundamental differences with the centralized scheduling.

The scalability of the system is improved since the addition of new components does not imply considerable alteration of the local *Scheduler* more generic than the centralized *Scheduler*s. The distributed scheduling also improves performance with the exploitation of parallelism induced by multiprocessors. For a parallel system, the execution control is provided through the description of intra-module and inter-module synchronization policies (when mapped on different processors). In Cometa, the distributed execution control mechanisms are defined locally by any structural element capable of supporting behavior. The structural concepts participate in the synchro-

nization of tasks and modules based on the resources at their disposal and their synchronization links. In this context, the main challenge is the coordination of the distributed *Scheduler*. Cometa allows the definition of local synchronization mechanisms placed on Cometa layer's (communication links and components) as shown in Figure 4.10. An SDF control behavior can, for example, be dispatched on several concepts i.e. *MoCPort*, *MoCConnector* and *MoCComponent*. The concurrent computation requests are received by the ports potentially supporting part of the control mechanism and the "port-connector-port" link constitutes synchronization between components incorporating synchronization semantics. Each interconnection may also define a different MoC-Based semantics (heterogeneity).

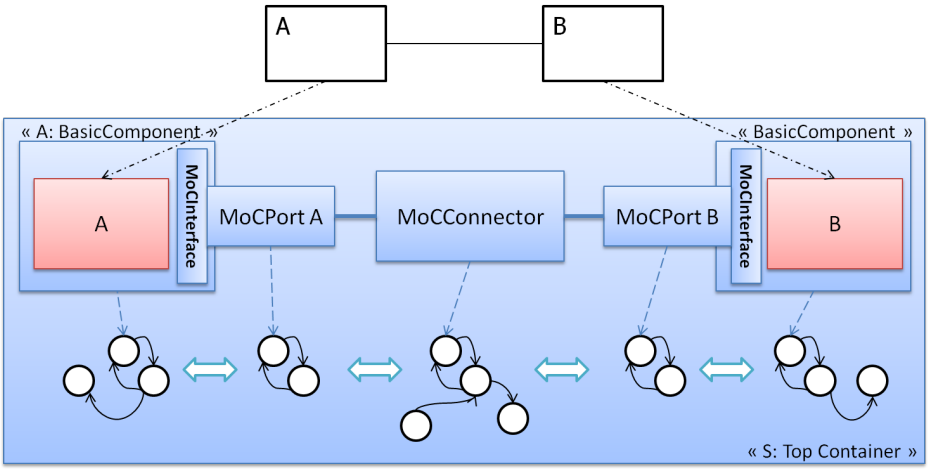

Figure 4.10: Example of container (*Top*) with MoC-based FSM for distributed Scheduling

The implementation of the different MoC-based control behaviors is provided in the form of FSMs interfacing with the application blocks via *MoCInterface* events and internal control events *MoCEvent* for FSM communication.

Control behaviors are generic and easily reusable. Indeed, these behaviors are independent of the number of components and reflect synchronization mechanisms that can interface with the application computations in some points. A control layer is then used to assemble them in a certain way to form the desired overall control semantics. We will see an example of use of this mechanism in Section 4.3.3.

4.3.2 Methodology for Applying Semantic Layers

The Cometa approach is designed to provide semantic enrichments and adaptation layers for models in an MDE context. The semantic layers integrate MoC-based operational semantics defined in the DSML. In this section, we present the steps prior to the description of semantic layers.

4.3.2.1 Application of semantic layers: from *MoC Aware* to *MoC Unaware* tools

The exchange of executable models between tools in a heterogeneous design flow is accompanied by the description of semantic adaptation layers. Semantic layers allow the reification of the execution properties that are specific to the models to be analyzed. The process of using Cometa's semantic adaptation models requires the description of

the topologies giving the structure of SLs. The SLs are then refined with the specification of the involved*MoCDomain* that make the targeted semantic types explicit for the model under specification. The functions are sufficient in some case to define the executability if there are tools capable of interpreting them and provide their execution semantics implementation.

As shown in Figure 4.12, the initial model has three sub-modules (A, B, C) that can be exchanged with three different environments that already implement the MoCs (e.g. ForSyDe, Ptolemy). The first environment refers to a tool that implements a homogeneous MoC, the second and third environments are tools describing heterogeneous MoCs.

In some cases, adding operational semantics to the source model induces more complex interpretation issues for the target environment. This is the case when the target tools already incorporate the required execution semantics. Most heterogeneous simulation tools are part of these categories that implement heterogeneous MoC semantics. For instance, Ptolemy [26], ForSyDe [174], HetSC [78] are environments already describing heterogeneous MoC libraries for analysis and model simulation. These tools define the *Scheduler*s and runtime engines that are able to process the application blocks according to a certain set of MoC semantics.

When these tools are integrated in a design flow, the use of Cometa is limited to the definition of references on the *BasicComponent* to designate the *Scheduler* or MoC to put in place in the target environments in order to effectively simulate the models. Thus, the references are carried by the semantic layers.

For the transformation phases, the references refer to scheduler types in the target environment to accommodate each application block as shown in Figure 4.11.

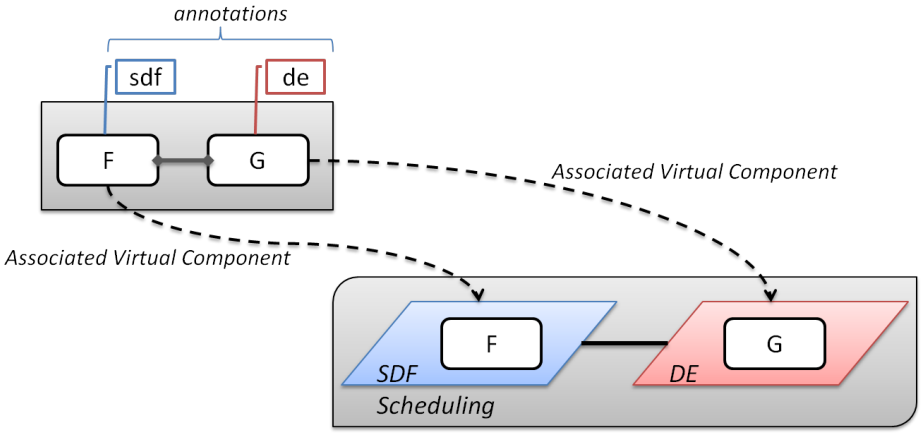

Figure 4.11: Application Block mapping into Semantic Layers

For these exchanges, the SLs explicitly describe references to the MoCs that are implemented in these tools, while providing a structure that preserves the topology of the initial modules. For the tools with homogeneous MoC (i.e. *Single-MoC* tools), the SL offers a topology with *Single-MoC Reference* for the adaptation. In the case of tools with heterogeneous MoC, the SL offers a topology with *Multi-MoC Reference* for the adaptation. The Libraries of Cometa executable behaviors are not necessarily used as the targeted tools provide their own implementation of the required MoCs. Tools such as [26] or [174] provide these implementations.

When the references are not sufficient, the execution control libraries and static properties are used to complete the models.

For the *MoC Unaware* tools, the SL not only give references to the required MoC, but most importantly they include execution control behaviors to make executable models with appropriate MoC semantics. The MoC-based control FSMs are placed on the various components that make up the SL. In this case, as shown in Figure 4.12, we provide the MoC implementation on an SL that targets a *MoC Unaware* tool. An example of implementation is shown in Section 4.3.3.

The top-down aspect means that depending on the targeted tools, and starting from an SL topology referencing the MoCs, we potentially can reach SL topology taking into account MoC implementations in the form of FSMs.

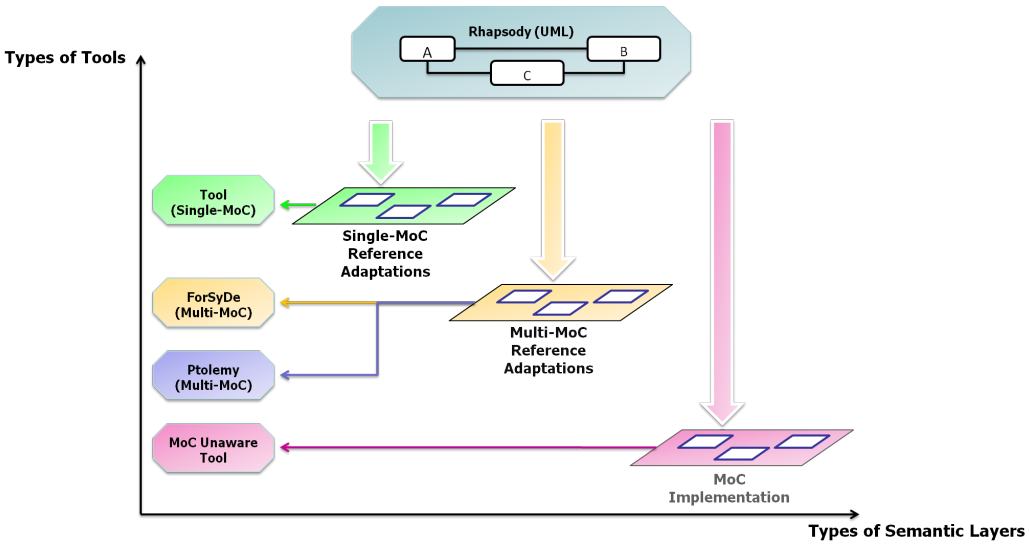

Figure 4.12: Positioning Semantic Layers between Tools in a Design Flow

However, the feasibility of the model's semantic adaptations between tools is strongly dependent on the compliance of the expressed MoCs by models and tools. To reach a good compromise for adaptation, it is necessary to define the basis for the study of the compliance of MoCs. Such a basis must be defined in a way that can easily be used for implementing adaptation properties. This is the goal of the next section. This description will not only help ensure the consistency of the interconnection of tools; but also identify compliant semantic properties for which adaptations are possible.

4.3.2.2 Rules to Identify MoC relations and Compliance

In this section, we propose techniques to characterize semantic interoperability using the different approaches for MoC classification and also the approaches for semantic domain definition. The semantic domain as defined by D. Harel *et al.* [72] is reused in a formal context with the MoC theory.

We define a new term *MoC-Based Semantic Domain* as the MoC domain on which the syntax of a given language has its execution formally defined. Consequently, the models produced become executable and follow the execution rules induced by the MoC. Before any further argumentation, we introduce some definitions.

Within a design flow, each interconnected tool uses a language L_{Tool} to describe models. For a given language L_{Tool}, one defines semantic mappings M to well-defined semantic domains. More particularly, mappings can be directed to so-called *MoC-Based*

semantic domains $MBSD_{MoC}$ to specify the models' execution rules. The mapping relation is denoted by $M : L_{Tool} \rightarrow MBSD_{MoC}$.

The relations between the $MBSD_{MoC}$ allows feasible model exchanges to be exhibited that emphasize semantics and behavior preservation. These relations are provided by the classification of MoC as defined in [176]. The classification is based on a description of the properties underlying each MoC and their degree of expressiveness. According to A. Jantsch [87], the main axes to characterize MoC properties are time, communication, behavior and data. Therefore, an $MBSD_{MoC}$ is defined by the tuple $\langle D_{MoC}, B_{MoC}, C_{MoC}, T_{MoC} \rangle$ where: D_{MoC} characterizes the data types specific to the MoC domain; B_{MoC} represents the underlying behaviors induced by the MoC rules; C_{MoC} represents how the communication is expressed in the MoC; T_{MoC} represents the way in which the time is expressed.

When $MBSD_{MoC}$ are compliant, it is possible to define a transformation T on the subset of compliant properties (tuples) to provide their translation. For instance, a transformation can be $T : D_{MoC1} \rightarrow D_{MoC2}$. Based on this, we can study the relationship between languages and the $MBSD_{MoC}$. In Figure 4.13, we depict the four main rules of relations.

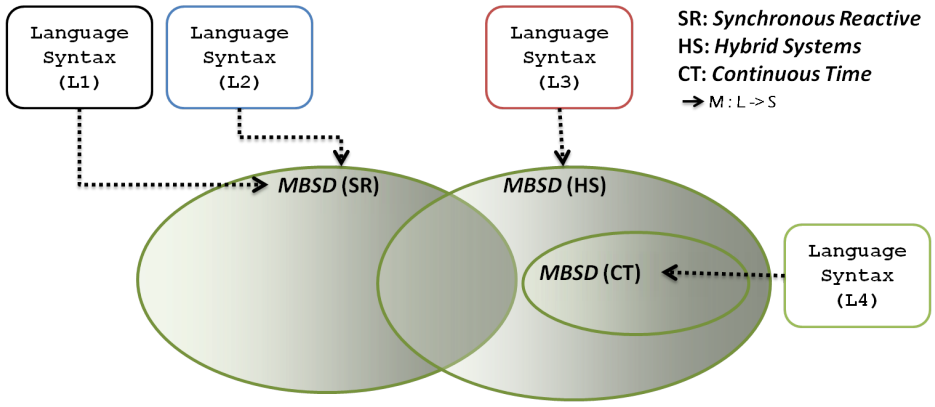

Figure 4.13: Language and MoC-Based semantic domains

- In the first rule, the language's syntaxes are mapped to the same semantic domain; e.g. L_1 and L_2 have their mapping to the same $MBSD_{SR}$. Here, even if the syntactic representations are different, there is a clear and common definition of the MoC domain elements where each syntactic element of L_1 and L_2 is mapped. The explicit definition of semantic mappings according to the four axes should allow the description of the relationship between the syntactic elements of languages.

- In the second rule, languages are mapped to different $MBSD_{MoC}$ that are disjoint; e.g. L_1 and L_4 have their respective mapping to $MBSD_{SR}$ and $MBSD_{CT}$, plus $MBSD_{SR} \bigcap MBSD_{CT} = \emptyset$. Consequently, the set of properties used to characterize $MBSD_{SR}$ and $MBSD_{CT}$ are disjoint (e.g. $D_{SR} \bigcap D_{CT} = \emptyset$). Therefore, the exchanges of data between tools from these domains cannot be achieved consistently because their underlying MoCs are not compliant.

- In the third rule, the languages are mapped to different semantic domains. However, the semantic domains are not completely disjoint (the semantic domains' intersection is not empty); e.g. L_1 and L_3 have their respective mapping to

$MBSD_{SR}$ and $MBSD_{HS}$. $MBSD_{SR} \bigcap MBSD_{HS} \neq \emptyset$, which means they have a subset of common properties. Here, at least one of the intersections between the tuples describing $MBSD_{SR}$ and $MBSD_{HS}$ is not empty. Hence, a subset of properties exchangeable between these domains exists. As a result, a transformation T (e.g. $T : C_{SR} \rightarrow C_{HS}$) can be defined for the tuples that have compliant elements. However, having no control over the rest of the MoC properties for each tool is error-prone. Consequently, it is still difficult to guarantee consistent model interpretation towards different tools.

- In the fourth rule, the languages are mapped to different semantic domains and the semantic domains are fully compliant (e.g. inclusion relation on the property sets); In this case, the properties of a source $MBSD_{MoC1}$ can all be transformed to equivalent $MBSD_{MoC2}$ properties on a target domain, while keeping the fundamental rules of the source MoC domain. For instance, $MBSD_{CT}$ and $MBSD_{HS}$ are fully compliant and the semantics expressed by CT [120] is expressible from HS semantics [121]. In this context, we can define a semantic transformation on each of the tuples to complement or transform a CT model into an HS model conforming to the constraints defined in CT.

There are several classification works studying the compliance between MoCs. The Figure 3.10 of the section 3.4.3 is an example of classification. Our approach is part of such a set-logic in which we look at the existing common borders between MoCs and we provide more precise rules for measuring compliance with the four axes of concern. This opens the study of relationship to all the MoCs. However, the results in Figure 3.10 are indicative of possible relationships in a bounded list of MoCs.

4.3.3 MoC Semantics Modeling with Cometa: *Sender/Receiver with CSP*

The Sender/Receiver example shows the description of $\mathcal{LTS}$ i.e. the possible states of the system and their relationships. In this model, we present an example of communicating processes. The MoC rules are tested on the mathematical model to prove the validity of the implemented control mechanism. The modeled components must synchronize to alternate between *Send*/*Receive* requests. This simple example must implement an interaction semantics inspired by the CSP MoC presented earlier.

The operating rule of CSP requires the synchronization of the (read/write) requests that must be alternated between each pair of interconnected components. We can summarize the requirement as follows: Each write request W is necessarily synchronized to a read R request, and vice versa. Several execution scenarios are specified below and tested on the mathematical model.

In Figure 4.14, we have two application blocks A (Sender), B (Receiver). The idea is that these entities can run concurrently, but require acquittals to enable the processing of the requests issued during their execution.

We also dispose of the semantic layer composed of 3 basic modules: *Sender* (*BasiCComponent*), *Receiver* (*BasiCComponent*) and *s2r* (*MoCConnector*). The *Sender* and *Receiver* modules are interconnected by the *s2r* connector that carries a CSP execution control FSM between the two components. The components have communication ports that also carry additional control mechanisms complementary with those of the connector *s2r*. All the elements of the semantic layer are contained in the container named Top.

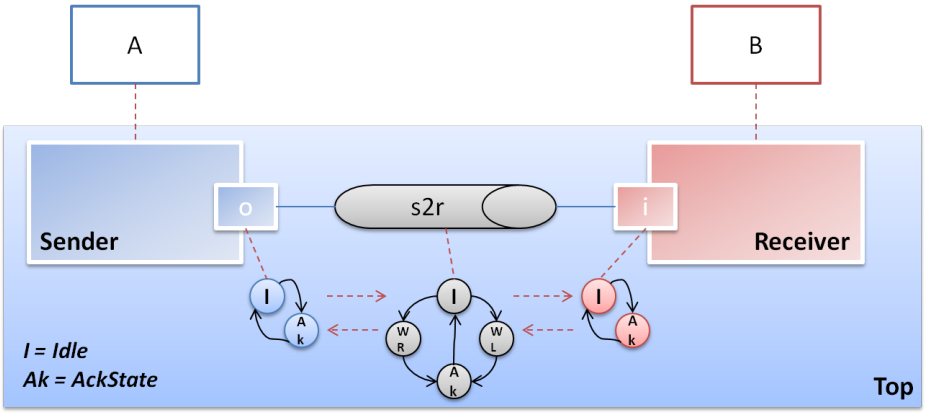

Figure 4.14: Sender/ Receiver example

$$\begin{cases} A, B \in \{Block\} \\ Sender, Receiver \in \mathcal{B}_c \\ Sender.o \in Sender.MoCPort \text{ and } , Sender.MoCPort \subseteq \mathcal{M}oC_P \\ Receiver.i \in Receiver.MoCPort \text{ and } , Receiver.MoCPort \subseteq \mathcal{M}oC_P \\ s2r \in \mathcal{M}oC_{Cn} \\ source(s2r) = Sender.o \ ; \ target(s2r) = Receiver.i \ ; \ bhv(s2r) = fsm_{s2r} \end{cases}$$

If we consider the transitional system $\mathcal{LTS}$ associated with our example, $\langle \mathcal{S}_T, \mathcal{I}, \mathcal{R}_T, \mathcal{L}, \mathcal{E}_t \rangle$

$\mathcal{S}_T = \{$
$Top.Sender.o.Idle, Top.Sender.o.AckState,$
$Top.S2R.Idle, Top.S2R.WR, Top.S2R.WL, Top.S2R.AckW,$
$Top.Receiver.i.Idle, Top.Receiver.i.AckState$
$\}$

$\mathcal{I} = \{Top.Sender.o.Idle, Top.S2R.Idle, Top.Receiver.i.Idle\}$

$\mathcal{R}_T = \{$
$(Top.Sender.o.Idle, Top.Sender.o.AckState, SendEvent)$
$(Top.Sender.o.AckState, Top.Sender.o.Idle, Ack)$
$(Top.S2R.Idle, Top.S2R.WL, WriteEvt)$
$(Top.S2R.Idle, Top.S2R.WR, ReadEvt)$
$(Top.S2R.WL, Top.S2R.AckW, ReadEvt)$
$(Top.S2R.WR, Top.S2R.AckW, WriteEvt)$
$(Top.S2R.AckW, Top.S2R.Idle, -)$
$(Top.Receiver.i.Idle, Top.Receiver.i.AckState, ReceiveEvent)$
$(Top.Receiver.i.AckState, Top.Receiver.i.Idle, ValuedAck)$
$\}$

$\mathcal{L} = \{$
$Ack, \ ValuedAck \in MoCEvent$
$SendEvent, ReceiveEvent \in MoCInterface$
$WriteEvt, ReadEvt \in MoCEvent$
$\}$

4.3.3.1 Mathematical validation

The following scenarios are used to test the functioning of the execution control mechanisms. The CSP execution control mechanism associated to *MoCConnector* is as follows: for each supported request, the *Idle* state of the connector is left as an Read / Write is issued by one of the application blocks A/ B. The synchronization between the two components will be reached when the *Idle* state is regained. From the perspective of the implemented mechanism, this means that, at some point in the exchange, a read request and writing were alternated.

The following scenarios highlight the major requests possibilities. We consider the following scenarios:

- (1) $\phi_A(W) \wedge \phi_A(W) \cdots \wedge \phi_A(W) \wedge \phi_A(W)$: The block A runs several times, while there are no requests from the block B.
- (2) $\phi_B(R) \wedge \phi_B(R) \cdots \wedge \phi_B(R) \wedge \phi_B(R)$: Conversely, block B runs several times without requests on the part of the block A. In our demonstration, we will only choose one of the two scenarios, because the result in terms of synchronization is substantially the same for both.
- (3) $\phi_B(R) \wedge \phi_A(W)$: There are alternations of a read and a write requests.
- (4) $\phi_A(W) \wedge \phi_B(R)$: There are alternations of a write and a read request requests. We will only test case 3 because the results are similar.
- (5) $\phi_B(R) \wedge \phi_B(R) \cdots \wedge \phi_B(R) \wedge \phi_A(W)$: Multiple read requests are followed by a write request.
- (6) $\phi_A(W) \wedge \phi_A(W) \cdots \wedge \phi_A(W) \wedge \phi_B(R)$: Multiple write requests are followed by a read request. There too, only the 5 cases will be tested.

Before testing the examples, there are some important questions to answer. For scenarios 1, 2, 5, 6 one can ask the question about how successive queries of the same type are managed. Although our focus is on the respect of the request alternation property, we can consider two scenarios for managing successive queries of the same type. In the first scenario, after the first query causing a change of State (e.g. $Top.S2R.Idle \rightarrow Top.S2R.WR/Top.S2R.WL$), the next similar requests are subsequently stored in a queue. Once a query of a different kind is processed, there is acquittal of the synchronization and a return of the synchronization FSM to the initial state ($Top.S2R.Idle$). In this case, a request stored in the queue is processed for a new synchronization (according to a mechanism of FIFO / LIFO).

In the second less likely scenario, the successive requests are not stored for processing later. If we consider that a request has caused a change of State (e.g. $Top.S2R.Idle \rightarrow Top.S2R.WR/Top.S2R.WL$), a new reception of the same kind of request will be lost as far as it cannot trigger any transition in the FSM. For the Sender/ Receiver example, we have 3 FSMs ($fsm_{Sender}; fsm_{s2r}, fsm_{Receiver}$).

The Figure 4.15 below shows the sequence of transition rules that are used to validate each of the scenarios. This pattern starts from the requests issued by different application blocks at ⓪, and then, on the basis of the existing properties of the $\mathcal{LTS}$ model, the defined transition rules are followed until reaching the desired CSP property (or not). After running the rules, each scenario is concluded giving the current states of the control machines, whether synchronization has taken place or not.

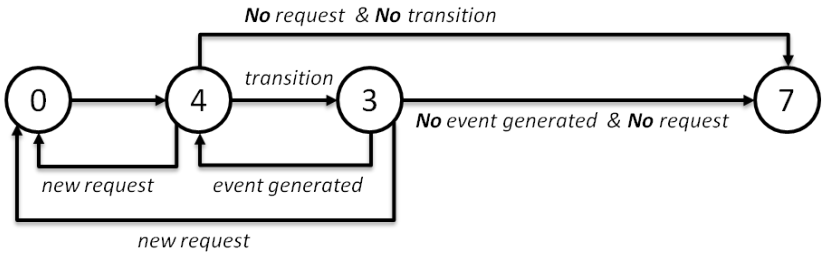

Figure 4.15: The different steps for scenario demonstration

The FSMs are all initially at their initial state. Now, for Scenario 1, the operational semantics produces the following results: The A block produces several successive write requests, then:

Scenario 1 Proof

⓪ $\phi_A(W) = SendEvent,\ SendEvent \in \epsilon_{Event}$

④ $$\frac{\gamma(SendEvent) = t : (Top.Sender.o.Idle, Top.Sender.o.AckState, SendEvent)}{next_{fsm_{Sender}}(Top.Sender.o.Idle, SendEvent) = Top.Sender.o.AckState}$$

③ returns the generated event from $Top.Sender.o.AckState$

$$\frac{\phi_A(W) = SendEvent\ ,\ \theta_{Top.Sender.o.AckState}(Top.Sender.o.AckState, SendEvent) = WriteEvt}{\theta_{Top.Sender.o.AckState}(SendEvent) = WriteEvt}$$

④ $$\frac{\gamma(WriteEvt) = t : (Top.S2R.Idle, Top.S2R.WL, WriteEvt)}{next_{fsm_{S2R}}(Top.S2R.Idle, WriteEvt) = Top.S2R.WL}$$

③ $$\frac{\phi_A(W) = SendEvent\ ,\ \ldots\ ,\ \theta_{Top.S2R.WL}(Top.S2R.WL, WriteEvt) = \emptyset}{\theta_{Top.S2R.WL}(WriteEvt) = \emptyset}$$

⓪ We introduce a new $\phi_A(W)$ request $\underbrace{\phi_A(W) \wedge \phi_A(W)}$

④ $$\frac{\gamma(SendEvent) = \emptyset \text{ i.e. } \nexists tsuchthat(Top.Sender.o.AckState, x{\in}\mathcal{S}, WriteEvt).}{next_{fsm_{Sender}}(Top.Sender.o.AckState, SendEvent) = \emptyset}$$

Generalizing the requests,
$\underbrace{\phi_{A_0}(W) \wedge \ldots \wedge \phi_{A_n}(W)}$.
.
.
The states remain the same, consequently:

$next_{fsm_{Sender}}(Top.Sender.o.AckState, SendEvent) = \emptyset$

⑦ $$\frac{(t_1(Top.Sender.o.Idle) = Top.Sender.o.AckState, \theta_{Top.Sender.o.AckState}(SendEvent) = WriteEvt)}{\left(t_2\left(t_1(Top.Sender.o.Idle) = Top.Sender.o.AckState, \theta_{Top.Sender.o.AckState}(SendEvent\right) = \right.} \ldots$$

$$\ldots \frac{(t_2(Top.S2R.Idle) = Top.S2R.WL, \theta_{Top.S2R.WL}(WriteEvt) = \emptyset)}{\left.Top.S2R.WL, \theta_{Top.S2R.WL}(WriteEvt) = \emptyset)\right)}$$

As a conclusion, there is no synchronization and the current states are:
$$\begin{cases} Top.Sender.o.AckState \\ Top.Receiver.i.Idle \\ Top.S2R.WL \end{cases}$$

The control FSM will be blocked on the *WL* state. The scenario combining successive write requests (2) will have a similar result with a lock on the *WR* state of the fsm_{S2R}.

Scenario 3 presents alternate requests where we seek the result of $\phi_B(R) \wedge \phi_A(W)$.

Scenario 3 Proof

⓪ $\phi_B(R) = ReceiveEvent$, $ReceiveEvent \in \epsilon_{Event}$

④ $$\frac{\gamma(ReceiveEvent) = t : (Top.Receiver.i.Idle, Top.Receiver.i.AckState, ReceiveEvent)}{next_{fsm_{Receiver}}(Top.Reveicer.i.Idle, SendEvent) = Top.Receiver.i.AckState}$$

③ returns the generated event from $Top.Receiver.i.AckState$

$$\frac{\phi_B(R) = ReceiveEvent\ ,\ \theta_{Top.Receiver.i.AckState}(Top.Receiver.i.AckState, ReceiveEvent) = ReadEvt}{\theta_{Top.Receiver.i.AckState}(ReceiveEvent) = ReadEvt}$$

④ $$\frac{\gamma(ReadEvt) = t : (Top.S2R.Idle, Top.S2R.WR, ReadEvt)}{next_{fsm_{S2R}}(Top.S2R.Idle, ReadEvt) = Top.S2R.WR}$$

③ $$\frac{\phi_B(R) = ReceiveEvent\ ,\ \dots\ ,\ \theta_{Top.S2R.WR}(Top.S2R.WR, ReadEvt) = \emptyset}{\theta_{Top.S2R.WR}(ReadEvt) = \emptyset}$$

⓪ We introduce a new $\phi_A(W)$ request $\underbrace{\phi_B(R) \wedge \phi_A(W)}$

$\phi_A(W) = SendEvent$, $SendEvent \in \epsilon_{Event}$

④ $$\frac{\gamma(SendEvent) = t : (Top.Sender.o.Idle, Top.Sender.o.AckState, SendEvent)}{next_{fsm_{Sender}}(Top.Sender.o.Idle, SendEvent) = Top.Sender.o.AckState}$$

③ $$\frac{\phi_A(W) = SendEvent\ ,\ \theta_{Top.Sender.o.AckState}(Top.Sender.o.AckState, SendEvent) = WriteEvt}{\theta_{Top.Sender.o.AckState}(SendEvent) = WriteEvt}$$

④ $$\frac{\gamma(WriteEvt) = t : (Top.S2R.WR, Top.S2R.AckW, WriteEvt)}{next_{fsm_{S2R}}(Top.S2R.WR, WriteEvt) = Top.S2R.AckW}$$

③ $$\frac{\phi_B(R) = ReceiveEvent\ ,\ \phi_A(W) = SendEvent\ ,\ \dots\ ,\ \theta_{Top.S2R.AckW}(Top.S2R.AckW, -) = \{Ack, ValuedAck\}}{\theta_{Top.S2R.AckW}(-) = \{Ack, ValuedAck\}}$$

④ $$\frac{\gamma(Ack) = t : (Top.Sender.o.AckState, Top.Sender.o.Idle, Ack)}{next_{fsm_{Sender}}(Top.Sender.o.AckState, Ack) = Top.Sender.o.Idle}$$

$$\frac{\gamma(ValuedAck) = t : (Top.Receiver.i.AckState, Top.Receiver.i.Idle, ValuedAck)}{next_{fsm_{Receiver}}(Top.Receiver.i.AckState, ValuedAck) = Top.Receiver.i.Idle}$$

③ $$\frac{\phi_B(R) = ReceiveEvent\ ,\ \phi_A(W) = SendEvent\ ,\ \dots\ ,\ \theta_{Top.Receiver.i.Idle}(Top.Receiver.i.Idle, ValuedAck) = \emptyset}{\theta_{Top.Receiver.i.Idle}(ValuedAck) = \emptyset}$$

$$\frac{\phi_B(R) = ReceiveEvent\ ,\ \phi_A(W) = SendEvent\ ,\ \dots\ ,\ \theta_{Top.Sender.o.Idle}(Top.Sender.o.Idle, Ack) = \emptyset}{\theta_{Top.Sender.o.Idle}(Ack) = \emptyset}$$

For the fsm_{S2R},

④ $$\frac{\gamma(\text{-}) = t : (Top.S2R.AckW, Top.S2R.Idle, -)}{next_{fsm_{S2R}}(Top.S2R.AckW, -) = Top.S2R.Idle}$$

③ $$\frac{\phi_B(R) = ReceiveEvent\ ,\ \phi_A(W) = SendEvent\ ,\ \dots\ ,\ \theta_{Top.S2R.Idle}(Top.S2R.Idle, -) = \emptyset}{\theta_{Top.S2R.Idle}(-) = \emptyset}$$

⑦ The rule is not defined here due to its size. However it can be similarly defined as in the previous demonstration of scenario 1. Here, our main focus is the final current states that demonstrate the synchronization rule we proposed for CSP in the beginning:

$$\begin{cases} Top.S2R.Idle \\ Top.Sender.o.Idle \\ Top.Receiver.i.Idle \end{cases}$$

Scenario 4 can be shown in a similar manner to Scenario 3. For Scenario 5, 6, one can imagine that the succession of requests of the same type will lead to both cases

below.

Scenario 5 Proof

$$\phi_{A_0} \wedge ... \wedge \phi_{A_n} \Rightarrow \begin{cases} Top.Sender.o.AckState \\ Top.Receiver.i.Idle \\ Top.S2R.WL \end{cases}$$

$$\phi_{B_0} \wedge ... \wedge \phi_{B_n} \Rightarrow \begin{cases} Top.Receiver.i.AckState \\ Top.Sender.o.Idle \\ Top.S2R.WR \end{cases}$$

In the first case, if we add a $\phi_{B_0} = SendEvent$ ($\phi_{A_0} \wedge ... \wedge \underbrace{\phi_{A_n} \wedge \phi_{B_0}}$), the coupling of the last two requests ends up as in the scenario 3. Consequently:

$$\phi_{A_0} \wedge ... \wedge \underbrace{\phi_{A_n} \wedge \phi_{B_0}} \Rightarrow \begin{cases} Top.Sender.o.Idle \\ Top.Receiver.i.Idle \\ Top.S2R.Idle \end{cases}$$

Similar results are obtained for $\phi_{B_0} \wedge ... \wedge \underbrace{\phi_{B_n} \wedge \phi_{A_0}}$ QED

In the latter case, since the connector's FSM has returned to the *Idle* State, then we can choose the next synchronization to put in place in the queue of requests ($\phi_{A_{n-1}}$ or $\phi_{B_{n-1}}$).

4.3.3.2 By Simulation

The Sender/Receiver model has been reproduced in the Rhapsody Modeling and Simulation environment.

Rhapsody [85] is a proprietary tool that provides a system development environment (mostly embedded systems) based on the use of UML language and profiles. Rhapsody incorporates several activities of SW development cycle (requirements specification, high-level system specification, code generation, simulation and testing, etc.). Regarding the specification of systems, the tool integrates UML component models to specify communicating concurrent entities. In such models, the components are interconnected via ports and connectors. These elements are classes that may have a behavior (*UML Statechart*) and attributes (*UML Attributes*). Besides, the communication is provided by event (signal) exchanges e.g. *callEvent*, *receptionEvent*.

The simulation of the Rhapsody engine can interpret models of interconnected components using the FSMs and relying on the DE execution semantics. Thus, the exchanges between the system components are considered as sequences of event requests temporarily stored in storage elements (queue, FIFO, LIFO, etc). The system model defines execution end conditions (e.g. stop Event, variable defining the number of allowed executions, etc). While the execution stop condition is not reached, the scheduling behavior constantly observes the storage elements to process events to the target components, and updates the static values which may affect the execution stop condition.

For the example, the structural aspect of the semantic layer is shown in Figure 4.16. In this figure, the colored boxes describe two *BasicComponent* (*Sender*, *Receiver*) to which *MoCPort* (itsO:o = *Sender.o*, itsI:i = *Receiver.i*) are associated; and functional application blocks (itsA:A, itsB:B) are allocated. The upper hierarchical view of the model shows the *MoCConnector* placed between these two components

(i.e. itsS2R:S2R = *Top.s2r*). The semantic layer itself is made up of *Sender*, *Receiver*, (itsO:o = *Sender.o*, itsI:i = *Receiver.i*) and (itsS2R:S2R = *Top.s2r*) that control the execution of (itsA:A, itsB:B).

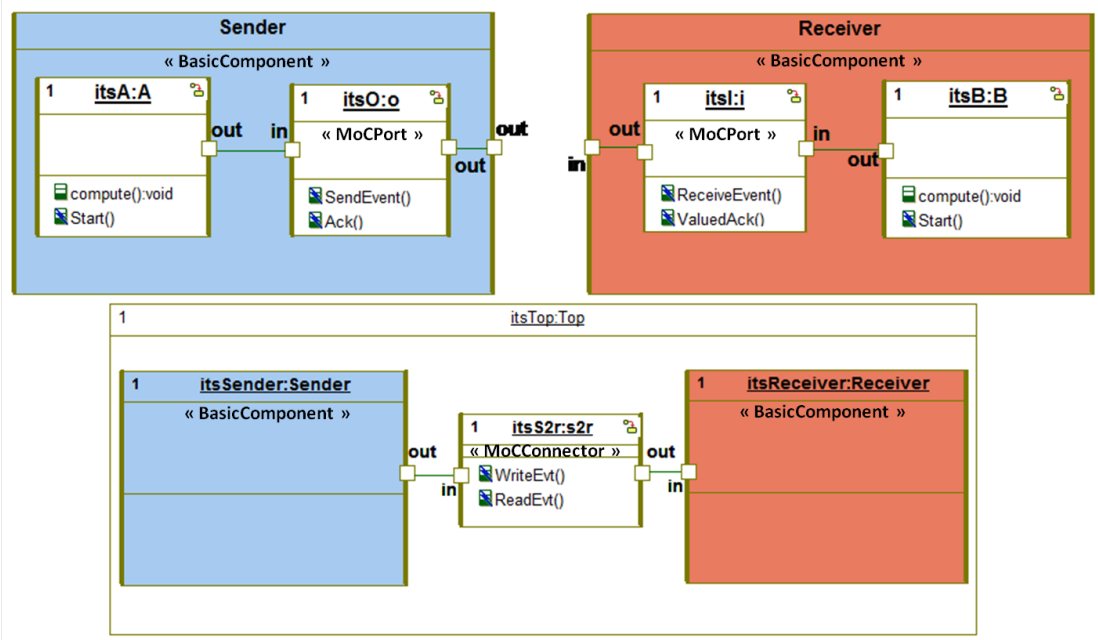

Figure 4.16: Sender/Receiver model in Rhapsody

Figure 4.17 shows the control FSMs that are placed on *Sender.o*, *Receiver.i*, and the FSM of *Top.s2r*. These FSMs constrain the execution of the (itsA:A, itsB:B) modules to obtain the presented execution traces after simulation. The blocks (itsA:A, itsB:B) being concurrent, itsB:B sends a *receive* request which is intercepted by the *Receiver.i* port of the semantic layer . At this level the SL starts the control mechanism. *Receiver.i* sends the event *ReadEvt* to other components of the SL i.e. *Top.s2r*. *Top.s2r* remains in a blocking state (*Top.s2r.WR*) until it receives a write event (*WriteEvt*). The event arrives as in the other side, the sending request from itsA:A is transformed into a *WriteEvt* by the *Sender.o* port. Finally, *Top.s2r* unblocks all the components by sending *Ack*, since the *handshake* has been completed. The *Ack* releases (itsA:A, itsB:B) to produce new requests.

The simulation traces obtained reflect the same results as those obtained with the formal models in Scenarios 1 and 3.

4.3.4 Time Description: Time-Based Control

Time models allow the control of exchanges between the parts of semantic layers in the sense that, linking these elements with explicit clocks provides control over execution with respect to the rates of clocks. The evolution of time with clocks is driven by the definition of time bases and constraints expressed between the clocks.
We will see these concepts in more detail later in this chapter. A Cometa model of time $\mathcal{T}_{mod}$ is described by the 6-tuple $\langle \mathcal{T}_{Str}, \mathcal{C}lk_{Type}, \mathcal{C}lk_{Cstr}, \mathcal{C}lk_{Inst}, \mathcal{C}lk_{Rel}, \mathcal{T}_{Base} \rangle$

- $\mathcal{T}_{Str}$ represents all the time structures that can be described using the *TimeStructure* concept which is abstracted in Cometa for time base description.

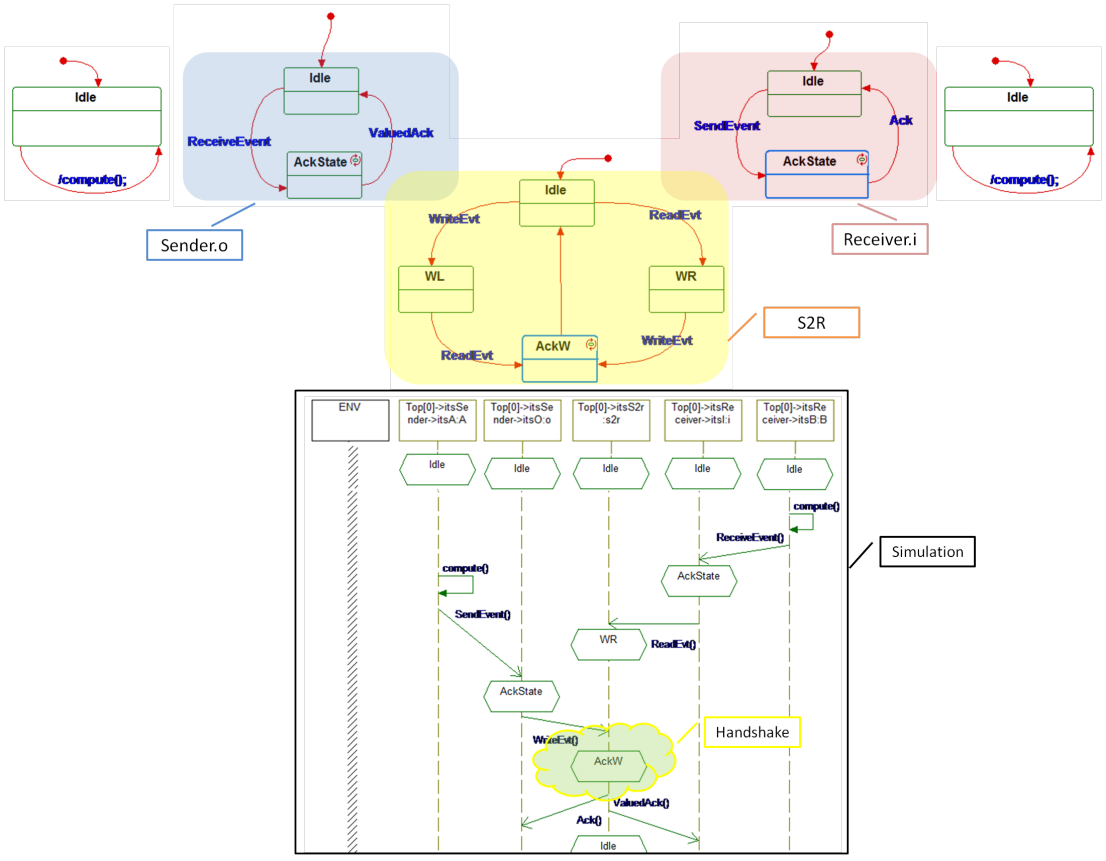

Figure 4.17: Sender/Receiver Simulation Traces

- $\mathcal{C}lk_{Type}$ represents all the types of clocks that can be set from the concept *ClockType* in Cometa. The clock types characterize clocks with their specific properties. This concept is inspired form the MARTE model of time.

- $\mathcal{C}lk_{Cstr}$ represents the set of constraints that can be described between clocks. Constraints are relationships such as interdependence (precedence, coincidence, etc). The Cometa concept for the description of the constraints is *ClockConstraint*.

- $\mathcal{C}lk_{Inst}$ represents the set of clock instances participating in the control of the execution for a given model. In Cometa, the concept for such specification is *ClockInstance*.

- $\mathcal{C}lk_{Rel}$ represents the set of relations that it is possible to describe between the types of clocks.

- $\mathcal{T}_{Base}$ represents all the time bases that are defined in the time structures. These time bases allow the characterization of the types of clock by specifying the kinds of instants that compose a *ClockType*. In Cometa, the abstraction of the *TimeBase* concept allows the representation of time bases.

In the sequel to our specification, we clarify the meaning of the concepts that are used for the description of time models. As shown in Figure 4.18, these different concepts are needed for the description of clocks and their relationship.

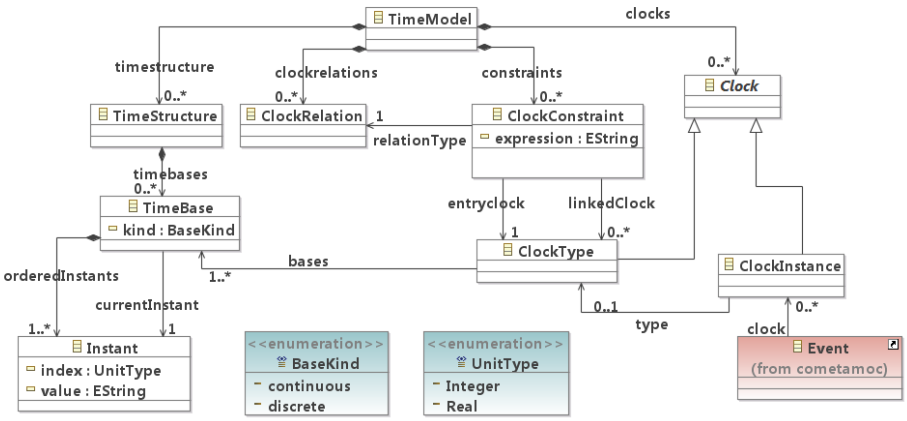

Figure 4.18: Excerpt of the Time concern in Cometa DSML

4.3.4.1 Cometa *TimeModel*

A *TimeModel* (4.21) allows the capture of different entities participating into the description of the time as well as the relationships that bind these entities. The concept is used to describe the structures of time (*TimeStructure*), clocks (*Clock*), several types of relations that could bind the clocks (*ClockRelation*) and finally constraints that exist between pairs of clocks. The formalization for a *TimeModel* is provided below:

$$\forall\ tm \in \mathcal{T}_{mod} \text{ then } \begin{cases} tm ::= \{TimeStructure\}; \{ClockInstance\}; \{ClockType\}; \\ \{ClockConstraints\}; \{ClockRelation\} \\ \{TimeStructure\} \subseteq \mathcal{T}_{Str} \\ \{ClockInstance\} \subseteq \mathcal{C}lk_{Inst} \\ \{ClockType\} \subseteq \mathcal{C}lk_{Type} \\ \{ClockConstraints\} \subseteq \mathcal{C}lk_{Cstr} \\ \{ClockRelation\} \subseteq \mathcal{C}lk_{Rel} \end{cases} \tag{4.21}$$

4.3.4.2 Cometa *TimeStructure*

A *TimeStructure* (4.22) is a concept enabling the description of several time bases *TimeBase*. In our approach, the structures of time serve more to organize the time bases into libraries.

$$\forall\ ts \in \mathcal{T}_{Str} \text{ then } \begin{cases} ts ::= \{TimeBase\} \\ \{TimeBase\} \subseteq \mathcal{T}_{Base} \end{cases} \tag{4.22}$$

4.3.4.3 Cometa *TimeBase*

Each *TimeBase* (4.23) defines an ordered set of instants in which clocks take a description of their evolution. From the time bases, each clock has information about numbers of instants of its cycle; the ordering of the instants; the current instant in the sequence of instants, as well as the type of the time base. This latter concept is important because it allows the type of evolution of the clocks (discrete, continuous) to be specified. The following formalization highlights this aspect.

$$\forall\ tb \in \mathcal{T}_{Base} \text{ then } \begin{cases} tb ::= \{Instant\}; currentInstant; kind \\ currentInstant\ : \mathcal{T}_{Base} \rightarrow tb.\{Instant\}, \\ kind\ : tb.\{Instant\} \rightarrow \{discrete,\ continuous\} \end{cases} \tag{4.23}$$

4.3.4.4 Cometa *Instant*

The *Instant* concept (4.24) represents a time evolution step. It allows the capture of the specific instants that serve as abstract time measurement units. In the Cometa approach, each *Instant* has a unit $< unit >$ which allows the fineness of the captured instant to be specified. The unit types are expressed either as Integers ($\mathbb{N}$) or Reals ($\mathbb{R}$). The description of *Instant* is formalized in the following manner:

$$\forall\ ti \in \mathcal{C}lk_{Inst} \text{ then } \begin{cases} ti \text{ is defined by the following functions:} \\ index\ : \mathcal{C}lk_{Inst} \rightarrow \{\mathbb{N}|\mathbb{R}\}, \\ value\ : \mathcal{C}lk_{Inst} \rightarrow \{String\} \end{cases} \tag{4.24}$$

4.3.4.5 Cometa *ClockType*

A *ClockType* (4.25) allows the definition of a type of clock linked to a time base. A *ClockType* can be associated with several time bases already defined from *TimeStructure*. The association of a clock to a *ClockType* allows the specification of the rates of clock events used in the semantic layers. The following formalization describes the *ClockType*:

$$\forall\ ck \in \mathcal{C}lk_{Type} \text{ then } \left\{ ck ::= \{TimeBase\} \right. \tag{4.25}$$

4.3.4.6 Cometa *ClockConstraint*

A *ClockConstraint* (4.26) allows the constraints linking two defined clocks to be modeled. Constraints use the relationships (*ClockRelation*) for coupling the instants of the time bases related to the clocks in a coherent manner. The formalization of the *ClockConstraint* is as follows:

$$\forall\ cc \in \mathcal{C}lk_{Cstr} \text{ then } \begin{cases} cc ::= \{ClockType\}; rl : ClockRelation; rc : ClockType \\ rl \ : \mathcal{C}lk_{Cstr} \rightarrow \{ClockRelation\} \\ rc \ : \mathcal{C}lk_{Cstr} \rightarrow \{ClockType\} \\ express \ : \mathcal{C}lk_{Cstr} \rightarrow \{String\} \end{cases} \tag{4.26}$$

The $< express >$ function ($< expression >$ attribute of *ClockConstraint* in Figure 4.18) provides a field to more finely describe the relationship between *ClockType* e.g. in the form of CCSL constraints. The $< rl >$ function ($< relationType >$ in Figure 4.18) define the type of the constraints in regard to the existing *ClockRelation* (e.g. precedence, simultaneity). Finally, the $< rc >$ function ($< entryClock >$ in Figure 4.18) gives the entry *ClockType* for which relations are defined with other existing *ClockType*.

4.3.4.7 Cometa *ClockRelation*

The *ClockRelation* (4.27) are used to describe relationships between the types of clocks (*ClockType*). In Cometa, elements of type *ClockRelation* are described in the form of strings thus, leaving flexibility to the definition of new rules. However, as previously addressed, relations are of types: e.g. coincidence, precedence, synchronization. The *ClockRelation*, once defined, are used for the description of constraints between pairs of clocks. The formalization is described as follows:

$$\forall\ cr \in \mathcal{C}lk_{Rel} \text{ then } \exists fname \text{ such that } fname \ : \mathcal{C}lk_{Rel} \rightarrow \{String\} \tag{4.27}$$

In the Cometa approach, *Clock* represents an abstract concept to capture time evolution. Such time evolution is measured thanks to the time bases and events generated by clocks. Fine descriptions of the relationship between the instants of the supplied time bases provide the event occurrence rate. The events are injected into the semantic

layer's control FSMs. The Figure 4.19 shows an abstract vision of the impact of a time base on the communication between control FSMs. The time base is composed of several instants $(i_0, i_1 ... i_n)$. At the instant i_1, there is an occurrence of the clock event *Clk_Evt* which causes a change of state between *SA* and *SB*. The occurrence frequency of the instants can be based on reals (continuous), or on integer values (discrete).

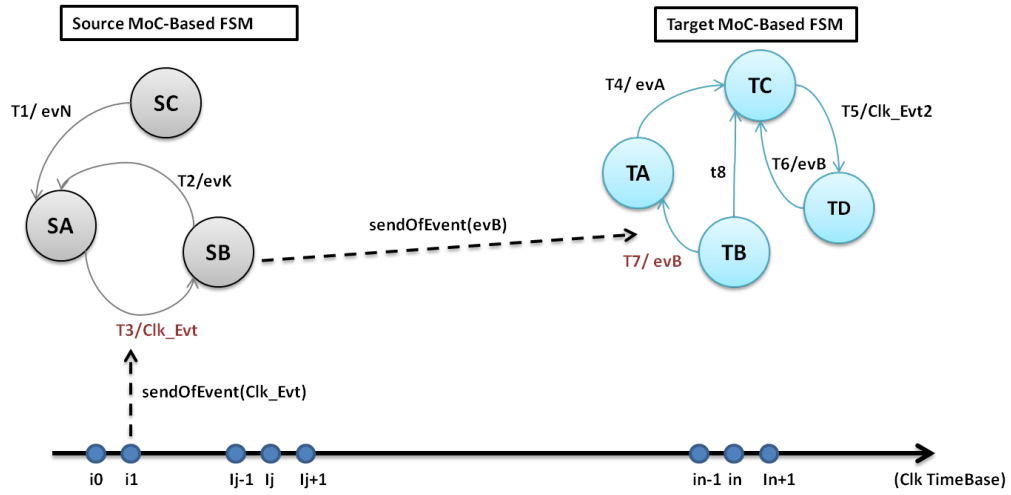

Figure 4.19: Representation of Time model Usage in Semantic Layers

Only basic experimentations have been made around the modeling and specification of a time model in this thesis. We mainly worked on the definition and use of the mechanisms of execution control using FSMs. Consequently, our focus is on the description of the contribution around the semantic layers and the control FSMs.

4.4 Conclusion

In this chapter, the focus was on the description of the DSML Cometa, its formalization and the presentation of simple validation samples of the operational semantics. The first part of the chapter offers a practical separation of concerns that are represented in the DSML to meet the need of representing semantic layers. Separating the concerns adheres to the overall scheme of system representation which nowadays promotes the separation of concerns as a major principle for complexity reduction.

The Cometa DSML abstract several concepts for the description of the layers. The resulting metamodel allows the representation of topological and semantic layers for the execution control of requests from concurrent application blocks. The DSML is formalized in order to facilitate its use for analysis and validation. Indeed, it offers a formal and easy description of concepts as well as links between concepts.

With the phases of formal specification of the operational semantics, we can see the relevance of models that are produced for the execution control on some simple examples. A transition system model is then associated with each FSM. The FSMs implement the MoC control rules. These simple examples are representative of the types of exchanges between components and the modus operandi of the layers. Our goal in the next chapter is to use semantic layers in design chains to adapt and enrich the application models for the specification and analysis between different tools.

5

Tool Semantic Interoperability using Cometa and Experimentation

Contents

5.1 Introduction

During the System Level Modeling (SLM) steps, reusing models between tools is difficult despite the important contributions of the MDE (e.g. metamodeling, model transformation). Indeed, it becomes difficult to ensure that a system model will preserve its consistency after having been manipulated by various tools. This shortcoming is due to the fact that the design tools often rely on semantics and syntaxes from different formalisms and environments. While the syntax aspect is quite well addressed in research, semantics continues to be the bane of tool-providers and designers. Consequently, the preservation of semantics and the accuracy of the models is a real challenge for system designers.

In this chapter, we demonstrate the use of Cometa to ensure the preservation of the semantics of models during the exchange phases between tools dedicated to SLM and formal design.

This experiment is done through the description of a methodological approach to designing a system including several new phases (comparison of semantics, inclusion of semantic layers, etc.) necessary to avoid inconsistent semantic issues. Through the Radar application use case, we present the different design steps starting from the identification of activities, the choice of tools (depending on design activities), to implementing an executable solution for various SLM and formal design environments used for the synthesis of a final implementation.

This example also illustrates the positioning of Cometa in the activities of the design process to reduce the risk of semantic misinterpretations; thus demonstrating its potential. Finally, we present a novel automated design flow connecting the following SLM and formal design tools: Modeler IBM Rhapsody, the ForSyDe tool and the SpearDE environment.

5.2 Integrating Semantic Layers into Design Flows

The enrichment of the models produced by different tools is based on weaving and allocation mechanisms of: application models with the MoC based semantic layers.

A Semantic Layer, similar to the source application model, is built with Cometa in order to describe the combination of execution control behaviors. The SLs allow the missing semantic properties (static and operational semantics) to be placed. Static and operational semantic properties are captured in the form of reusable libraries.

5.2.1 Overview of the Approach

As shown in Figure 5.1, one can define several models of semantic properties in the repository (e.g. SDF, KPN, CSP). From the libraries, the designer uses the captured models to produce a model combining the source application model and a model of semantic layer called Allocated Semantic Model (ASM).

The ASM now gathers the source model properties and includes new properties mandatory for their correct interpretation in the source and target environment. The application blocks allocation / encapsulation are done using *BasicComponent*s.

We provide more details on the mechanisms of allocation/ encapsulation in Section 5.3.2 related to the weaving of application models and semantic layers.

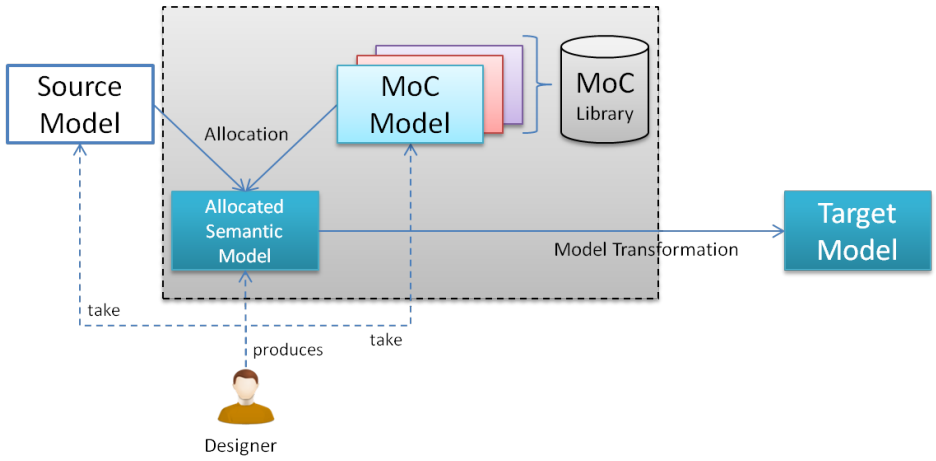

Figure 5.1: Guidelines for MoC model integration with Application Models

As previously stated, the descriptions of semantics can start with static properties then, if necessary, end with an operational specification of the execution control mechanism based on the MoCs.

In order to apply the approach in a concrete case, in the next section we propose to describe the experimentation made with industrial partners in the context of the iFEST project.

5.2.2 Defining a Tool Chain for a Radar Streaming Application

Our approach focuses on the design activities such as specification, analysis and code generation. In this use case, we look at the description of the *BurstProcessing* module of a Radar system; and we look at the semantic properties related to such a type of system.

The choice of the Radar application is motivated by the fact that the system contains several sub modules that are heterogeneous in terms of performed computations and interaction mechanisms between modules. Moreover, we have a system description which has required the expertise of several engineers. As a result, this is a major change compared to the simple examples previously used for experimental purposes.

Accordingly, on the one hand, the size of the system and its complexity requires a high-level design approach (based on system engineering) that will be successively refined up to the final design of the system. On the other hand, the semantics implemented i.e. CSP, KPN and especially the Array-OL semantics are interesting in terms of formal specification and execution properties (e.g. synchronous communication, use of multidimensional data, tasks and data parallelism, and execution semantics based on the scheduling of dependency graphs with production and consumption rates specifying the size and the nature of the exchanged data).

The various semantics and methodological constraints cited above imply the use of several design tools in an operational design flow. The models produced in this design flow must preserve their semantics towards the different tools. The tool chain used for the Radar system will be presented in Section 5.3. The use of Cometa in this tool chain is legitimate because it allows the expression of the missing semantic models between inter-tool and intra-tool model exploitation. Different semantics will be formally described in the following sections. In the remainder of this section, we will describe the components of the Radar model.

5.2.2.1 Description of the Radar Streaming Application

A Radar system has one or more antennas capable of emitting signals in the environment (see Figure 5.2).

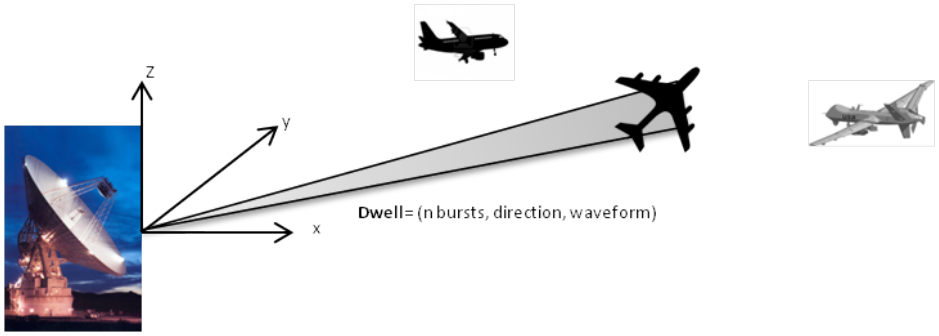

Figure 5.2: Radar Detection System: Target Detection

Radar signals are oriented in *preferential detection* directions with a minimization of the impact of *parasite* data such as: noise (involuntary), jamming (voluntary).

The objective of the system is to be able to detect objects (e.g. aircraft, obstacles, etc.) in a given detection area. The important information is related to their distance, their speed, their direction. Thus, signals are sent in all directions. At each send sequence, an angular area is scanned. Signals to send are grouped in a *Dwell*. A *Dwell* consists of a set of signals to send into the environment according to fixed angular parameters. Signals are called *Burst*.The concept of *Burst* is described in the following paragraphs. The specific settings of each Dwell determine the type of detection that the user wants to perform; and according to the type of mission the Dwell settings help to have accurate results. Several observation modes are defined for the Radar application:

- *Search Mode*: which consists in sending dwells to achieve a repeated and exhaustive scanning of areas one after the other. For this type of exploration, the dwell is composed of 3 bursts.
- *The Active Tracking Mode*: consisting of the tracking and the update of information related to previously sent signals. For this type of exploration the dwell is composed of single bursts that have variable duration (e.g. 1 single Burst with a send duration fixed between 10 and 20 ms).

A *Burst* is a signal sent into the environment by an antenna. It produces an echo after a certain time depending on the intersection with a target obstacle. The Burst consists of a fixed set of periodic pulses. The specificities of the Burst are described as: a waveform describing the carrier frequency of a pulse, or the number of times that a pulse is sent. The echoed burst after a target is met has a signature that relates the echo to its bound Burst. The nature of the echo allows the distance (the delay of the echoed pulse), direction, speed (the phase variation of the echoes from pulse to pulse) of an object or an obstacle to be determined.

The Pulse is characterized by information including: the distance covered d, the horizontal sending angle of the pulse i.e. *azimuth*, or the *elevation* angle of the pulse. d is also known as *range* and is given either in the form of a *dmax* representing the maximum distance reached by the signal, or it represents the distance covered by the signal before reaching the target. This value is calculated on the basis of the speed of light and the time taken to receive the *echoedBurst*.

The main Radar model consists of 5 modules as shown in Figure 5.3. These modules are:

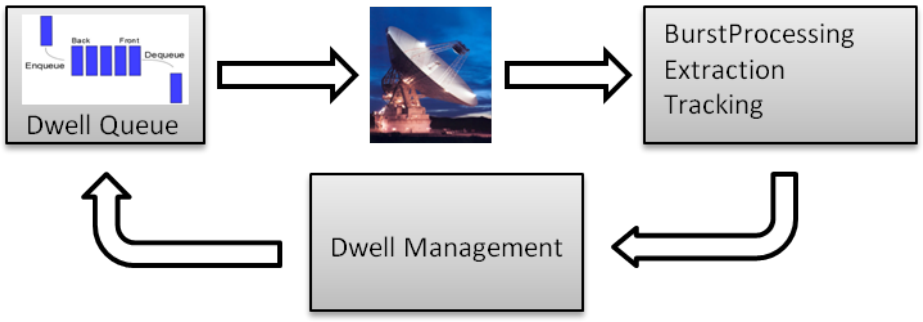

Figure 5.3: Modules of the Radar Detection System

- *DwellsManagement*: The *DwellsManagement* has responsibility for selecting the types of Dwell to send to antennas and their instant of sends, depending on the type of observation that the user wants to perform and on the Dwell's priority. The candidate Dwells for sending are stored in a FIFO waiting to be sent by the antennas.

- *AntennaControl*: The *AntennaControl* module includes all of the features for sending the Burst of a Dwell into the environment, and then retrieving the echoes generated by these Bursts to transmit them to the *BurstProcessing* for their processing. At this stage, the information that the Radar wants to retrieve depends on the direction and angle of sending of the Bursts, and the time that has elapsed before receipt of the echoes corresponding to the different Bursts.

- *BurstProcessing*: The *BurstProcessing* module has several features aimed at processing the echoes received by the *AntennaControl* for each Burst. The processings allow information related to the presence of targets, their distance, their speed and direction to be retrieved. The processings allow among other things the removal of noise in echoed signals in order to render only significant information. The processed data are multidimensional arrays and contain several parameters to be taken into account. For each echo, *BurstProcessing* produces Hits [1] for the *Extraction* sub-module.

- *Extraction*: The *Extraction* module waits to receive all Hits of a particular Dwell (i.e. all of the hits generated for all Bursts of a Dwell). Afterwards, it consolidates the results received by producing lists of Plots [2] with targets. The Plot lists are transmitted to the Tracking module.

- *Tracking*: The *Tracking* module, based on the information that it receives from *Extraction*, allows the determination of whether a target is present at a given environment location and thus identifies it. The parameters that it extracts from the received data enable the Tracking module to display the position of a target; as well as to define updates as long as the target is present in the observation area. To be able to update the position of the targets, tracks are produced allowing

[1] The hit number stands e.g. for a search radar with a rotating antenna for the number of the received echo pulses of a single target per antenna turn. source: http://www.radartutorial.eu/01.basics/

[2] Plot: Detected target echoes against a background noise

the generation of new specific signals in the environment to follow the target. A track contains the position and the velocity of a target vector.

We complete the system with a loop back to the *DwellsManagement* for a new selection of signals to send depending on the missions and to perform tracking. The two activities can be done in parallel.

The objective of this experiment is to focus on a subset of the Radar (the *BurstProcessing*) system and make it executable modeling within and towards different tools.

5.2.2.2 Focus on The Radar Burst Processing Application

For the implementation of the system, each of the modules that we have presented is refined into several hierarchical sub modules which add complexity to the system specification. We propose to focus on the computations of *BurstProcessing*. The choice of this module is motivated by the fact that it is based on a rich semantic specification manipulating multidimensional data and parallel task execution mechanisms.

The sub-system below (see Figure 5.4) shows a specific focus on exchanges between the *AntennaControl* and the refined *BurstProcessing* modules.

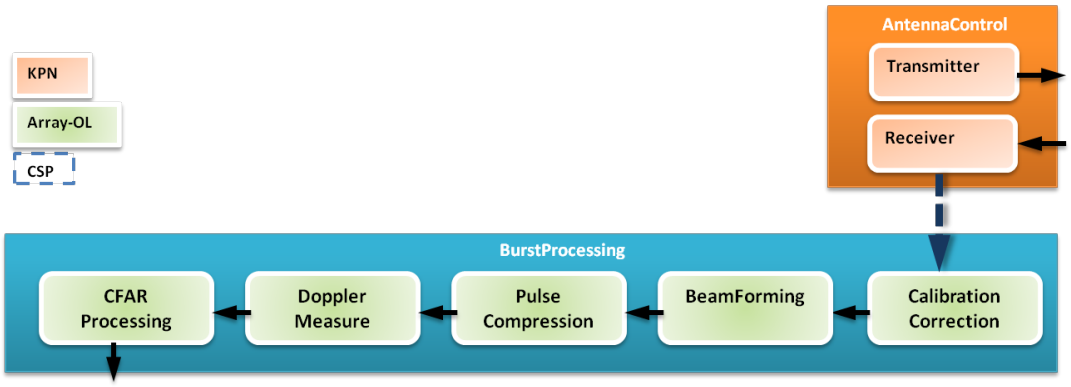

Figure 5.4: Presentation of the *BurstProcessing* module

Both modules are refined as hierarchical modules presented in the following paragraphs:

AntennaControl is a hierarchical module that contains a signal transmitter *Transmitter* sub-module, and a signal receiver sub-module *Receiver*.

- *Transmitter* is connected to the mechanical antenna for signal sending, it allows Bursts contained in a Dwell in the environment to be sent. The Transmitter converts the signal (into analog) with a D/A converter before it is sent into the environment. Each burst has a signature; the signature allows the recognition of the signal reflected by a target (echo) to the *Receiver* module. The reflected signal can also be called *echoedBurst*. The *echoedBurst* are the echoes of pulses contained in the Burst.

- *The Receiver* is the module responsible for retrieving and transforming (A/D) the echoed signals before they are transmitted to the *BurstProcessing*. The received signals are converted into digital signals by an A/D converter. The resulting signal is transmitted to the *BurstProcessing* that applies several processings and

filterings on the signal sequences to reduce the noise introduced by the environment in the reflected signal.

In order to calculate the distance to the target, we will need to build a matrix that stores the signature of a burst, its date of sending, as well as its return date. This captured information (power, range gate, gate speed, etc) are in the form of multi-dimensional arrays operated by the *BurstProcessing* module. For example, the input data in the detection mode represent a cube of 64 antennas, N pulses and M range gates (e.g. a 64 x 19 x 2000 data cube to the third mode having 19 pulses per burst repeating every 0.2ms).

The *BurstProcessing* module is a signal processing module consisting of five sub modules as shown in Figure 5.4: *CalibrationCorrection*, *BeamForming*, *PulseCompression*, *DopplerFiltering* and *CFAR_Processing* modules.

- *The Calibration* module takes as input the *echoedBurst* signals and evaluates their calibration based on a given calibration table.
- *The BeamForming* module calculates the filtering coefficients (weights) based on the input data, i.e. the radar bursts and the calibrated echoedBurst signals provided by *CalibrationCorrection.*
- *PulseCompression*: this phase consists in improving the signal-to-noise ratio and the interference (time domain convolution). This behavior is implemented in the classic mode by a point 32 direct FFT on the *range gate* axis, followed by the multiplication of the Fourier domain and inverse FFT.
- The Radial velocity (range-speed) of a target is calculated thanks to *Doppler-Filtering.* The pulses returned from a reflected signal are processed to calculate the landslide frequency between the transmitted signal and what is received. To perform this calculation, an FFT is made on the pulse axis.
- With the *CFAR_Processing module*, the detection step uses the constant false-alarm rate CFAR to compare a radar signal response to its nearby signal responses to determine if a target is present and uses the consolidation of targets to eliminate multiple targets when it is in reality only a single target report.

We will see the functioning implications of these different modules in Section 5.2.2.4 describing the semantic constraints. In what follows, we adopt the SLM design methodology and focus on design activities; selection and description of tools for this use case.

5.2.2.3 Tool Selection for the Design Flow

Like most complex system development processes, HW and SW codesign development integrates several steps such as specification, analysis, refinement and partitioning to achieve the final implementation of the system. The different activities occur depending on the direction of the design flow i.e. bottom-up or top-down. For the experimentation a top-down approach is implemented.

5.2.2.3.a Design Process Activities

As shown in Figure 5.5, a top-down design approach promotes several specification steps (SLM). The specification stages integrate activities for requirements engineering, structural modeling, behavioral modeling and refinement of high-level specification models.

The ideal situation in this design schema emphasizes the ability to analyze and validate the models before refinement, thus finding design flaws that can involve changes in the models.

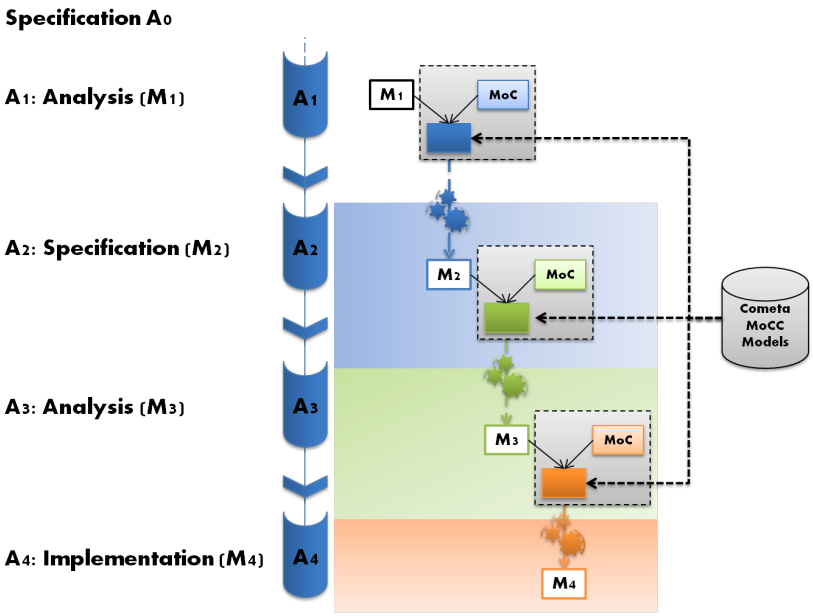

Figure 5.5: Positioning Cometa Models within a Design Process with heterogeneous semantics

Analysis activities include several variations depending on the level of abstraction and the type of analysis to perform. For instance, analysis can relate to model-checking based on mathematical foundations for verification of static and dynamic properties. Analysis also refers to the simulation based on the study of the execution traces induced by behaviors. Finally, the analyses also include Design Space Exploration activities. Successive refinements and verification steps lead to a final implementation of the system by synthesis (code generation).

The different activities are carried out by different tools. Each of these tools potentially uses a specific formalism to express the models, to offer different services, and more significantly, to implement different semantics identified by different colors on the Figure.

In these design flows, we define a reusable semantic specification to adapt models between the tools. At each connection point between tools, semantic layer models providing the adaptation are produced, combined with the application model, and then operated by transformation rules to produce output models containing the required semantic properties.

For the system of the use case, we primarily focus on the specification and analysis by simulation and Design Space Exploration. The synthesis part will be managed by dedicated tools. The exercise is to select effective tools to carry out these activities in a "user friendly" environment while maintaining a good quality of the system models i.e. preserving the semantics of the models.

There are a large number of development environments dedicated to the design of

embedded systems. Some of these environments have been presented in Section 2.3.1.3 and are "user friendly".

For the above activities, tools such as IBM Rhapsody, Papyrus, TopCased, RSA that are model-based design environments allow the study of requirements, specification and simulation of models.

In our experiments we use Rhapsody for specification and analysis as it provides easy and executable model specification facilities. The specified and analyzed models from Rhapsody are correlated with dedicated HW/SW codesign environments which are ForSyDe and Spear. The advantage of ForSyDe is that it allows the description of heterogeneous, formal and analyzable models (described in SystemC). Finally, the tool SpearDE from Thales is for simulation, Design Space Exploration and synthesis of code analysis.

5.2.2.3.b Tool Description

Rhapsody [85] is a tool that provides an environment for system development (mostly embedded systems) based on the use of the UML formalism and its different profiles for embedded system development. Rhapsody incorporates several activities of a design process such as: requirement specification, specification and system level modeling (SLM), code generation, simulation, etc). For specification, the tool integrates UML component diagrams to specify connected concurrent entities.

In the models, the components can be directly connected by links (Link), or connect through ports and connectors (see the conceptual model in Figure 5.6), these elements are classes that can have behavior (StateChart of UML) and several attributes (UML attributes) and functions. In addition, the designer can define hierarchical components through the concept of *Composite* Class. The communication is ensured by function calls and through exchange of events (signals) e.g. CallEvent, ReceptionEvent.

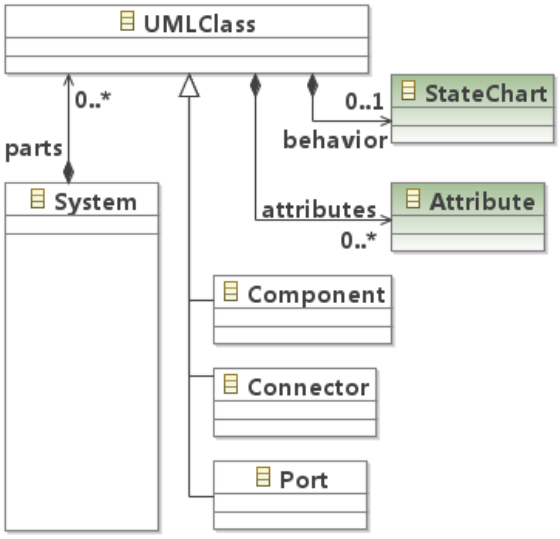

Figure 5.6: Conceptual Model of concepts to describe component models in Rhapsody

ForSyDe is a design framework for heterogeneous embedded system modeling and is implemented in the form of C++ class libraries on top of the IEEE standard SystemC. These system models are networks of hierarchical concurrent and executable processes

connected by signals. The signals are mapped to the SystemC FIFOs, and the processes are defined as SystemC modules that possibly rely on some legacy application functions provided by designers.

The processes are either composite i.e. they are created by the composition of other processes; or leaf i.e. that are directly created using process constructors (the process constructors are related to the semantics' description. Each process in the network can belong to a specific MoC.

For example, in Figure 5.7, *p3* is made with the *mealySY* constructor, allowing under certain initial conditions this process to be carried out using the synchronous mealey FSM execution semantics. The processes *p1*, *p2*, *p3* have the same MoC A and are *leafprocesse*s within the composite process *p123* located at a higher level of abstraction. Similarly, *p45* consists of the leafprocess *p4* and *p5* with the MoC B. Between the composite processes, *MoCDomain* Interfaces (*di12*) are layers to explicitly adapt the signals belonging to the two environments.

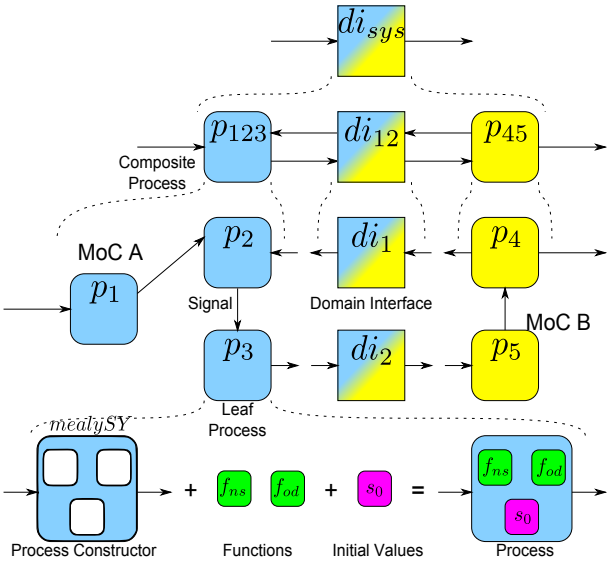

Figure 5.7: ForSyDe-SystemC sample model

Figure 5.8 shows the different layers that compose the ForSyDe BackEnd framework. The Layers will be used to go towards the synthesis of code on the HW platforms. The platform targeted in the design chain is a MPSoC, where all processors are connected via a Network-on-Chip (NoC). The NoC is implemented on an Altera FPGA.

The Backend Part uses a NoC generator to build platforms. The NoC generator takes an XML file as input describing the topology of the platform, its configuration, the specific properties, and the mappings of the various SW processes. This step was done manually before. In our tool chain we have contributed to the semi-automation of the generation of the XML file using the models described in Rhapsody and the intermediate ForSyDe-SystemC model. Spear is a Thales tool for the parallelization

of tasks and intensive multidimensional dataflow processing. The framework facilitates the architecture exploration for heterogeneous distributed architectures. It also allows the direct generation of code for an internal multiprocessor architecture on FPGA.

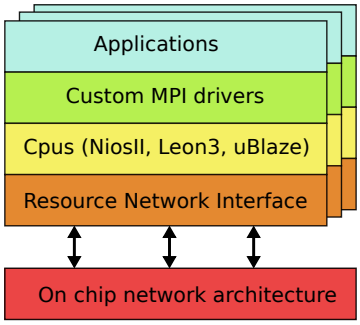

Figure 5.8: ForSyDe-BackEnd: Layered view of the platform

For the description of applications, Spear [116] uses a formalism defining components communicating through ports and connectors (excerpt Figure 5.9). The components have a vector defining the number of executions (loop). The multidimensional arrays are described at the application level by their shape and the elementary processing operations (Elementary-Transform). Each multi-dimensional data sets refers to the ports that produce it.

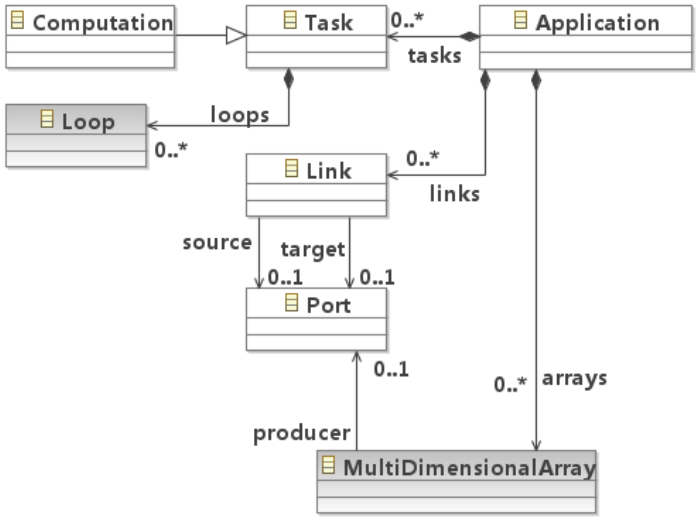

Figure 5.9: Conceptual Model of the *Spear Application Model* main concepts

5.2.2.3.c The envisioned Design Flow

Figure 5.10 describes the design flow for the Radar module with the sequence of activities and tools to perform these activities. This diagram hides several difficulties related to the formalisms of the tools and the semantics of the models and environments. Indeed, different metamodels are implemented in the tools, and the tools have different native semantics. To ensure the preservation of the semantics, the different semantic properties of models and tools must be taken into account.

5.2.2.4 Semantic constraints

Indeed, the models and tools have strong semantic properties linked to their functional requirements; while those for tools are related to the way in which tools interpret

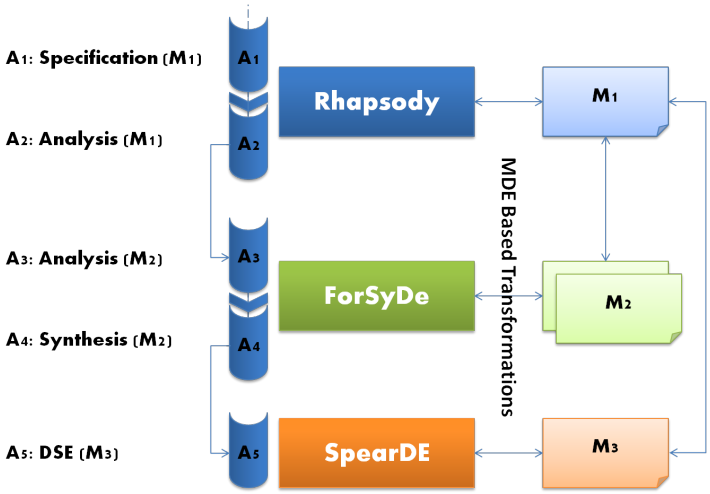

Figure 5.10: Tool Selection according to Design Process and Purpose

models i.e. *operational semantics* of the execution engines. To preserve the semantics of models between tools and through their execution engine, it is important to explicitly decline all semantics and examine their compliance.

The semantic properties of the models are specified during specification statically or dynamically. The dynamic semantics in particular reflect the inherent characteristics of scheduling according to the engineering domain i.e. (control-logic, signal processing, etc). Static and dynamic properties are based on the formal definition of the MoCs.

For most of the EmS design tools, these scheduling rules are implemented with the runtime engine and remain implicit for the user. Moreover, SLM tools (e.g. Rhapsody) runtime engine are not designed to take into account different patterns of scheduling rules for several particular engineering domain. In this case, the domain-specific semantics must be implemented to complete the system in the form of an intermediate layer between application models and the execution engine (using Cometa).

Being able to correctly transform the models requires ensuring that their semantics are correctly represented in the SLM tools and that the exchange of models between SLM and formal tools take into account sufficient information related to semantics (static, dynamic) for their correct interpretation towards each other.

Our approach is to identify the MoCs that are used for the representation of each module in the application, identify the tool MoCs, and provide an effective tool chain (Rhapsody, ForSyDe and Spear) preserving the semantics of the models.

5.2.2.4.a Explicit Model Semantics

Regarding the system, there are several functioning requirements that can be derived from the nature of the modules and their context of usage. In the *AntennaControl* module, the tasks performed by the Transmitter and Receiver are not interdependent and can therefore be carried out in parallel without any strong constraint for synchronization. Parallel execution semantics can be assumed by many execution semantics, including KPN.

The Receiver is however connected to the *BurstProcessing*. Given that the processing of the Bursts is done on a Burst by Burst basis, a synchronization handshake

between Receiver and BurstController semantics would achieve the Burst by Burst processing. In the literature, abstract semantics for the synchronous handshake-type of communication include the Hoare CSP previously described in Section 3.4.2. At high-level design steps, such abstract MoC can be used to describe the synchronization between the *AntennaControl* and *BurstProcessing*.

From the receipt of *echoedBurst*, the *BurstProcessing* module tries to provide the best estimate of radial speed, distance of reflectivity and signal-to-noise ratio of the potentially reached targets by applying different filters and computations.

BurstProcessing receives and processes the signals described in the form of multidimensional arrays. However, the computations and filtering only operate on a subset of the arrays, or only on certain vectors. In general, the size of the data that each function can process and take as input is different from the size of the data block that came as input in the *BurstProcessing* module. Furthermore, in order to build the final output data, different modules must read several parts of the arrays in parallel and rewrite one or more arrays of different size as output.

This description implies semantic choices to support such a type of behavior, knowing that all the sub modules can potentially be run in parallel (task parallelization). At the SLM level, it is important to be able to express these properties as early as possible, and also preserve such properties through all the system refinement process. There are two abstract semantics in the literature tackling the specification of properties for the intensive processing of multidimensional dataflow, MDSDF and Array-OL. We will focus on the Array-OL specification, MDSDF has many similarities with this semantics, and therefore only studying one of them is sufficient.

The Array-OL specification describes an application as a set of "parallelizable" task and data (with the data described on multidimensional arrays). In this specification, there are several static properties that make up the semantics e.g. *repetition space*, *patterns*, *paving*, *fitting*, *tiling*, etc. The idea of the Array-OL semantics is to parallelize the processing tasks, as well as the data exploitation (read, write) of data arrays. In fact, the specification includes two main definitions to exploit and process data:

- *Task parallelism* is obtained with the definition of the dependency graph where each node is a processing component.
- The *Data parallelism* is linked to the definition of the *repetition* component that has a *repetitionSpace*. A *repetitionSpace* sets the number of times a component is executed to fully exploit an array. These components build multi-dimensional arrays of different sizes specified by their *Shape*.

The Data extraction mechanisms are provided by the definition of *Tiler* vector attached to ports or connectors of the components. The *Tiler* is composed of an *Origin* vector (determining the starting position for extraction in an given array e.g. when reading the array / or the starting point to fill an array when writing in the array). The *Fitting* matrix determines the spacing between the selected items in the array, and the matrix *Paving* allows the *Origin* in each repetition of the component to be changed.

The above described semantics are requirements on how the system modules should interact to globally remain coherent.

Besides, we have the way in which the specification and analysis tools interpret the modules, which strongly depend on the execution semantics implemented on the tools and that can be different from the semantics described here. In the following, we will give semantics of the different chosen tools.

5.2.2.4.b Explicit Tool Semantics

The Rhapsody IBM Modeler provides runtime based on Discrete Event (*DE*) semantics. The exchanges between the components of the system are considered to be sequences of requests (corresponding to events) temporarily stored in queues (FIFO, LIFO, etc.). The execution *end* conditions are defined in the model's behaviors. As long as the condition is not reached, the scheduling behavior observes the elements of the queue to process, and updates the static values that can affect the execution and *end* conditions.

In ForSyDe, MoC semantics is defined by the process constructor only attached to the leaf process. These constructors are formally defined and stored in libraries. Their implementation is side-effect-free, which allows correct processes to be built. Most of the implemented semantics are based on the synchrony paradigm defined in Section 3.4. Several execution semantics are therefore built on top of this paradigm e.g. SDF, CT. In this purely synchronous approach, the MoC describes the abstraction of time and how the concurrent components communicate. The description of the different process constructors (e.g. *MealySY*, *CombSY*) in ForSyDe is provided in [145].

In our experiments, the goal is to show that the UML Rhapsody Modeler (specification and Simulation in C++) can be enriched with MoC-based operational semantic layers ensuring the execution of the models in a consistent manner within Rhapsody with respect to the functioning requirements. And, since ForSyDe offers a more formal analysis tool than Rhapsody, our benefits will be also to fill the semantic gap between Rhapsody and ForSyDe by providing these semantic layers facilitating the transformation of models between the tools. Several references have been defined in Cometa to match the related MoC definitions in ForSyDe in order to maintain the correctness of the behavior of the models within ForSyDe.

The other part of the design flow targets the SpearDE design tool. We have already stated that Spear implements the Array-OL semantics and presented the static properties. For the dynamic part of the semantics' description, the Array-OL semantics requires scheduling descriptions.

The scheduling of tasks depends on the topology of the system (directed acyclic graphs) that gives the dependencies between the components. Scheduling also depends on the expression of the data parallelism, where the number of times that each component should be repeated to produce or consume an array is given. Any scheduling that is able to take into account these constraints and to provide a consistent execution order is usable for execution. Nevertheless, this execution semantics is very similar to that provided by SDF, where data production and consumption rates are exploited to provided a scheduling of the components.

We can see that there are various semantics associated with the tools we are planning to use. In the first place, our motivation was only to select tools that can be candidate for several activities within the design process. Now, we see that, besides the ability of tools to carry out activities, there are often several different semantics that should be carried out in order to keep the accuracy of the models towards different tools. In the next section, we present the design flow implemented from the connection of the selected tools.

5.2.2.5 Analysis of the Semantic Compliancy between Tools

The study of the compliance between the tool MoCs can be used for different purposes. For instance, this can be useful to determine tool selections or to evaluate the feasibility

of an adaptation between the semantics of models or tools.

In Section 4.3.2.2, we gave the basic rules to effectively study the semantic links between the tools in a tool chain. In this part, we are implementing this technique to study semantic compliance between Rhapsody and the other tools of the chain. This study will allow the clarification of to what extent Rhapsody can be improved (through Cometa) not only for the internal simulation of semantics other than DE, but for enriching the models exchanged with other tools in the chain.

The mechanism to study the semantic compliance within a tool chain is similar for all the tools. Consequently, we will restrict ourselves to the description of the compliance relations between the semantics of Rhapsody and the Array-OL of SpearDE.

For the connection between Rhapsody and Spear, several questions must be asked by designers: what are the elements implemented in Spear to run the Array-OL semantics correctly that the Rhapsody UML environment cannot natively provide? - What are the elements that Rhapsody must integrate to run the Array-OL semantics and that Spear does not have?. For the first question, the missing elements are static semantic properties (e.g. repetition parameters, multidimensional data types, execution control mechanisms or scheduling policies). For the second question, Spear, a priori, already has all of the required concepts to express and run the Array-OL semantics; the problem comes from the adaptation efforts in Rhapsody to integrate the maximum of analyzable properties of Array-OL, knowing that once it is done, their translation to Spear is less time-consuming.

Figure 5.11 shows the respective association of the Spear and Rhapsody languages with their semantic domains when it comes to the execution of the models. The Languages L_{uml} and L_{spear} are mapped to their respective SD which are $MBSD_{DE}$ and $MBSD_{ArrayOL}$.

The Array-OL semantics have different flexibility and expressiveness level compared to DE. Indeed, as we described earlier, the semantics of DE is mainly based on the use of event and event queues for the communication between connected components. A *Scheduler* manages the queues and decides the transmission of the requests. Array-OL on the other hand, is more restrictive in terms of properties. The execution constraints are less flexible because they are specific to an engineering domain which is not the case for DE. In general, semantics related to tools are more flexible than the models' semantics because they are used as support for the models semantics. The tuples defined in Section 4.3.2.2 are used here to compare Array-OL and DE:

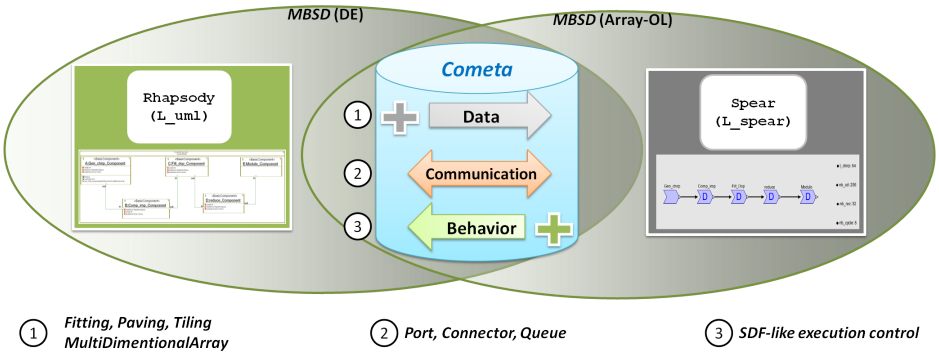

Figure 5.11: Rhapsody-Spear and the positioning of semantics.

- $D_{ArrayOL} \bigcap D_{DE} \neq \emptyset$. see ① in Figure 5.11. ArrayOL defines the multi-dimensional

data arrays. The data are read and written in parallel by the modules of the system. Such data structures do not exist natively in Rhapsody. However, the basic concepts of the UML (i.e. class diagrams) allow similar multidimensional data structures to be built. In fact, a precise description of the array features (sizes, number of vectors, etc) must be provided. Since it is possible to build the required data and their properties in Rhapsody, then we can also define a transformation on the data axis between Spear and Rhapsody UML.

- $B_{ArrayOL} \bigcap B_{DE} \neq \emptyset$. see ③ in Figure 5.11. The ArrayOL specification defines rules for the scheduling of concurrent entities. The scheduling properties require the management of the dependency relationships between entities taking into account the shape of the input and output multidimensional arrays and the vectors defining the number of executions authorized for each component. Any mechanism of centralized or distributed execution control respecting these requirements would describe and simulate the components with respect to the semantics.

  The above constraints were partly solved by another variant of dataflow-oriented semantics i.e. SDF. The proposed scheduling rule is based on the resolution of linear Diophantine equations [84] that solve systems of equations constructed on the basis of the relations between production and consumption rates of components on their I/O. In the Array-OL semantics, the production and consumption rates are provided by the Shape defined for each port. The resolution of the system of equation is not inconsistent with the use of events for communication between entities. One can define a resolution equation and encapsulated system data exchanged in events that will serve as a support for communication between components. Scheduling and control mechanisms are also describable by using a more abstract formalism such as Event-based FSM. Therefore it is possible to provide the execution control behaviors that reproduce the Array-OL semantics in Rhapsody. The execution control for Array-OL is presented in Section 5.3.1.

- $C_{ArrayOL} \bigcap C_{DE} \neq \emptyset$. see ② in Figure 5.11. With regard to the communication infrastructure, in Rhapsody DE and Spear communication is established using ports, connectors and storage entities available for the components and the scheduler. However, for the communication mechanism, DE components exchange events. Therefore, items that are stored in queues are events. This information is a constraint on the content of storage spaces. Hopefully, storing data in events is not problematic for the ordering of components. Moreover, in Cometa the formal description of events allows the addition of parameters on the events and, the parameters can represent objects such as data (e.g. multidimensional data). Therefore, it is possible to define a transformation that is an encapsulation of the arrays in events.

- $T_{ArrayOL} = \emptyset, and T_{DE} = \emptyset$. The Array-OL specification does not mention the description of the time as an essential and explicit concept for the description of the MoC properties and for the mechanisms to control the execution. For the sake of simplicity, we consider that this concern will not be addressed for our tool chain as this axis is not required for the model to be executable. It is still important to know that some implementation of schedulers can explicitly manipulate time concepts to control the execution of modules.

In light of our compliance analysis, the semantic domains are not disjoint at least from the data, communication and behavior points of view as shown in Figure 5.11.

In addition, Array-OL semantics by its characteristics is more restrictive than the semantics DE. As a consequence, DE is the support language that can help to express the Array-OL semantics.

One can now imagine the capture of semantics properties defining the necessary adaptations to maintain the Array-OL semantics of the models inside the Rhapsody tool. Thus, allowing the simulation of the model in a consistent way within the environment that natively does not have this type of semantics. In addition, the successive addition of Array-OL properties systematically reduces the efforts for future customization and interpretation of models in the target tool.

The Cometa models corresponding to the capture of the static and dynamic properties of Array-OL are presented in Section 5.3.1. We enrich models in Rhapsody with the captured execution control mechanisms to ensure the preservation of the rules imposed by the Array-OL.

The explicit definition of the $MBSD_{MoC}$ and their relationships (mapping) improves the reasoning on the definition of tool interoperability. It allows less focus on technical support for interoperability and promotes the possibility of taking decisions on the consistency and feasibility of certain connections between tool.

5.3 A Novel Design Flow connecting: *Rhapsody*, *Spear* and *ForSyDe*

In this section, we present the partial tool chain that allowed us to experiment with the use case. The tool chain connects Rhapsody IBM, Spear and ForSyDe Design and Implementation tools. As shown in Figure 5.12, the automated design flow is divided into three steps.

As a first step, we will present the automation phase providing the connection between Rhapsody and ForSyDe-SystemC. The objective of this connection is to ensure consistent specification, simulation and analysis of models in the two different environments. In the second step, we present the automation that has enabled us to generate files for the final HW synthesis using ForSyDe. Finally the third stage shows the connection of IBM Rhapsody and SpearDE tools in order to have Design Space Exploration in Spear and simulation of specific semantics in Rhapsody. Throughout the exchange, models remain consistent from the functional and behavioral point of view. The different model transformation rules will be illustrated in Figure 5.16.

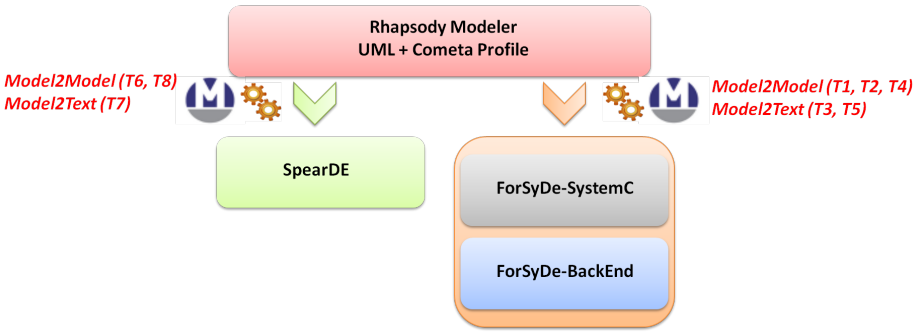

Figure 5.12: The Rhapsody-Cometa-ForSyDe and SpearDE Design Flow

5.3.1 Capturing Semantic Layers for the Design Flow

In this section, we present the use of Cometa to retrieve the properties of the domain $MBSD_{ArrayOL}$.

The captured properties focus on the 3 axes where points of compliance were found i.e. Communication, Behavior and Data. As shown in Figure 5.13, we add an additional view that corresponds to the topology of the semantic layers. The MoCComponent parameters are used to capture the *repetitionSpace*, and the Array-OL control behavior is attached to the components of the semantic layer. We will see the description of the behavior in the following sub-section.

The concepts abstracted in the Cometa data concern are used to capture specific data properties: the concept of *Matrix* is used to capture the vectors that explore data arrays (e.g. *Tiler*, *Fitting* and *Paving*); The concept of *Vector* is used to capture the data sizes accepted on each port Vector (e.g. *Origin*, *Shape*); and the concept of *Parameter* for the capture of the repetition space (*Parameter* associated to *repetitionSpace*).

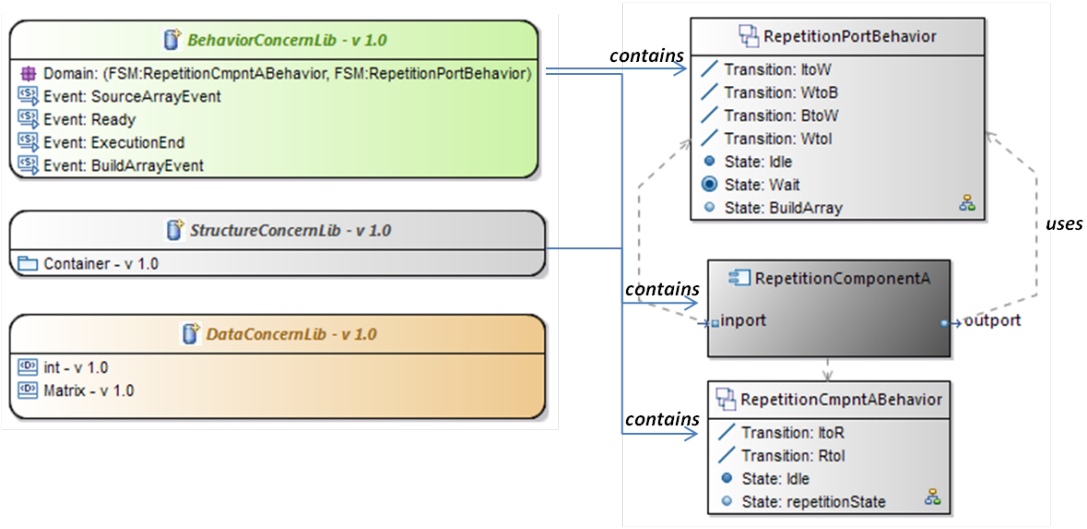

Figure 5.13: Array-OL semantic domain (static and operational) capture in the Cometa DSML

The execution control mechanisms captured with Cometa (Behavior) is close to the SDF scheduling policies, applied to multidimensional data. The control behaviors are distributed; this means that it does not define a global scheduler to manage the sequences of executions as would a traditional SDF static scheduler. Therefore, the execution control behaviors are attached to the communication elements such as ports, connectors, etc. The communication events created (ReadEvent, WriteEvent) have parameters to contain the multidimensional data. In addition, the shapes of the data are placed on the communication elements (port, connectors). This variation on the placement of the control shows new possibilities of express scheduling of parallel tasks.

In Cometa, execution control mechanisms are described with three FSMs attached to the BasicComponent and the MoCPort (IN/ OUT). The FSMs are presented in Figure 5.14 and describe the behavior of components and ports to process the data. The mechanism of *Tiling* is placed on the ports.

- the FSM attached to the BasicComponent has 2 States: Idle and repetitionState. In the Idle state, the component waits to be notified by the MoCPort (IN) of the arrival/ availability of data. On receipt of the notification, the *BasicComponent* fires n extraction requests. The value of n depends on the product of the values defined in the repetitionSpace vector. The items selected in the arrays at each extraction are also a function of the values of the repetitionSpace vector and the matrices fitting, paving, etc.
- the behavior of MoCPort (IN) has three States: Idle, Wait and BuildArray. In Idle mode, the port is waiting for data. Upon receipt of a data array, the port produces notification to the *BasicComponent* and waits for a response (Wait state). After receipt of the request for the extraction of data from the *BasicComponent*, the port uses the *Tiling* matrix to retrieve samples of data on the input array.

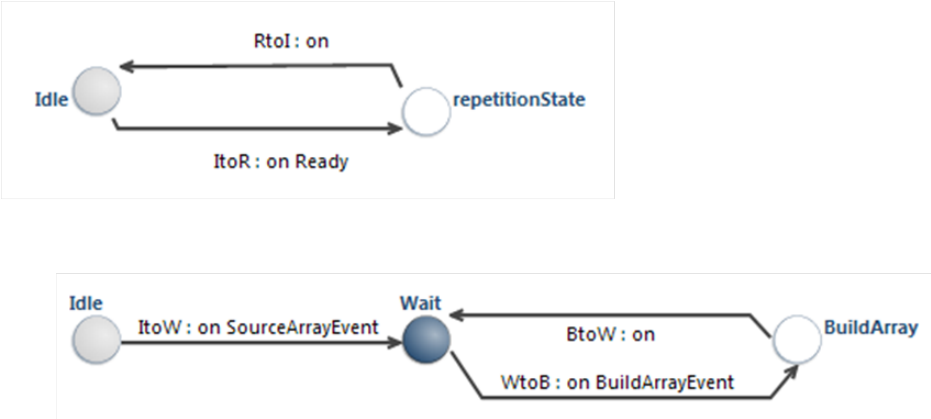

Figure 5.14: Excerpt of the Execution control FSM for Array-OL in Cometa Modeler: (up) *BasicComponent*, (down) *MoCPort* FSM.

In the same way, a generic FSM is defined to build the output arrays of data on (OUT) MoCPort. These descriptions of FSMs are generic and can be reused in multiple environments and for the different topology of semantic layer.

The Figure 5.15 example represents an abstract description of the exploitation of the defined FSMs.

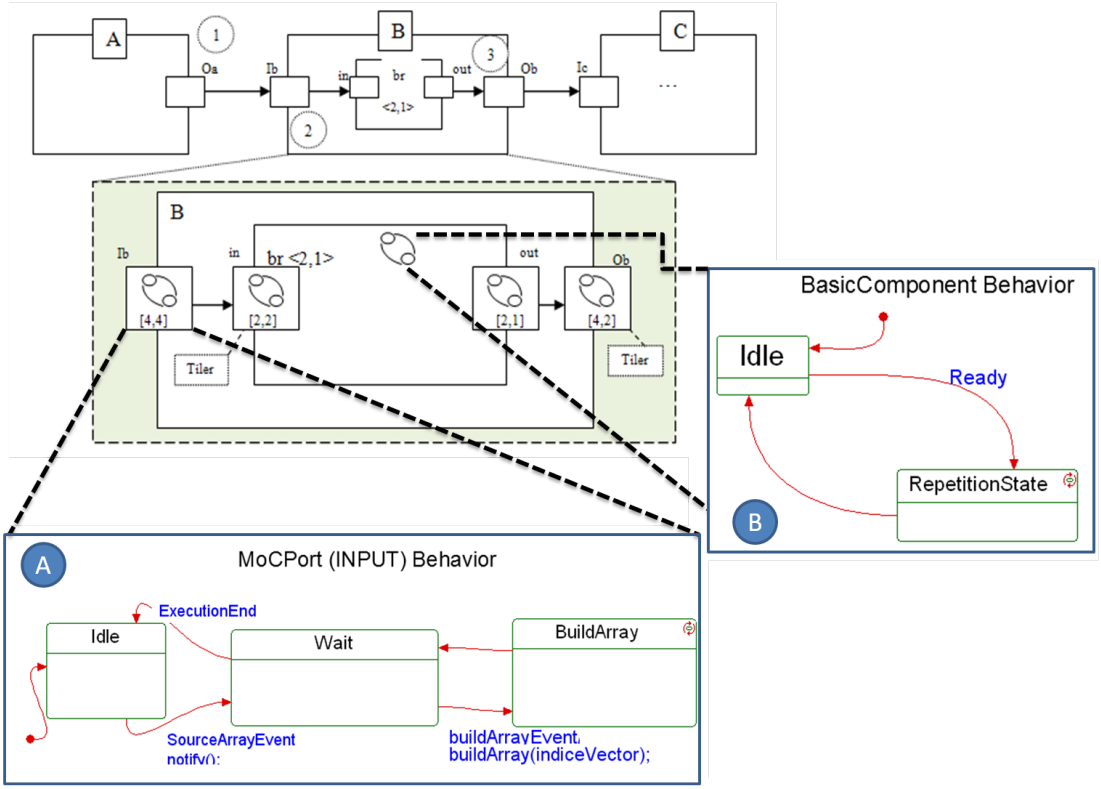

Figure 5.15: Example of 3 inter-connected components with Array-OL semantics

A, B and C are composite components, and *br* is a *repetition* component which can be seen as an atomic computation unit. In Phase ① the *Scheduler* enables the execution of component A, which produces an array of a predifined size on its port *Oa*. The array and its size are defined using the metaclass *Array* of Cometa. The data are received by the port and sent to $br \rightarrow in$ in the subcomponent *br*. In Phase ②, the port $br \rightarrow in$ has a generic behavior and mechanism of extracting patterns (*Tiler*). Once the Array is received, it notifies the state machine of the *br* component that data are available. The *BasicComponent* after receiving the notification will run 2 times (2x1) every execution, it will send to $br \rightarrow in$ request for the construction of a sub-array and produce a sub-array output on $br \rightarrow out$. In Phase ③, the sub-array output is received by the port *Ob* which also has a generic behavior and Tiler mechanism. At each receipt of a sub-array, using the Tiler, *Ob* places the elements of the sub-arrays on a defined position in the output array. Once all repetitions of *br* are reached, an output array is built and sent to *Ic* from *Ob*. *Ic* on receipt of the Array, continues processings.

5.3.2 Weaving Cometa Models with IBM Rhapsody

The mechanisms of allocation/encapsulation are often handled by different languages, which can be a source of difficulty. Indeed, the semantic layers are entirely made in Ecore, while the source and target models may use other formalisms and formats. This leaves several interpretations on how source models integrate with Cometa models:

1. The first solution is a complete translation of the source model to the Cometa formalism. In Cometa, this is manifested by the use of the concept *Block* associated with the *BasicComponent* to incorporate part(s) of the source model knowing

that each application block is associated with a *BasicComponent*. *Block* will integrate the content/ functionality of the application block and its interpretation is delegated to a given runtime capable of parsing the model in its entirety. In other words, the formal semantics implemented in the target tool must be able to interpret the transformed allocated model; in this case, each concept and relationship in the allocated model has semantic equivalence in the formalism of the language of the target language.

2. The second solution is a full integration of the Cometa models in the source environment with the source formalism, thus producing a uniform allocation model in this environment. In this case, the formalism of the source environment should be sufficiently expressive to allow the integration of semantic layer models. The UML is one of the formalisms offering such flexibility. With good experience of the MoC theory and the UML formalism, the designer can directly describe the semantic layer libraries in the UML language. In this case, the allocation mechanism remains the same.

   From the allocated Cometa model, the designer can define appropriate transforms to the target formalism. Besides, if the source environment has an engine that can interpret the model, then the allocated model allows the integration of new execution semantics based on natively undefined MoCs in this environment.

3. The third solution is based on the combination of the two input formalisms i.e. source model formalism and semantic layer formalism (Ecore). In such cases, the *Block* concept represents an extension point to the part of the source model to which it is bound (a given application block). This extension link, with the semantic model, shapes the allocation model. In fact, the link serves as a bridge between the two input formalisms to parse and to transform into a target model. All the concepts which are processed must naturally find their semantic equivalence in the target formalism to be relevant.

The process described above explains how the execution control models are combined with application models to form a unique model preserving the semantics.

5.3.3 Connecting the Tools within the Tool Chain

Rhapsody integrates different SLM types of diagrams to describe the structure of the systems, their functionality, and the types of data exchanged between entities (e.g. class, structure diagrams). As we previously explained, in this environment, the execution engine is based on the DE semantics, which means that all models produced in the environment will be executed according to DE semantics.

In order to integrate new semantics of the other tools, we integrate the semantic layers defined by Cometa. The semantic layers can preserve the execution logic induced by the engineering domain of other tools and based on the theory of computational models. Cometa can be used in two different ways at this level. Either as built-in profile in Rhapsody to specialize the Rhapsody designed models; or by direct reuse of Cometa models captured in external libraries.

The description of Cometa models in Rhapsody uses UML classes and stereotypes (annotations UML) to describe the hierarchical components. The links for application block allocation are defined and associated to the Cometa components controlling their execution. At the operational level, since we are using FSMs, this link also highlights the facts that the blocks generate events that are received and controlled by the semantic

layer. The control behaviors (FSMs) are hooked to the components in the form of StateCharts without hierarchical states. Once again, the StateCharts are obtained either by transformation of the Ecore FSMs or by their internal implementation in Rhapsody.

With the fully allocated model, the bulk of the work is then to build a mapping table between the concepts of the allocated model and the concepts in the target tool. Because the semantic layers are representations of missing semantic information from the source model based on the semantics of the target model, all the concepts used in this sense are translatable to the target tool.

To connect the tools of the the design flow, several transformations were defined to ensure the translation of the UML-based enriched models to various representations specific to ForSydeSystemC, ForSyDeBackend and SpearDE environments. The diagram below is a summary of the different steps of model transformation which have been made; its purpose is to situate the upcoming extracts of transformations that will be shown.

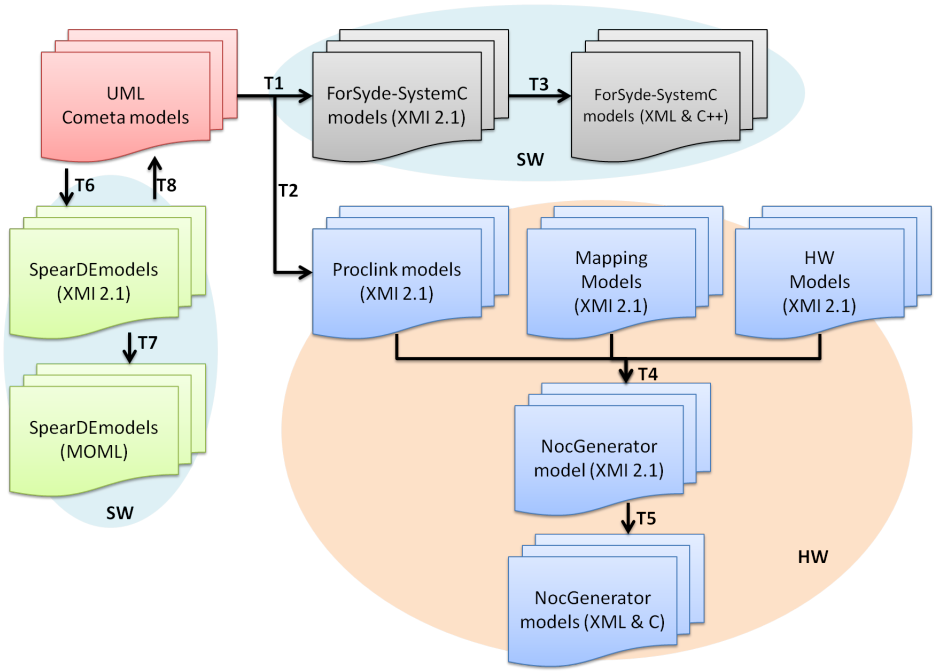

Figure 5.16: Overview of the implemented Transformation rules with MDWorkBench

From enriched UML models (with Cometa libraries), the transformations t1 and t3 allow the production of ForSydeSystemC files used for application analysis in ForSyDe. The t2 transformation rule produces a duplication of the application model generated after t1 while filtering out useless information for the definition of the NoCGenerator model. Indeed, this file (called ProcLink) only contains the names of the application processes and their relationships in terms of the MoC that they implement and their sources and targets. The t4 transformation takes the simplified model (ProcLink) and two other input models i.e. the HW architecture description model and the HW/SW mapping rule model to provide an output file representing the allocated NoC architecture. We will see excerpts from t1 and t4 in the next sections.

Besides, the t6 and t8 transformations are rules to translate the UML/Cometa models to the formalism of Spear and vice versa. The set of transformations are available

in [186].

5.3.3.1 Connecting Rhapsody and ForSyDe-SystemC

In this section, we describe the implementation of the connection between Rhapsody and ForSyDe dedicated to the formal analysis i.e. ForSyDeSystemC.

5.3.3.1.a Overview

Our goal is to consistently integrate while preserving the behavior of models.

In order to generate the SystemC models of the system using another language, ForSyDe-SystemC provides an intermediate representation combining the XML formalism and the C++ programming language. The XML files describe the hierarchical structure of the system, while all of the C++ files make up features and libraries of pre-implemented MoC constructors in ForSyDe.

The addition of a semantic layer in Rhapsody is an independent process. Once it is done, we offer automatic transformation mechanisms to transform the allocated model into this intermediate structure representation.

The MDWorkbench transformation tool has a connector to Rhapsody allowing the exploitation of the models produced in Rhapsody. In order to be fully model-based, we defined a metamodel for the ForSyDe intermediate format A.1.2. In this DSML, we abstract the concepts of ProcessNetwork, CompositeProcess, LeafProcess, etc. The DSML is used to produce the model corresponding to the Rhapsody allocated model. From the intermediate model generated, we apply a second transformation to produce the final XML and C++ files.

The alignment between the ForSyDe design concepts and those of Cometa was made taking into account the structural similarities such as the hierarchical components (composite), atomic components, ports, etc.

Then, taking into account semantic similarities i.e. semantic roles associated with execution control behaviors and layers that correspond to process constructors and signals. The categories of roles that can be associated with a component of the Semantic Layer were introduced in 5.2.2.4.b and correspond to the different process constructors detailed in [145]. For example, the named concept of ProcessConstructor is equivalent to Cometa named Behaviors that have a role "BehaviorScheduler". To reuse the implemented MoC libraries of ForSyDe, the name of the corresponding Cometa behaviors are parsed and used only to index their equivalent implementation in ForSyDe. These basic alignment guidelines are implemented as model transformation rules between Rhapsody and ForSyDe-XML.

5.3.3.1.b Transformation rule patterns: Rhapsody/ ForSydeSystemC

MDWorkbench allows imperative transformation rules to be set that are similar to Java code used to parse a Rhapsody UML model, and build other models or generate text (e.g. code generation xml, C++).

For Rhapsody and ForSyDe, the parsing mechanism adopts the following algorithm: all the *BasicComponent*s are transformed into *LeafProcess*, so are all the *CompositeComponent*s transformed into *ProcessNetwork*s, *MoCConnector*s into *Signal*s, etc. Similarly, the behaviors with the role "BehaviorScheduler" (formerly "MoCOrchestrator") associated with the Cometa components are analyzed; each role, and its associated

behavior settings, is reused to refine the LeafProcess defining their process constructors and specifying their parameters.

In Listing 1, we show an extract of the rule to transform an entry UML model into a ForSyDe-XML compliant output model. Several rules based on the same approach have been developed to translate the entire system model [186]. The resulting model is a ForSyDe-XML model containing the structure of the system and all its artifacts required to generate XML files and also a set of C++ files containing the functionalities.

```
private rule manageCompositeComponentContent (component : CompositeComponent, forsydeProcess
    : ProcessNetwork)
{
var allcomposites : MDWList = component.getInstances("CompositeComponent");
var allbasics : MDWList = component.getInstances("BasicComponent");
foreach(composite: cometa.CompositeComponent in allcomposites)
                {
                        var newProcessNet : forsyde.ProcessNetwork = createProcessNetWork(
                            composite.Name);
                        @manageCompositeComponentContent(composite, newProcessNet,root);
                        root.processNetworks.add(newProcessNet);
                }
foreach(basic : cometa.BasiCComponent in allbasics)
                {
                        var newLeafProcess : forsyde.LeafProcess = createLeafProcess(basic.
                            Name);
                        @manageBasicComponentContent(basic, newLeafProcess);
                        forsydeProcessN.leaf_process.add(newLeafProcess);
                }
                ...
}
```

Listing 5.1: Excerpt of Transformation Rule in MDWorkBench for composite components

In order to obtain the final SystemC model, a second code generator takes the ForSyDe XML model and C++ files as input to produce a ForSyDe executable model. The code generator (based on Extensible Stylesheet Language Transformations XSLT) was provided by the developers of ForSyDe-SystemC.

5.3.3.2 Connecting Rhapsody and ForSyDe-Backend

In this section, we present the connection of the Rhapsody tool with the BackEnd of ForSyDe, while taking into account the structural model produced for the formal analysis (i.e. the ForSyDeSystemC model).

5.3.3.2.a Overview

In the second integration phase, our goal is to automatically reach the description of NoC architectures targeting different implementations of multicore platforms.

The automation of the NoC architecture generation step requires the use of several models as input to produce the NoC architecture as output.

The input models are: the intermediate XML ForSyDe system model, a model describing the NoC HW architecture, as well as a model describing the mapping rules of the system processes on the architecture. Metamodels defining the last two models were created for use in MDWorkbench.

The mapping DSML defines associations from the system model processes to the nodes described on the HW NoC architecture. The NoC architecture model (see appendix A.1.4) defines networks of nodes and the physical characteristics of the platform.

From the three entry models, several transformation rules were defined to generate the allocated NoC architecture and the C processes corresponding to the computations integrating the semantic properties.

5.3.3.2.b Transformation rule patterns: Rhapsody/ ForSydeBackEnd

In this section, we present an excerpt from the rules defined for the production of the architecture NoC model (NoCGenerator File). This transformation corresponds to t4. The listings 5.2, 5.6, 5.4, show 3 extracts of transformation rules.

In the first excerpt, we use both @manageAllHardwareElement and @manageSoftwareMappings extraction rules. The model resulting from the execution of these two rules is operated by the code generator defined in $xmlGenNoCGenerator (t5) producing the final XML file representing the NoC allocation architecture model.

```
entry rule main()
        {
                // Create the Noc_Generator Root
                var hardwareSystem : hwmetamodel.System = hwmodel.getInstances("System").
                     first();
                var rootElement : forsydebackend.System = @Initialization(hardwareSystem.name
                     );

                // Generating all C files for Backend
                var source : process.AllProcess = proclink.getInstances("AllProcess").first()
                     ;
                var procnets : com.sodius.mdw.core.model.MDWList = source.getProcesses();
                foreach(procnet :process.Process  in procnets)
                {
                        if(procnet.filename !=null)
                        {
                                $cGen(procnet);
                        }
                }

                var processLinkDefinition: process.AllProcess = proclink.getInstances("
                     AllProcess").first();

                // Generate all Elements of Hardware in Noc_Generator
                @manageAllHardwareElement(hardwareSystem, rootElement);

                // Use Mappings to get the software in noc_generator
                @manageSoftwareMappings(mapmodel, processLinkDefinition, rootElement);

                // Call function to generate the template
                $xmlGenNoCGenerator(rootElement);

        }
```

Listing 5.2: Excerpt of Transformation Rule in MDWorkBench for NoC model generator

The @manageAllHardwareElement rule, as shown in the second excerpt (listing 5.6), takes as input an HW model and reproduces its entire contents in the output NoC architecture model represented by its entry point “rootElement”.

```
private rule manageAllHardwareElement (hw : hwmetamodel.System, root: forsydebackend.System)
        {
                var systParams:com.sodius.mdw.core.model.MDWList = hw.getParameters();
                var hardHard: hwmetamodel.Hardware = hw.getHw();

                foreach(systParam : hwmetamodel.Parameter in systParams)
                {
                        var newNocGenPram : forsydebackend.Parameter = @NoCGeneratorFactory(
                             nocgen).NoCGeneratorParameter(systParam.name);
                        newNocGenPram.setValue(systParam.value);
                        root.params.add(newNocGenPram);
                }
                if (hardHard!=null)
                {
                        var newNocGenHard : forsydebackend.Hardware = @NoCGeneratorFactory(
                             nocgen).NoCGeneratorHardware();
                        @manageNoCGeneratorHardwareContent(hardHard, newNocGenHard);
                        root.hw = newNocGenHard;
                }
        }
```

Listing 5.3: Excerpt of Transformation Rule HW exploration

For any property described in the input HW model, then the rules @NoCGeneratorFactory(nocgen).NoCGeneratorParam and @manageNoCGeneratorHWContent sub

rules duplicate the property in the NoCGenerator model (more detail on these rules are provided in the [186]. Once the HW model is fully reproduced, the last excerpt (Listing 5.4) shows the algorithm to operate the mapping rules. The mapping rules are then used to describe the allocation of the processes on the nodes of the HW architecture in the NoCGenerator model.

```
private rule manageSoftwareMappings (ma: mapping, proclinking: process.AllProcess,
    rootElement:forsydebackend.System)
        {
                var mapp : mapping.Mapping = mapmodel.getInstances("Mapping").first();
                var maprepo: mapping.Repository = mapmodel.getInstances("Repository").first()
                    ;
                var maps : com.sodius.mdw.core.model.MDWList = mapp.getMapProcess();
                var processes: com.sodius.mdw.core.model.MDWList = proclinking.getProcesses()
                    ;
                if (maps!=null)
                {
                        //
                        var soft : forsydebackend.Software = @NoCGeneratorFactory(nocgen).
                            NoCGeneratorSoftware();
                        foreach(map:mapping.MapProcessType in maps)
                        {
                                foreach(process:process.Process in processes)
                                {
                                        if(map.name.equals(process.name)==true)
                                        {
                                        @manageNoCGeneratorSoftwareMaps(map, soft, process);
                                        }
                                }
                        }
                        var paramRepo : forsydebackend.Parameter = @NoCGeneratorFactory(
                            nocgen).NoCGeneratorParameter(maprepo.name);
                        paramRepo.setValue(maprepo.path);
                        soft.swparams.add(paramRepo);
                        rootElement.sw = soft;
                }

        }
```

Listing 5.4: Excerpt of Transformation Rule @manageSoftwareMapppings

@manageSoftwareMappings allows the addition of an instance of SW in the target NoCGenerator model. For each process declared in the mapping model, if the process also exists in the application model of the system (i.e. ProcLink model), then the rule @manageNoCGeneratorSoftwareMaps allows the addition of the allocation of a process on the "Node" to which it is linked and provides information on this process such as: its MoC, its sources and targets. An excerpt of the NoC model produced for the BurstProcessing module will be shown in the coming sections.

5.3.3.3 Connecting Rhapsody and SpearDE

This last integration step aims at describing the connection between Rhapsody and Spear.

5.3.3.3.a Overview

The connection between the two tools is bidirectional. Thanks to the semantic properties provided by Cometa, we make consistent semantic links between the properties contained in the UML/Cometa model and those present in the Spear formalism. Indeed, with the semantics compliance analysis performed on both languages (i.e. Spear and Cometa), mapping tables between their concepts has been established including structural and semantics details.

With regard to the transformation rules, in one direction, the input Rhapsody models are transformed into Spear XMI models (t6). The models generated are further

successively transformed (t7) to generate Moml code corresponding to the formalism accepted as input for the Spear tool.

The t8 rule produces the opposite result by taking a Spear model as input and outputs a UML/Cometa model incorporating all of the manageable static semantic properties in Rhapsody (i.e. Array, repetition vector, structure, etc.). The idea is to show that models from Spear to Rhapsody can also be simulated using the Cometa execution control models included in the Rhapsody Modeler. The models produced in Rhapsody are combined with the FSMs described for the Array-OL execution semantics.

5.3.3.3.b Transformation rule patterns: Rhapsody/ SpearDE

We have implemented the following two transformation rules: the first rule translates a Rhapsody allocated model into a Spear model, the second transformation defines translation mechanisms from a Spear model towards a Rhapsody model.

- *umlCometa2spear*: for this transformation step, the transformation of the topology is trivial as for the other transformations. This triviality stems from the fact that most codesign system architecture description languages are based on an ADL approach for component description. This is also the case of Rhapsody and Spear. To transform the multidimensional data models, the rule parses the data structures defined in Cometa, the matrices, and builds their equivalence using the concept in the Spear basic language (see Spear metamodel in appendix A.1.1). Thus, the final Spear model integrates data properties that natively did not exist in Rhapsody, but were added with Cometa data concern.

- *spearCometa2uml*: Similarly, the description of the application topology elements are trivially from Spear to Rhapsody. The Spear parts intrinsically linked to the static Array-OL semantics are transformed and added into a semantic layer defined using Cometa. All the static properties such as repetition vectors, the sizes of array, data types, etc are attached to their supports in the semantic layer.

  When it comes to the execution control part, Spear does not provide an explicit implementation of the mechanisms for scheduling tasks. Therefore, only axes "data" and "communication" are concerned by the transformation. The execution control behaviors are taken from the operational MoC libraries captured with Cometa (inside or outside Rhapsody). The behaviors are mapped on their support on the semantic layer.

The first transformation steps produce XMI models that conform to the metamodel of Spear (see appendix A.1.1), the models are further transformed into MOML code which complies with the format accepted by the Spear tool.

The rule @getSubModule takes a UML/Cometa model as input; the model is parsed to retrieve multidimensional data arrays, tasks (Computation) structures and their relations to define the topology of the system. Each Computation retrieves its repetition vector determining the number of executions to fully consume or produce arrays of data.

As shown in listing 5.5, to retrieve the topology and relations, @manageInstancesInStruct and @manageRelationsInStruct rules are used. The rule @manageInstancesInStruct parses each structural element and creates a new "Computation" in the Spear output model. For each Computation created, the parsing uses sub rules such as @manageOperations to retrieve the functions, the unique ID of the Computation and

semantic properties such as the Loop (repetition vector); @managePorts retrieves the links between the computations and specifies the arrays produced by the ports.

```
/**  Getting the Sub-Modules**/
        private rule GetSubModule(pack : rhapsody.Package, root : Spearlite.modeling, app:
             Spearlite.application) : Spearlite.application
        {
                // Getting Arrays
                if(pack.name.equals("DataTypes")==true)
                {
                        // Getting Arrays
                        @ManageArrays(pack, app);
                }
                // Getting Module Elements
                if(pack.name.equals("RadarSystem")==true)
                {
                        var structs      : com.sodius.mdw.core.model.MDWList = pack.
                             getStructureDiagrams();
                        foreach (struct : rhapsody.StructureDiagram in structs)
                        {
                                if(struct.name.equals("CompositeComponent")==true)
                                {
                                        // Getting the Parameters
                                        app = @ManageHighLevelProperties(pack, app,struct.
                                             name);
                                        // Getting the Computations
                                        @manageInstancesInStruct( struct, app);
                                        @manageRelationsInStruct(struct, app);
                                }
                        }
                }
        }
```

Listing 5.5: Excerpt of Transformation Rule for Spear

Listing 5.5 also shows how the algorithm explores and recovers the arrays of data created in Rhapsody from the Cometa libraries. The @GetSubModule parses the directory containing the data structures and restores all of the properties related to data structures and their exploitation. In the @manageArrayrs rule, the data are created, and for each data array, the rule gets through the corresponding UML structures to refine the output data arrays produced in Spear. In the experiment, we show an excerpt of the Spear model corresponding to the BurstProcessing module.

```
/**  Retrieving Application Arrays  **/
        private rule ManageArrays(pack:rhapsody.Package, appl: Spearlite.application):
             Spearlite.application
        {
                // Getting Arrays
                var classes: com.sodius.mdw.core.model.MDWList = pack.getClasses();
                                foreach(class: rhapsody.Class in classes)
                                {
                                        if(class.getStereotype().name.equals("Array")==true)
                                        {
                                                var speararray: Spearlite.
                                                     MultiDimensionalArray = @SPEARFactory(
                                                     outmodel).createMultiDimensionArray(class
                                                     .name);
                                                @ManageArrayValues(speararray, class);
                                                appl.arrays.add(speararray);

                                        }
                                }
        }
```

Listing 5.6: Excerpt of Transformation Rule in MDWorkBench for Array exploration

5.3.4 Metrics

The efforts to improve the UML models to obtain the maximum number of model semantic properties in different environments have allowed us to establish qualitative relationships between the core modules and enriched models. If we restrict ourselves to the subset of targeted tools i.e. Rhapsody, ForSyde and Spear, one can work on

Table 5.1: Concepts taken into account during transformation of basic models according to MoC Criteria

Transformation/MoC Criteria	Data	Communication	Structure	MoCBehavior	Time
UML to ForSyDe	Yes	Yes	Yes	No	No
UML to Spear	No	Yes	Yes	No	No
ForSyDe to UML	Yes	Yes	Yes	No	No
Spear to UML	No	Yes	Yes	No	No

two evaluation results for assessing the improvements that are made. Here are the two possible analyses of the situation:

- An analysis starting from a basic non-enriched model that was converted in ForSyDe and Spear environments
- An analysis in the case of enriched models in terms of MoC references and MoC behavior which is translated to these environments.

Tables 5.1 and 5.2 highlight the elements supported during the transformation in both analysis cases. The criteria taken into account are the ability to transform information from one environment to another, taking into account the criteria related to Data, Time, MoCBehavior, Structure, and Communication.

The MoCBehvaior criterion gathers all the entities that can contain a reference and/or MoC behavior to control the execution of the entities.

In the above analysis Table 5.1, a basic application model presents a number of concept that can be translated into ForSyDe. Regarding the concepts of Structure, Data and Communication, a consistent semantic interpretation can be found in ForSyDe environment. However, the model would be necessarily incomplete since the properties related to the MoCBehavior, Time are not expressed in the UML model and therefore not taken into account. The main consequence of the lack of description of the MoCBehavior is that the process constructors are not specified which prevents any possible interpretation of the model in the target environment.

For the translation of the application model to Spear, the Structure and Computation criteria have a semantic equivalence between the two environments. The lack of elements related to the MoCBehavior and Data implies that application models in the source environment cannot be executed properly according to the Array-OL semantics of Spear. Thus, no coherent analysis is made at this level without the MoCBehavior and Data properties. In addition, the data natively described in the UML model are different from those handled in Spear. In order to further simulate the models in Spear, a refinement of the Spear model is needed to complete the missing properties and data structures. The Time criterion is not necessarily significant for this specific case of semantics i.e. Array-OL.

For the inverse transformations (Spear to UML and ForSyDe to UML), the same problems arise on the same criteria. In the case of ForSyde, without the Cometa libraries, the basic UML model produced cannot be interpreted since there is no consistent execution semantics implemented or integrated i.e. the execution control mechanisms, nor the expression of time.

In the case of Spear, the Scheduling implementing the control mechanisms according to Array-OL, as well as the data arrays cannot be supported directly in Rhapsody without the Cometa data and execution control models.

Table 5.2: Concepts taken into account during transformation of enriched models according to MoC Criteria

Transformation/MoC Criteria	Data	Communication	Structure	MoCBehavior	Time
UML/Cometa to ForSyDe	Yes	Yes	Yes	Yes	No
UML/Cometa to Spear	Yes	Yes	Yes	Yes	No
ForSyDe to UML/Cometa	Yes	Yes	Yes	Yes	No
Spear to UML/Cometa	Yes	Yes	Yes	Yes	No

In the second analysis (Table 5.2), we can see that the concerns not taken into account previously are added by Cometa. When the transformation starts from Rhapsody to ForSyDe and Spear, respectively UML models are enriched with the MoCBehavior and Data concerns i.e. description of the MoC ProcessConstructor and their settings. For Spear, the concerns related to data structures are taken into account. The mechanisms for execution control defined by the Cometa FSM are not transformed into Spear since their implementation already exists in the Tool.

In reverse, all the elements handled in one transformation direction, may be processed the other way around taking into account the same criteria and transforming the same elements. However, the Cometa execution control mechanisms will replace the MoC implementations based on the definition of MoC ProcessConstructor or the Spear *Scheduler*.

For Spear to UML/Cometa transformations, all the criteria are taken into account except for Time. For the MoCBehavior criteria, as for ForSyde, the execution control mechanism is provided by Cometa models and the data description are also taken into account with the Cometa multidimensional data types libraries.

5.3.5 Burst Processing System Design and Analysis

In this section, we present the experimentation and the prototypes that have been made in the context of designing the Radar System. This experiment highlights the results obtained using the Cometa models presented in the previous sections. In the UML Rhapsody Modeler, the Figure 5.17 presents the Cometa profile and the libraries of MoCBehvaior which have been integrated into the tool to allow model enrichment. These elements are used to add MoC references via the profile and MoC-based operational semantics to the entities that are handled.

The entire Radar model is not shown in this section. Our approach is rather to provide a representative subset showing the parts where the Cometa models were used, thus emphasizing analysis by simulation in Rhapsody. After simulation, the transformations presented earlier are used to generate the output models for ForSyde and Spear environments taking as many as possible of semantics properties in the transformations. Several examples of generated models will be shown and the properties preserved throughout the analysis and transformation chain.

Considering the different criteria for the preservation of semantics, we can start by specifying the data for the system (Figure 5.18).

The data in the system are mostly of multidimensional type. Their description in Rhapsody is done through the concept of *Class* for which the various dimensions are identified in the form of attributes of the *Class*. As a result, the samples of data types *Beam*, *Burst*, *Plot* have several dimensions. For the case of Burst, the vector has three dimensions which are NB_PULSE, NB_RANGE_CELL and NB_ANTENNA. These data types are identified and referenced as Cometa *Array*. The reference makes

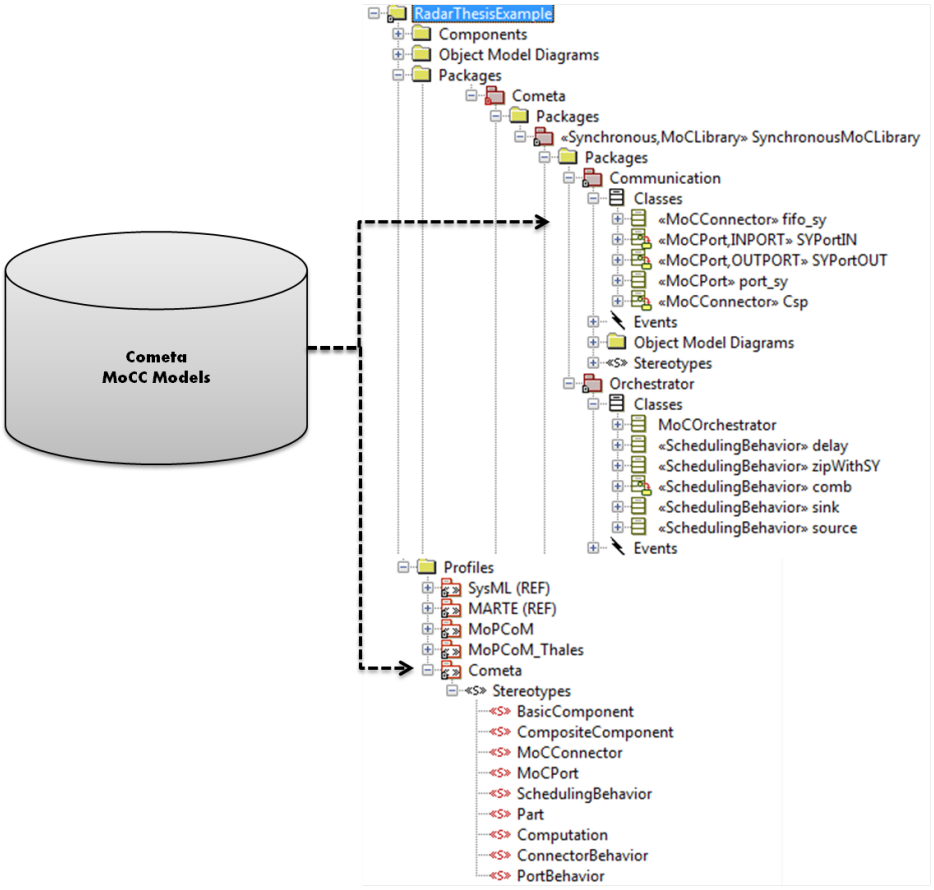

Figure 5.17: Cometa Libraries in the UML Rhapsody Modeler

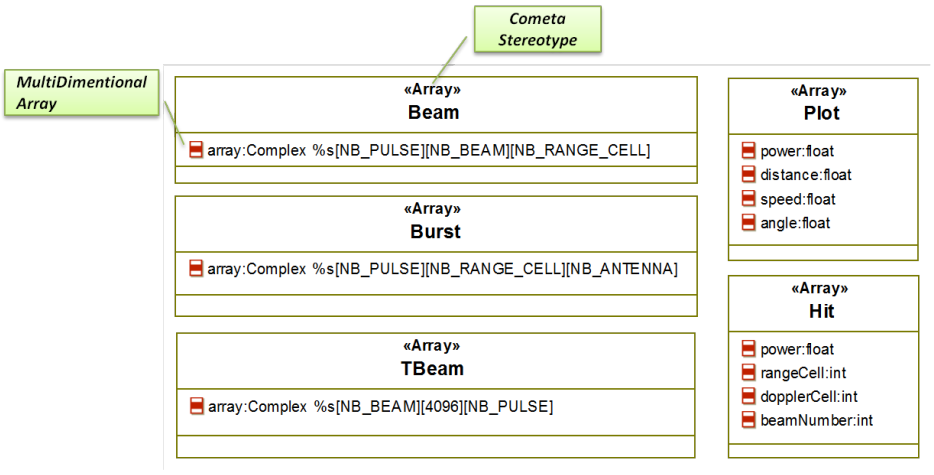

Figure 5.18: MultiDimentional Data Arrays in Rhapsody

the difference between normal data types and those that are multidimensional.

During the transformation steps, all the UML *Class* with a stereotype *Array*, are parsed to retrieve their specificities (size, dimensions) that will be reused and transformed into their corresponding elements in the target tools. These data type excerpts are used for processings in the *BurstProcessing* module.

As specified by the Array-OL semantics, since each vector has several dimensions and different sizes, the execution of modules depends on the scheduling and execution control mechanisms based on the size of the vectors. Any scheduling mechanism must find the most optimal execution sequence to avoid accumulations of data, or blocking states due to lack of data (starvation).

The execution vector (repetition vector) and the execution control mechanisms are associated with the processing functions (computations) using parallel allocation components that form a semantic layer around the processing functions. The below allocation models (Figure 5.19, Figure 5.20) provide a description of the semantic layer for the use case. The small part A of Figure 5.19 is illustrated in Figure 5.20; and the small part B of Figure 5.20 is illustrated in Figure 5.19.

The Multidimensional data types and the semantic layers (with their behavior and execution vector) capitalize all the properties related to the semantics for the correct execution of the model in Rhapsody (with the CSP, Array-OL and KPN semantics).

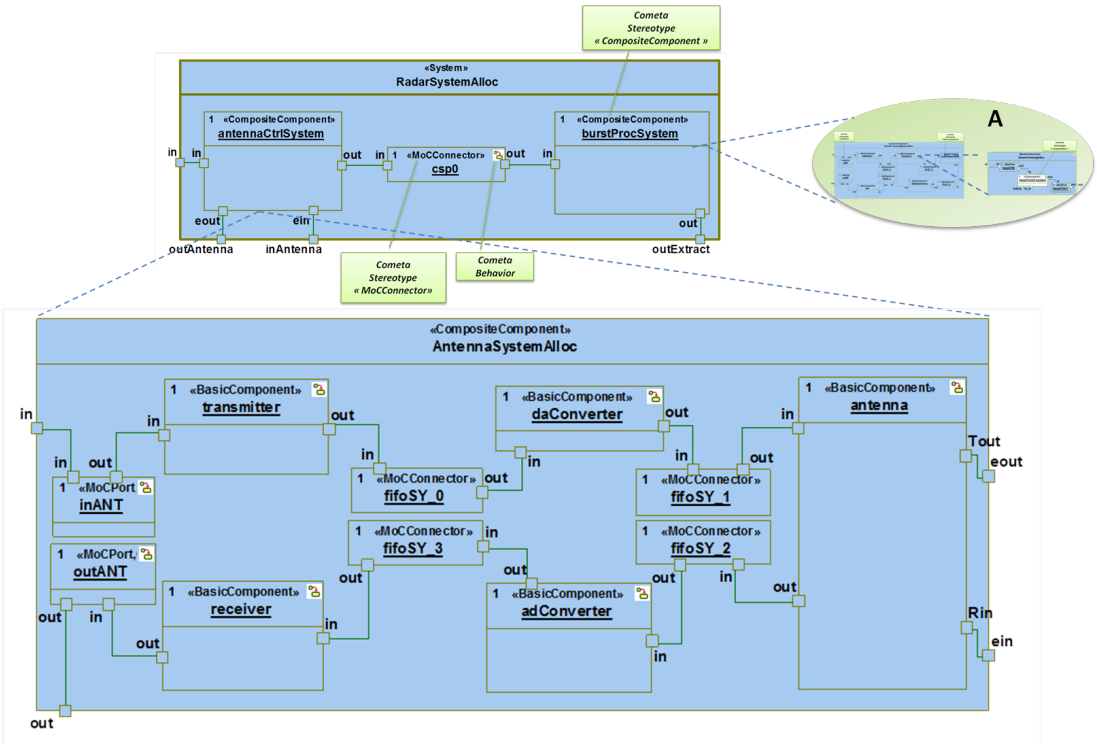

Figure 5.19: Allocated Radar System Model in Rhapsody: The AntennaSystemAlloc

The components in blue represent the semantic layer. The model is decomposed on 2 hierarchical levels. The first hierarchical level presents the connection between the antenna control system (*AntennaSystemAlloc*) and the signal processing system (*BurstProcessingSystemAlloc*). These two sub modules have different computing modes or MoC. The *AntennaSystemAlloc* module contains five sub modules that interact in parallel. The *BurstProcessingSystemAlloc* module interacts with the *AntennaSystemAlloc* by retrieving the echoes produced (which are similar to the *Burst* signals),

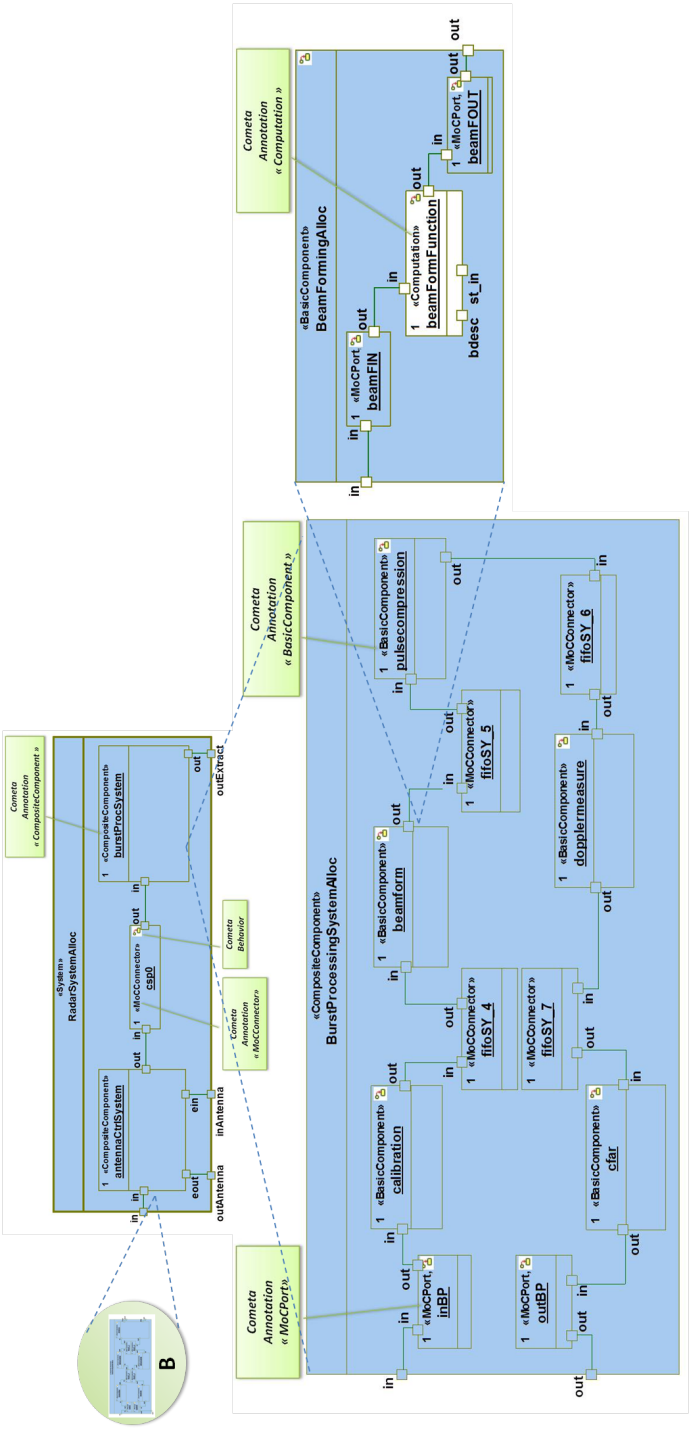

Figure 5.20: Allocated Radar System Model in Rhapsody: The BurstProcessingSystemAlloc

its computations are based on the Array-OL semantics since the data produced by *AntennaSystemAlloc* are multidimensional arrays.

Consequently, the *RadarSystemAlloc* system presented is defined as a *CompositeComponent* containing two sub modules also described as *CompositeComponent*. The composite sub modules are connected with a CSP connector. The choice of the CSP connector is justified by the assumption that the Bursts transmitted by the antenna are processed one by one which has the consequence that each Burst blocks the process as long as it is not fully consumed. The class describing the connector is referenced as a *MoCConnector* and the MoC it references is *SY* in the context of ForSyDe or CSP in Rhapsody. The synchronization behavior associated to the class is provided by a Cometa FSM stored in the libraries of execution control behaviors. The CSP execution control mechanism was demonstrated in Figure 4.17 of section4.3.3.

In the second hierarchical level, the sub modules of the *AntennaSystemAlloc* module are described in the form of classes referenced as *BasicComponent* and the connectors are also *MoCConnector* classes with behavior. The sub components implemented in *AntennaSystemAlloc* are *transmitter*, *daConverter*, *antenna*, *adconverter* and *receiver* which are respectively the instances of *TransmitterAlloc*, *DaConverterAlloc*, *AntennaSystemAlloc* , *AdconverterAlloc* and *ReceiverAlloc*, their composite structures.

The basic components reflect the presence of atomic computations allocated in the components. Each *BasicComponent* describes a composite structure that contains the computation.

BurstProcessingSystemAlloc contains five *BasicComponent* connected with synchronous semantics in the case of ForSyde. The sub components are *calibration*, *beamform*, *pulscompression*, *dopplermeasure* and *cfar* which are respective instances of *CalibrationAlloc*, *BeamformAlloc*, *PulscompressionAlloc*, *DopplermeasureAlloc* and *CfarAlloc*. Each *BasicComponent* also contains repetition vectors in the form of *BasicComponent* class attributes.

We have previously emitted a design hypothesis that in Cometa, the execution control mechanisms for Array-OL are distributed on the structural elements such as the *BasicComponent* and the *MoCPort* as shown in Figure 5.13. Each *BasicComponent* of the *BurstProcessingSystemAlloc* has a matching behavior as defined in Figure 5.14 of Section 5.13. The behaviors associated with the ports are described in the *MoCPort* class internal to the structure of the *BasicComponent*.

The last hierarchical level presents the refinement of the *BasicComponent*. Taking the example of the *BeamformAlloc* (Figure 5.20), its refinement gives a structure composed of an allocated computation *beamFormFunction* and Cometa *MoCPort* for the synchronization according to the Array-OL semantics.

The generic execution control mechanism for Array-OL semantics has been associated to the input and output *MoCPort*. The mechanisms are also described in Figure 5.14.

Regarding the computation module, it is described in the form of an FSM and an application code block. Figure 5.21 shows an excerpt of the FSM corresponding to the computation of *BeamformAlloc*. In the operational mode, on receipt of an input Burst, the module enters a State of processing the data. An excerpt of the operated code block is provided in the right part of Figure 5.21.

The multidimensional array received as input is parsed and processed depending on the available steering settings. Once the processing is done, a new array *beamOUT* is produced as output and sent to *PulseCompression* via the *MoCPort* named *beamOUT*.

After a complete description of the different computations, behaviors and seman-

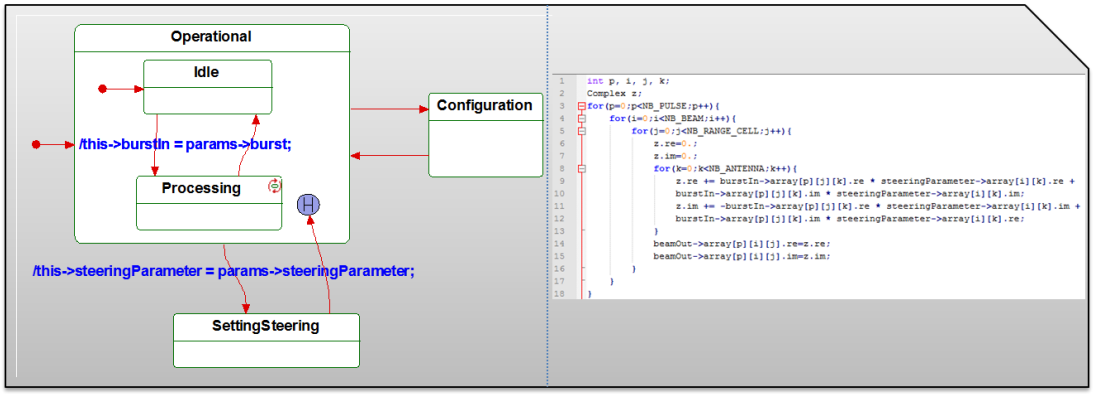

Figure 5.21: Excerpt of the BeamForm computation

tic layers, simulation is performed in Rhapsody. The simulation results presented in Figure 5.22 highlight execution traces of the Array-OL semantics for the BurstProcessing module in an environment that does not natively integrate such semantics. These results are promising because the execution orders are consistent with the possible execution sequences imposed by the Array-OL semantics taking into account produced and consumed data arrays.

Focusing on the interactions between *BeamformAlloc* and *PulscompressionAlloc*, the block of data produced by *BeamformAlloc* is sized (84,32) which equals (2,4) times the blocks consumed by *PulscompressionAlloc* sized (42,8). As a result, the *BasicComponent* around these two processing modules define repetition vectors that make *PulscompressionAlloc* 2x4 times faster than *BeamformAlloc* when execution is launched. The observed traces show that to each production of *BeamformAlloc*, *PulscompressionAlloc* executes 8 times to entirely process the data array produced. At the whole system model scale, execution orders related to the BurstProcessing module obey the semantic rules introduced in the models.

5.3.5.1 Transformation results from Rhapsody to ForSyDe and Spear

In the following transformation steps toward ForSyDe and Spear, the processed models preserve the semantic properties. The allocation model is our starting point for targeting both tools.

The rules to generate ForSyDe models from the Rhapsody model exploit the MoC semantic layers including the description of the signals, structure and elementary tasks. Figure 5.23 and Listing 5.7 show respectively the XMI and XML models corresponding to the highest hierarchical level of the system and resulting from the use of the t1 and t3. In both models, the system is defined as a *ProcessNetwork* containing two *CompositeProcess* connected through a signal *csp0*. All the details are extracted from the Cometa semantic layer in Rhapsody.

The *csp0* signal between the two modules is provided with its associated semantics, the type of data supported by the signal and the linked communication ports. At this hierarchical level, there is no process constructor associated with the MoC since the MoC process constructors are defined at the leafprocess level, and each composite component is described in the form of a ProcessNetwork with its own XML description file. Taking the example of the BurstProcessing module, the following listing 5.8 shows an excerpt of the generated XML file.

In this model, the communication signals are described, related to their source and

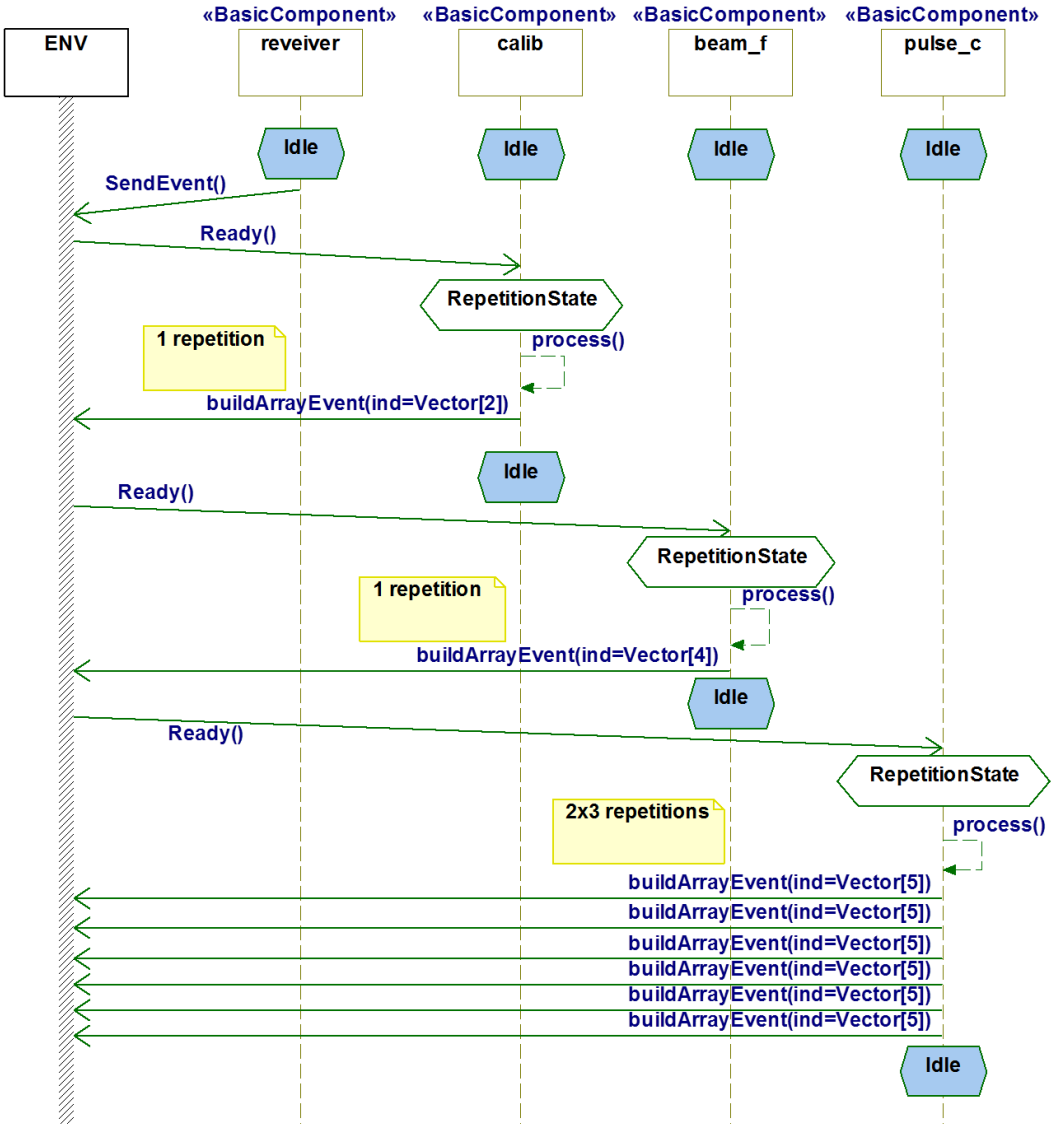

Figure 5.22: Excerpt of the Simulation Results

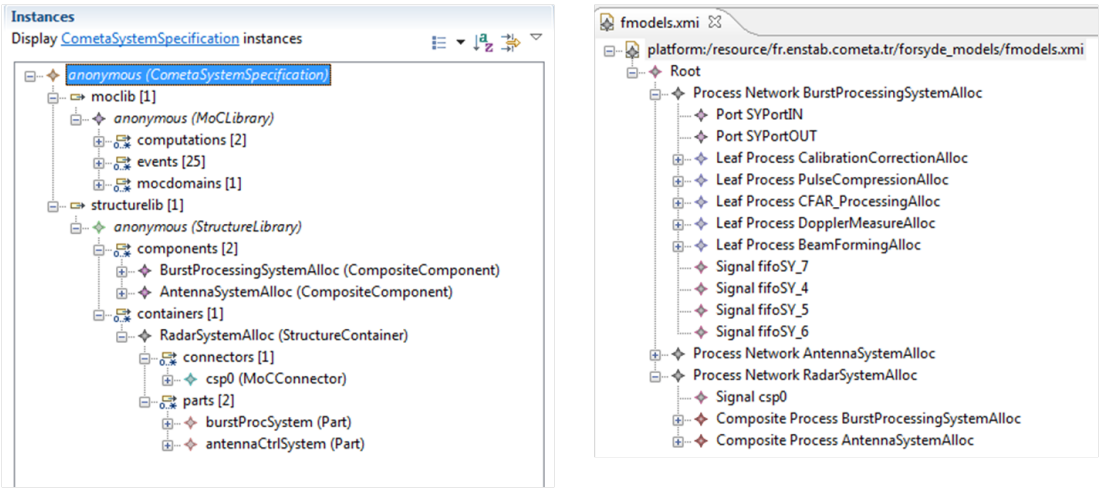

Figure 5.23: Intermediate representations of the Cometa (left) and ForSyDe (right) Radar model

```
<?xml version="1.0" ?>
<?xml-stylesheet type="text/xsl" href="./XSLT/forsyde-systemc.xsl"?>
<!-- Automatically generated from MDWorkBench + Cometa -->
<!DOCTYPE process_network SYSTEM "./DTD/forsyde.dtd" >
<process_network name="RadarSystemAlloc">
        <signal name="csp0" moc="sy" type="int" />
        <composite_process name="BurstProcessingSystemAlloc" >
                <port name="SYPortIN" type="int" direction="in" />
                <port name="SYPortOUT" type="int" direction="out" />
        </composite_process>
        <composite_process name="AntennaSystemAlloc" >
                <port name="SYPortIN" type="int" direction="in" />
                <port name="SYPortOUT" type="int" direction="out" />
        </composite_process>
</process_network>
```

Listing 5.7: Excerpt of the ForSyDe XML model of the Radar example

```
<?xml version="1.0" ?>
<?xml-stylesheet type="text/xsl" href="./XSLT/forsyde-systemc.xsl"?>
<!-- Automatically generated from MDWorkBench + Cometa -->
<!DOCTYPE process_network SYSTEM "./DTD/forsyde.dtd" >
<process_network name="BurstProcessingSystemAlloc">
        <port name="SYPortIN"  type="int" direction="in"  />
        <port name="SYPortOUT"  type="int" direction="out"  />
        <signal name="fifoSY_7" moc="sy"  type="int" source="DopplerMeasureAlloc" source_port
            ="dopplerOUT" target="CFAR_ProcessingAlloc" target_port="cfarIN"/>
        <signal name="fifoSY_4" moc="sy"  type="int" source="CalibrationCorrectionAlloc"
             source_port="calibOUT" target="BeamFormingAlloc" target_port="beamFIN"/>
        <signal name="fifoSY_5" moc="sy"  type="int" source="BeamFormingAlloc" source_port="
            beamFOUT" target="PulseCompressionAlloc" target_port="pulseCIN"/>
        <signal name="fifoSY_6" moc="sy"  type="int" source="PulseCompressionAlloc"
             source_port="pulseCOUT" target="DopplerMeasureAlloc" target_port="dopplerIN"/>
        <leaf_process name="CalibrationCorrectionAlloc">
                <port name="calibOUT"  type="int" direction="out" bound_process="
                    BeamFormingAlloc" bound_port="beamFIN"/>
                <port name="calibIN"  type="int" direction="in"  />
                <process_constructor  >
                </process_constructor>
        </leaf_process>
        <leaf_process name="PulseCompressionAlloc">
                <port name="pulseCOUT"  type="int" direction="out" bound_process="
                    DopplerMeasureAlloc" bound_port="dopplerIN"/>
                <port name="pulseCIN"  type="int" direction="in"  />
                <process_constructor  >
                </process_constructor>
        </leaf_process>
        <leaf_process name="CFAR_ProcessingAlloc">
                <port name="cfarIN"  type="int" direction="in"  />
                <port name="cfarOUT"  type="int" direction="out"  />
                <process_constructor  >
                </process_constructor>
        </leaf_process>
        <leaf_process name="DopplerMeasureAlloc">
                <port name="dopplerOUT"  type="int" direction="out" bound_process="
                    CFAR_ProcessingAlloc" bound_port="cfarIN"/>
                <port name="dopplerIN"  type="int" direction="in"  />
                <process_constructor  >
                </process_constructor>
        </leaf_process>
        <leaf_process name="BeamFormingAlloc">
                <port name="beamFOUT"  type="int" direction="out" bound_process="
                    PulseCompressionAlloc" bound_port="pulseCIN"/>
                <port name="beamFIN"  type="int" direction="in"  />
                <process_constructor  >
                </process_constructor>
        </leaf_process>
</process_network>
```

Listing 5.8: Excerpt of the ForSyDe XML model of BurstProcessing Sub-module

target data types and finally associated with a type of MoC. Since we have *BasicComponent* modules, we define the leafprocess and ProcessConstructor associated with them. We can see that among other generated properties, the different components are associated with the synchronous *combSY* ProcessConstructor which was specified in the UML semantic layer model for *BurstProcessingSystemAlloc* model. Similarly, for each port, the data type property is taken into account.

The XML model is an intermediate representation that allows the generation of SystemC simulation modules. These entities are described in Listing 5.9 which is the final SystemC executable model associated with the structure of the system. This representation uses the MoC process constructor (ProcessConstructor) for simulation defined in the form of libraries and available in the ForSydeSystemC environment.

```
SC_CTOR(System)
{
    // leaf processes
    auto CalibrationCorrectionAlloc = ForSyDe::SY::make_comb("CalibrationCorrectionAlloc
        ",CalibrationCorrection_func,fifoSY_4,csp0);
    CalibrationCorrectionAlloc->calibOUT(fifoSY_4);
    auto  BeamFormingAlloc = ForSyDe::SY::make_comb("BeamFormingAlloc",BeamForming_func,
        fifoSY_5,fifoSY_4);
        BeamFormingAlloc->beamFOUT(fifoSY_5);
    auto PulseCompressionAlloc = ForSyDe::SY::make_comb("PulseCompressionAlloc",
        PulseCompression_func,fifoSY_6,fifoSY_5);
        PulseCompressionAlloc->pulseCOUT(fifoSY_6);
    auto DopplerMeasureAlloc = ForSyDe::SY::make_comb("DopplerMeasureAlloc",
        DopplerMeasure_func,fifoSY_7,fifoSY_6);
        DopplerMeasureAlloc->dopplerOUT(fifoSY_7);
        auto CFAR_ProcessingAlloc = ForSyDe::SY::make_comb("CFAR_ProcessingAlloc",
            CFAR_Processing_func,fifoSY_7);
        CFAR_ProcessingAlloc->cfarIN(fifoSY_7);
        . . .
}
```

Listing 5.9: Excerpt of the generated SystemC code from the UML Radar model

In this description, the difficulty comes from how ForSyDe should represent the structured multidimensional data types, since the execution control relies on these data types. The Array-OL semantics can be scheduled by SDF like execution semantics where production and consumption rates of the components are handled. ForSyDe allows the SDF execution semantics to be supported which gives us the possibility to encapsulate the Array-OL data types in the SDF MoC data of ForSyDe. The last stage involves using the right ProcessConstructor in order to enhance the model execution.

This alternative solution can be replaced by a more efficient solution where the SDF ProcessConstructor of ForSyDe is refined to support specifically a Scheduling for Array-OL. For instance, the execution control model of Cometa allowing such functionality can be used to automatically generate the corresponding ProcessConstructor. Unfortunately such a solution was not implemented during this thesis.

5.3.5.2 Generation of NoC architecture for ForSyDe BackEnd

The description of the NoC architecture for the system is accompanied by a description of the HW model and the mapping model. Excerpts of the concepts of these two descriptions are provided in A.1.4 and A.1.3).

The HW model specifies 4 parallel nodes forming a 2x2 ring (Node0, Node1, Node2, Node3). The properties of the nodes are described by different parameters illustrated in Listing 5.10.

The mapping model combines the Node0 with *CalibrationCorrectionAlloc PulseCompressionAlloc CFAR_ProcessingAlloc*, Node1 with *DopplerMeasureAlloc BeamFormingAlloc* components, Node2 with *AntennaSystemAlloc DigitalAnalogConverterAlloc*

AnalogDigitalConverterAlloc *TransmitterAlloc* and *ReceiverAlloc* are mapped to Node3. The processes were already described in the *ProcLinck* model with the interactions (e.g. sources, targets).

After the t4 transformation, the NoCGenerator model defines the allocation of the processes on their associated parallel nodes. The Listing 5.10 shows two distinct parts dedicated to the description of the HW and SW allocation. In the lower part, the SW allocation shows the mapping of the various processes into the nodes, their sources and targets, and the processing files corresponding to each process with the extension *.c*. The type of MoC is also specified knowing that its implementation is provided in the kernel of ForSyDe. This information is derived from the definition of MoC introduced in the ProcLink model for ForSydeSystemC. The NoC architecture model generated is used for the synthesis of code to FPGA platforms, which we did not address.

```
<?xml version="1.0" encoding="UTF-8"?>
<system name="Ring2x2">
        <parameter name="targetDirectory" value="C:/Ring2x2_singleproc_v52"/>
        <parameter name="targetManufacturer" value="Altera"/>
        <parameter name="targetManufacturerVersion" value="10.1"/>
        <parameter name="boardType" value="DE3"/>
        <parameter name="boardFrequency" value="50 MHz"/>
        <parameter name="Clock" value="{sys_clk,T33}"/>
        <parameter name="Reset" value="{reset,U31}"/>
        <hardware>
                <parameter name="nocType" value="Mesh"/>
                <parameter name="nocKind" value="2DNoC"/>
                <parameter name="rni_version" value="v2.0"/>
                <parameter name="HDLrootDirectory" value="C:/NoC_v52"/>
                <parameter name="nrofCols" value="2"/>
                <parameter name="nrofRows" value="2"/>
                <parameter name="nrofLayers" value="1"/>
                <parameter name="framesize" value="64"/>
                <parameter name="GlobalSync" value="1 Hz"/>
                <parameter name="LayoutMethod" value="floating"/>
                <node nr="0" mem_size="4096" jtag="yes" perf_counter="no" pio="{o,16}"
                     noc_irq="no" cpu="{nios,tiny}" />
                <node nr="1" mem_size="4096" jtag="yes" perf_counter="no" pio="{o,8}" noc_irq
                    ="no" cpu="{nios,tiny}" />
                <node nr="2" mem_size="4096" jtag="yes" perf_counter="no" pio="{o,8}" noc_irq
                    ="no" cpu="{nios,tiny}" />
                <node nr="3" mem_size="4096" jtag="yes" perf_counter="no" pio="{o,8}" noc_irq
                    ="no" cpu="{nios,tiny}" />
        </hardware>
        <software>
                <parameter name="Repository" value="C:/NoC_v52/Examples/ENSTA"/>
                <process name="CalibrationCorrectionAlloc" node="0" sources="{}" targets=
                    "{BeamFormingAlloc}" files="{CalibrationCorrectionAlloc.c}"/>
                <process name="PulseCompressionAlloc" node="0" sources="{BeamFormingAlloc}"
                        targets="{DopplerMeasureAlloc}" files="{PulseCompressionAlloc.c}"/>
                <process name="CFAR_ProcessingAlloc" node="0" sources="{DopplerMeasureAlloc
                    }" targets="{}" files="{CFAR_ProcessingAlloc.c}"/>
                <process name="DopplerMeasureAlloc" node="1" sources="{
                    PulseCompressionAlloc}" targets="{CFAR_ProcessingAlloc}" files="{
                    DopplerMeasureAlloc.c}"/>
                <process name="BeamFormingAlloc" node="1" sources="{
                    CalibrationCorrectionAlloc}" targets="{PulseCompressionAlloc}" files="
                    {BeamFormingAlloc.c}"/>
                <process name="AntennaAlloc" node="2" sources="{DigitalAnalogConverterAlloc
                    }" targets="{AnalogDigitalConverterAlloc}" files="{AntennaAlloc.c}"/>
                <process name="DigitalAnalogConverterAlloc" node="2" sources="{
                    TransmitterAlloc}" targets="{AntennaAlloc}" files="{
                    DigitalAnalogConverterAlloc.c}"/>
                <process name="AnalogDigitalConverterAlloc" node="2" sources="{AntennaAlloc
                    }" targets="{ReceiverAlloc}" files="{AnalogDigitalConverterAlloc.c}"/>
                <process name="TransmitterAlloc" node="2" sources="{}" targets="{
                    DigitalAnalogConverterAlloc}" files="{TransmitterAlloc.c}"/>
        </software>
</system>
```

Listing 5.10: NoCGenerator model Sample

5.3.5.3 Generation of Spear model of the BurstProcessing module

Spear is based on the Array-OL semantics; therefore the multidimensional data arrays, the computations and repetition vectors are natively part of the Spear formalisms.

Table 5.3: Use Case Activities Coverage

Activities/Tools	Rhapsody Modeler	Rhapsody/ Cometa	Spear DE
UML Specification	Yes	Yes	Yes
(DE) Simulation	Yes	Yes	No
Array-OL Simulation	No	Yes	Yes
Design Space Exploration	No	No	Yes

The semantics models and profile that have been added in Rhapsody by Cometa aim at providing as many properties as possible related to the Array-OL semantics: first for a coherent analysis of the models in Rhapsody, then to keep and transform the static properties through the transformation rules from Rhapsody to Spear. The Cometa semantic layers and the Spear transformation rules (t6) have allowed the generation of an intermediate Spear XMI model incorporating these properties (Figure **??**-left). The model is further converted to its equivalent representation in Moml (serialization format for the Spear tooling).

An extract of the MOML model of the BurstProcessing module is presented in the right part of Figure **??**. The excerpts from the models show automatic integration of information related to data (Array), the repetition vectors (Loop) and the Computations. The execution control is assumed by the *Scheduler* implemented natively in Spear.

The Spear simulation results also follow the Array-OL possible execution sequences. In the case of Rhapsody the semantics was not natively defined and was incorporated thanks to the use of the Cometa libraries. The properties used for simulation were then transformed into the Spear environment.

5.4 Conclusion

In this chapter, we have tried to describe the impact of the use of Cometa on a tool chain towards several activities (mainly specification, simulation and code synthesis). The use case allows the needs to be explained in terms of expression of semantics which often disappear during model exchanges between tools causing ambiguous interpretation of the models.

Through this experiment, one can clearly see that semantics' management through a design tool chain is tedious and requires several intermediate steps to express the semantics. Omitting these semantics may end up in inconsistent model exchanges. Hopefully, our experiment shows that it is possible to integrate heterogeneous models' semantics in tools such as Rhapsody in order to foster the coherent analysis of semantically rich models.

In our example, Table 5.3 shows a subset of the activities (first column) that highlight the positioning and contribution of Cometa for interoperability and preservation of semantics in different design processes, especially between the MDE tool oriented SLM and the formal design tools. For example, in the second column, Rhapsody offers support for the UML specification and simulation of models based on discrete events.

However, the MoC (e.g. Array-OL) based semantic properties are not natively present in the UML Modeler. Therefore, they cannot be simulated correctly, which reduces the consistency and reliability of the models during the exchanges between tools. Cometa contributes to this specific concern, using the relationship between

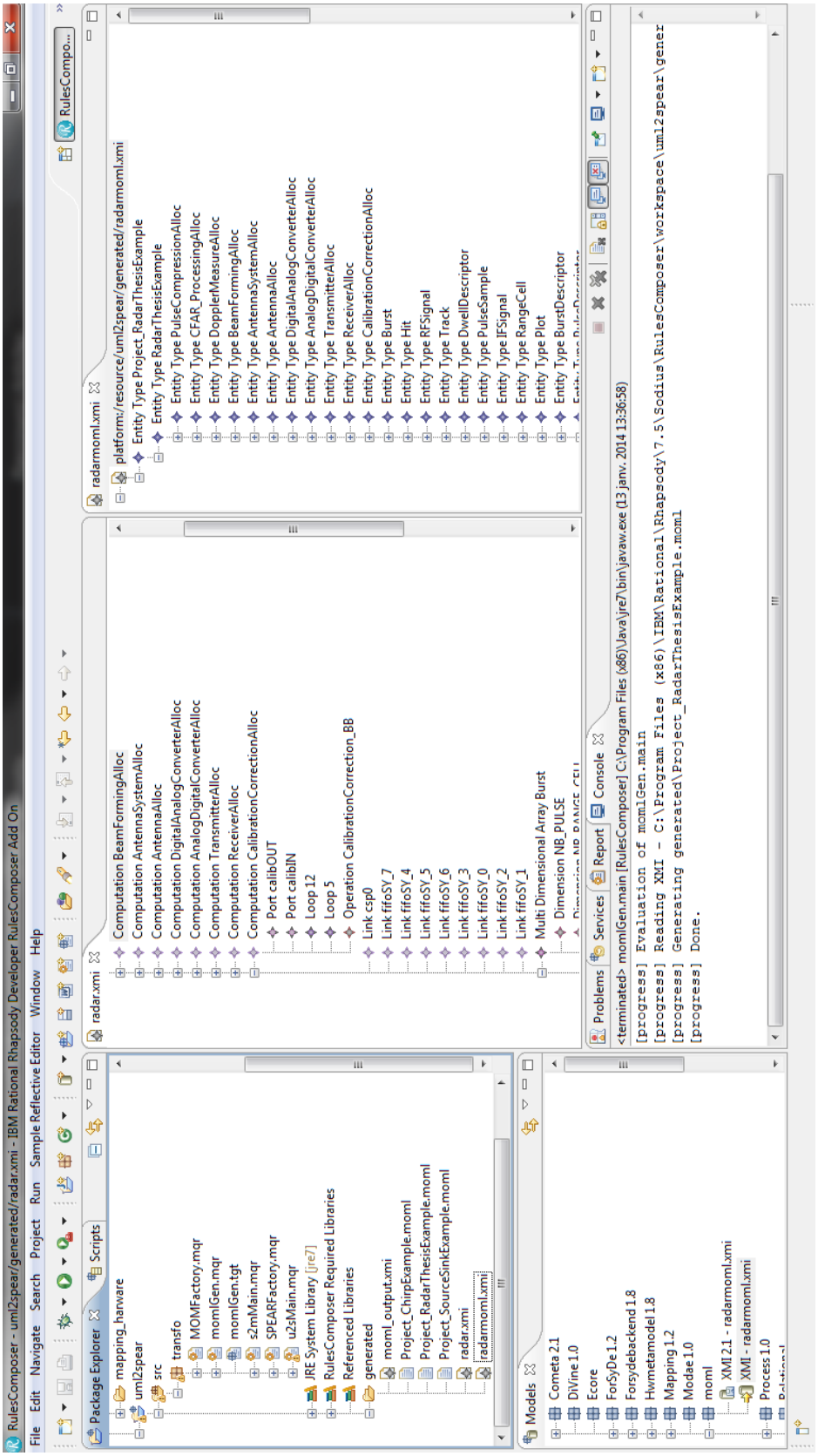

Figure 5.24: Generated Intermediate representations of the Spear XMI (left) and Spear MOML (right) Radar models

semantic domains to establish compliance detection and to provide semantics' capture when compliance is proven.

6 Conclusions and Perspectives

Contents

In this part we present the conclusions and perspectives of our research. We will synthesize our critical opinion on the obtained results in the next two sections.

6.1 Conclusions

The definition of tool chains for complex and heterogeneous embedded systems is a major challenge which requires the taking into account of several issues which are the creeds of several research communities.

The heterogeneity of the systems is measured by the number of design paradigms and the number of engineering domains combining different execution models, models of communication and scheduling mechanisms (synchronous, asynchronous, directed data flow, timed, etc).

In the design steps, these constraints can be assimilated to semantic constraints and must be taken into account throughout the development process and especially during the exchange of models between tools for design, analysis and refinement. In this thesis, we try to bring some responses aiming precisely to correct the loss of semantics during the above mentioned phases.

The thesis allowed contributions to be implemented in the context of the description of tool chains, analysis and semantics' preservation of heterogeneous models. The results presented are evidence of accomplished objectives at least on the description and preservation of the semantics of exchanged models in a design flow.

The semantics which were available only in dedicated tools could now be expressed and even analyzed in environments natively implementing a different semantics. This paves the way for the integration of new design tools in tool chains and to use semantic adjustments that can be provided in the form of MoC semantics' libraries.

From this point of view, the results are satisfying, even though there are many points open to discussion and improvement, which will be certainly addressed in coming development or in the context of other projects such as GEMOC.

For the preservation of the semantics of models, we could show through the experimentation results in Section 5.3 that it is possible under certain conditions to capture the missing semantics and re-inject them into models to provide a consistent analysis of executable models and their refinement in different environments.

The methodological steps for identifying the MoCs and their compatibility partly meet *Aim 1*. The specification of the missing semantics and the demonstration of the semantics' preservation complete the rest of *Aim 1* allowing us to conclude that the proposed technique is a possible and formal solution.

Through the experimentation of the Radar application, the properties related to the semantics of this type of model were captured and reflected in several tools that implement different semantics thus addressing the second target *Aim 2*.

In General, we have shown that the semantic questions, usually hardly addressed by integration tool environments or left at the discretion of designers, can be treated generically by a methodological approach focused on the preservation of the semantics.

With the definition of the MoC operational semantics using Event-based FSMs, we offer executability to the MoC models thus promoting their use for execution control during the analysis by simulation phases. Furthermore, the experiments helped show that it was possible to represent semantics such as CSP, KPN, SDF, Array-OL with the Cometa DSML.

This ability to express the properties related to the semantics of models in an executable way also opens up the possibility of using strong value added tools such as

the model-checking tools that manipulate formalisms such as FSMs, PetriNets, etc.

Similarly, when the homogeneous execution semantics of an analysis tool is known, our solution allows MoC models to be imported that offer the ability to express heterogeneous semantics that natively was not implemented in the tool. Several experiments have been conducted on use cases with satisfactory results in terms of semantics preservation.

The DSML formalization gives a new dimension to the Cometa approach, since it makes it possible to formally demonstrate the conformity of the MoC model's operational description to the execution properties that constitute their foundation.

The approach has nevertheless some points to improve that we will present in the perspective. The shortcomings of the approach include: the lack of proper execution engine for the Cometa Modeler. For the moment, this failure is not annoying since the implemented models are exclusively reused in a context of tool chains where the source and target tools are supposed to have exploitable execution engines. The other difficulty for the use of the approach is related to the MoC theory i.e. the necessary efforts of MoC learning to be able to identify, represent and compare them. To preserve the semantics, this step is mandatory. Finally, our approach does not express MoC semantics based on differential equations at the time.

6.2 Perspectives

The perspectives are divided into three points that we detail in the remainder of this section.

6.2.1 Definition of an Execution Engine

The definition of an execution engine would make the Cometa Modeler usable for analysis and simulation regardless of any simulation or analysis tool. Since different heterogeneous semantics are expressed using the same uniform description format based on Event-Based FSMs, the execution engine must allow the interpretation of the semantics of this formalism. The operational rules defined in section 4.2.4 can help for this purpose.

One of the objectives in the ANR GEMOC project is to define an execution engine for a MoC DSML combining CCSL and Cometa. Indeed, the project has a package dedicated to the description of a MoC description language reusable for the execution control of various models (that conform to various DSL and DSML). The language in question will consist in a merger of the CCSL and Cometa DSML providing a unique and formal language to describe executable and heterogeneous MoCs (e.g. synchronous, asynchronous, or timed). The above Diagram 6.1 shows the conceptual vision of the GEMOC approach with the positioning of Cometa.

In order to provide an execution engine for the Cometa Modeler, a Model Explorer (NEMO) is currently being developed. The NEMO engine aims at calculating all possible execution paths for a component based system that integrate FSMs. NEMO Explorer permits the construction of the execution graph corresponding to all the states and transitions resulting from the combination of the Cometa MoC model and the system application models. Each path of the graph represents a possible execution sequence. The implemented solution will probably be part of the contributions in the GEMOC project.

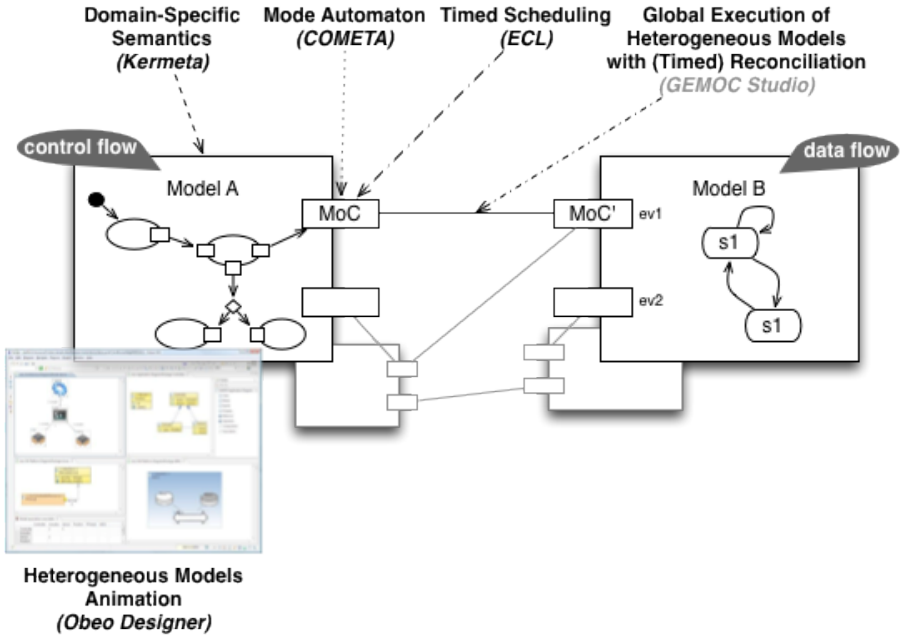

Figure 6.1: Positioning Cometa in the GEMOC Approach

Besides, the use of NEMO opens the door to the use of tools to perform model-checking activities where execution properties will be observed on the enriched system model. The observation of property patterns can lead to the identification of implicit MoC implementations.

6.2.2 From Denotational semantics to formal MoC model description

When we did the state of the art on the MoC theory, we could observe that all the manipulated MoCs have a mathematical formalization (denotational representation). In the design and analysis tools, such formal semantics are expressed in the form of operational implementations. In our case, we have captured them in the form of Event-based FSMs but manually. The idea of this perspective is to ensure the automatic translation from the denotational MoC representation to operational representations using the Cometa FSMs. Even if several works have addressed the passage of denotational semantics to operational semantics, their relation to MoC theory is not yet established.

Since the denotational descriptions are more abstract than the operational, the MoC models could then be expressed directly in the form of mathematical models more easily handled and related to MoC specifications. When it is necessary that these models are expressed for analysis or dynamic specification then automatic transformation will allow the translation from one formalism to another.

Accomplishing such translation would require the consideration of a fixed set of symbols to express a denotational semantics on mathematical basis. The mathematical language should be flexible enough to express most of the formal MoCs.

Afterwards, the difficult part involves the specification of translation rules from mathematical representation to FSM models knowing that all the semantics might not be expressed in the form of FSMs.

This allows us to switch on the last perspective which addresses the expression of untreated MoC families.

6.2.3 Differential Equation Description

The expression of semantics based on differential equations is one of the limitations of Cometa. That would mean, in a sense, the extension of the Cometa DSML to add the description of concepts to express ODE or ADE formalism. Consequently, we should also give their associated operational semantics, their composition semantics with the other parts of the DSML using FSMs. This task is very difficult and we did not spend time on this aspect. A second solution would be to consider the modules described with ODE semantics as black-boxes where the I/O exchanges are captured and adapted according to the external semantics around the modules.

Bibliography

[1] Thomas Abdoul, Joel Champeau, Philippe Dhaussy, P-Y Pillain, and J Roger. Aadl execution semantics transformation for formal verification. In *Engineering of Complex Computer Systems, 2008. ICECCS 2008. 13th IEEE International Conference on*, pages 263–268. IEEE, 2008.

[2] Luca Aceto, Wan Fokkink, and Chris Verhoef. Structural operational semantics. In *Handbook of Process Algebra*, pages 197–292. Elsevier, 1999.

[3] Jörg Ackermann and Klaus Turowski. A library of ocl specification patterns for behavioral specification of software components. In Eric Dubois and Klaus Pohl, editors, *Advanced Information Systems Engineering*, volume 4001 of *Lecture Notes in Computer Science*, pages 255–269. Springer Berlin Heidelberg, 2006.

[4] Altera. Avalon interface specifications. `http://www.altera.com/literature/manual/mnl_avalon_spec.pdf`.

[5] Charles André. Syntax and Semantics of the Clock Constraint Specification Language (CCSL). Rapport de recherche RR-6925, INRIA, 2009.

[6] Charles André, Frédéric Mallet, and Robert De Simone. Time Modeling in MARTE. In *ECSI Forum on specification & Design Languages (FDL)*, pages 268–273, Barcelona, Espagne, 2007. ECSI, ECSI. The original publication is available at http://www.ecsi-association.org/ecsi/main.asp?l1=library&fn=def&id=268.

[7] W R Ashby. *An introduction to cybernetics*, volume 16 of *University Paperbacks*. Chapman & Hall, 1956.

[8] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: an integrated electronic system design environment. *Computer*, 36(4), 2003.

[9] Henk Barendregt and Erik Barendsen. Introduction to Lambda Calculus. *Nieuw archief voor wisenkunde*, 4(March):337–372, 2000.

[10] A Benveniste and G Berry. The synchronous approach to reactive and real-time systems, 1991.

[11] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the signal language and its semantics. *Sci. Comput. Program.*, 16(2):103–149, 1991.

[12] Olivier Berger, Sabri Labbene, Madhumita Dhar, and Christian Bac. Introducing OSLC, an open standard for interoperability of open source development tools. In *ICSSEA*, pages ISSN–0295–6322, Paris, France, 2011. FUI.

[13] G. Berry, P. Couronne, and G. Gonthier. Synchronous programming of reactive systems: an introduction to esterel. In *Proceedings of the first Franco-Japanese Symposium on Programming of future generation computers*, pages 35–56, Amsterdam, The Netherlands, The Netherlands, 1988. Elsevier Science Publishers B. V.

[14] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, November 1992.

[15] Ludwig Von Bertalanffy. *General System Theory: Foundations, Development, Applications*, volume 21. George Braziller, 1969.

[16] Jean Bézivin. Model engineering for software modernization. In *WCRE*, page 4, 2004.

[17] Jean Bézivin and Olivier Gerbé. Towards a precise definition of the omg/mda framework. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, ASE '01, pages 273–, Washington, DC, USA, 2001. IEEE Computer Society.

[18] T Bollaert. Catapult synthesis: a practical introduction to interactive C synthesis. In *High-Level Synthesis from Algorithm to Digital Circuit*, pages 29–52. Springer Verlag, 2008.

[19] Frédéric Boulanger and Cécile Hardebolle. Simulation of Multi-Formalism Models with Modhel'X. In *ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 318–327, Washington, DC, USA, 2008. IEEE Computer Society.

[20] Pierre Boulet. Array-OL Revisited, Multidimensional Intensive Signal Processing Specification. Rapport de recherche RR-6113, INRIA, 2007.

[21] Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors. *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*, volume 247 of *Lecture Notes in Computer Science*. Springer, 1987.

[22] Tim Bray, J Paoli, and CM Sperberg-McQueen. Extensible Markup Language (XML), 2000.

[23] L.B. Brisolara, M.F.S. Oliveira, R. Redin, L.C. Lamb, and F. Wagner. Using UML as Front-end for Heterogeneous Software Code Generation Strategies. *2008 Design, Automation and Test in Europe*, 2008.

[24] Allan R Broadhurst and Donald K Darnell. An Introduction to Cybernetics and Information Theory. *Quarterly Journal of Speech*, 51(4):442–453, 1965.

[25] Manfred Broy. Towards a formal foundation of the specification and description language SDL, 1991.

[26] Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Ptolemy: a framework for simulating and prototyping heterogeneous systems. *IEEE*, 10:527–543, 2002.

[27] Frank Budinsky. *Eclipse modeling framework: a developer's guide.* Addison-Wesley Professional, 2004.

[28] Ravi Budruk, Don Anderson, and Ed Solari. *PCI Express System Architecture.* Pearson Education, 2003.

[29] David Byrne. *Complexity theory and the social sciences: an introduction*, volume 67. Routledge, 1998.

[30] 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.

[31] Michael J Chen and Edward A Lee. Design and implementation of a multidimensional synchronous dataflow environment. In *Signals, Systems and Computers, 1994. 1994 Conference Record of the Twenty-Eighth Asilomar Conference on*, volume 1, pages 519–524. IEEE, 1994.

[32] Patrick Cousot. Methods and logics for proving programs. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 841–994. 1990.

[33] Arnaud Cuccuru, Chokri Mraidha, Franois Terrier, and Sbastien Grard. Enhancing UML Extensions with Operational Semantics. In *Model Driven Engineering Languages & Systems (MoDELS)*, pages 271–285, 2007.

[34] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, volume 45, pages 1–17, 2003.

[35] Julien DeAntoni and Frédéric Mallet. Timesquare: Treat your models with logical time. In *Objects, Models, Components, Patterns*, pages 34–41. Springer, 2012.

[36] D. Densmore and R. Passerone. A Platform-Based Taxonomy for ESL Design. *IEEE Design & Test of Computers*, 23(5), 2006.

[37] Davide Di Ruscio, Frédéric Jouault, Ivan Kurtev, Jean Bézivin, and Alfonso Pierantonio. Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs. RR 06.02 RR 06.02.

[38] Papa Issa Diallo, Jo"el Champeau, and Lo"ic Lagadec. A Model-Driven Approach to Enhance Tool Interoperability using the Theory of Models of Computation. In Richard F. Paige Martin Erwig and Eric van Wyk, editors, *6th International Conference on Software Language Engineering (SLE 2013)*, Lecture Notes in Computer Science, Indianapolis, 'Etats-Unis, 2013. Springer-Verlag.

[39] Rainer Dömer, Andreas Gerstlauer, Junyu Peng, Dongwan Shin, Lukai Cai, Haobo Yu, Samar Abdi, and Daniel D. Gajski. System-on-chip environment: A specc-based framework for heterogeneous mpsoc design. *EURASIP J. Embedded Syst.*, 2008:5:1–5:13, January 2008.

[40] Cormac Driver, Sean Reilly, Éamonn Linehan, Vinny Cahill, and Siobhán Clarke. Managing embedded systems complexity with aspect-oriented model-driven engineering, 2010.

[41] Inc Eclipse Foundation. About the Eclipse Foundation, 2010.

[42] Stephen Anthony Edwards, Stephen Anthony Edwards, and Stephen Anthony Edwards. The specification and execution of heterogeneous synchronous reactive systems. Technical report, University of California, Berkeley, 1997.

[43] Johan Eker, Jorn Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003.

[44] Johan Eker, Jorn Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Sonia Sachs, Yuhong Xiong, and Stephen Neuendorffer. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[45] Cagkan Erbas, Andy D. Pimentel, Mark Thompson, and Simon Polstra. A framework for system-level modeling and simulation of embedded systems architectures. *EURASIP J. Emb. Sys.*, 2007, 2007.

[46] Rik Eshuis and Roel Wieringa. A real-time execution semantics for uml activity diagrams. In Heinrich Hussmann, editor, *Fundamental Approaches to Software Engineering*, volume 2029 of *Lecture Notes in Computer Science*, pages 76–90. Springer Berlin Heidelberg, 2001.

[47] Jean-Marie Favre. Foundations of meta-pyramids: Languages vs. metamodels-episode ii: Story of thotus the baboon1. *Language Engineering for Model-Driven Software Development*, 4101, 2004.

[48] Jean-Marie Favre. Foundations of Model (driven) (Reverse) Engineering - Episode I: Story of the Fidus Papyrus and the Solarus. In *postproceedings of Dagsthul Seminar on Model Driven Reverse Engineering*, 2004.

[49] Jean-Marie Favre. Towards a Basic Theory to Model Model Driven Engineering. In *3rd Workshop in Software Model Engineering WiSME*, volume 1, pages 262–271. Citeseer, 2004.

[50] Peter H. Feiler, David P. Gluch, and John J. Hudak. The architecture analysis & design language (AADL): An introduction. Technical Report CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University, 2006.

[51] Alberto Ferrari. An Overview of (Electronic) System Level Design: beyond hardware-software co-design. `http://www.sti.uniurb.it/events/sfm06hv/slides/Ferrari.pdf`.

[52] Marcelo P. Fiore, Achim Jung, Eugenio Moggi, Peter O'Hearn, Jon Riecke, Giuseppe Rosolini, and Ian Stark. Domains and Denotational Semantics: History,

Accomplishments and Open Problems. *Bulletin of the European Association for Theoretical Computer Science*, 59:227–256, 1996.

[53] R W Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):19–32, 1967.

[54] The International Technology Roadmap for Semiconductors. 2007 edition. Technical report, ITRS, 2007.

[55] Daniel D. Gajski, Nikil D. Dutt, Allen C.-H. Wu, and Steve Y.-L. Lin. *High-level synthesis: introduction to chip and system design.* Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[56] D.D. Gajski, A.C.-H. Wu, V. Chaiyakul, S. Mori, T. Nukiyama, and P. Bricaud. Essential issues for IP reuse. *Proceedings 2000. Design Automation Conference. (IEEE Cat. No.00CH37106)*, 2000.

[57] David Garlan, Robert Monroe, and David Wile. Acme: an architecture description interchange language. In *CASCON First Decade High Impact Papers*, CASCON '10, pages 159–173, Riverton, NJ, USA, 2010. IBM Corp.

[58] Sebastien Gerard, Jean-Philippe Babau, and Joel Champeau. *Model Driven Engineering for Distributed Real-Time Embedded Systems.* Wiley-IEEE Press, 2010.

[59] Sébastien Gérard, Cédric Dumoulin, Patrick Tessier, and Bran Selic. Papyrus: A UML2 Tool for Domain-Specific Language Modeling. In Holger Giese, Gabor Karsai, Edward Lee, Bernhard Rumpe, and Bernhard Schätz, editors, *Model-Based Engineering of Embedded Real-Time Systems*, volume 6100 of *Lecture Notes in Computer Science*, pages 361–368. Springer Berlin / Heidelberg, 2011.

[60] Andreas Gerstlauer, Christian Haubelt, Andy D. Pimentel, Todor Stefanov, Daniel D. Gajski, and Jürgen Teich. Electronic system-level synthesis methodologies. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 28(10):1517–1530, 2009.

[61] F. Ghenassia. *Transaction level modeling with systemc: Tlm concepts and applications for embedded systems.* Springer Verlag, 2005.

[62] Antoon Goderis, Christopher Brooks, Ilkay Altintas, Edward A. Lee, and Carol Goble. Heterogeneous composition of models of computation. Technical Report UCB/EECS-2007-139, EECS Department, University of California, Berkeley, Nov 2007.

[63] Mudit Goel. *Process networks in Ptolemy II.* PhD thesis, UNIVERSITY of CALIFORNIA, 1998.

[64] Martin Gogolla, Fabian Büttner, and Mark Richters. Use: A uml-based specification environment for validating {UML} and {OCL}. *Science of Computer Programming*, 69(13):27 – 34, 2007. Special issue on Experimental Software and Toolkits.

[65] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. Introduction to Parallel Computing; 2nd Edition. *Search*, page 856, 2003.

[66] Matthias Gries and Kurt Keutzer. *Building ASIPs: The Mescal Methodology.* Springer Publishing Company, Incorporated, 1st edition, 2010.

[67] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[68] T. Grötker, S. Liao, G. Martin, and S. Swan. *System design with SystemC.* Kluwer Academic Pub, 2002.

[69] Carl A Gunter, Peter D Mosses, and Dana S Scott. Semantic domains and denotational semantics. Technical report, DTIC Document, 1989.

[70] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. In *Proceedings of the IEEE*, pages 1305–1320, 1991.

[71] D. Harel and A. Pnueli. Logics and models of concurrent systems. chapter On the development of reactive systems, pages 477–498. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

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

[73] M Hause. The SysML Modelling Language. *Fifteenth European Systems Engineering Conference*, 9(September), 2006.

[74] David Hearnden, Michael Lawley, and Kerry Raymond. Model Driven Engineering Languages and Systems. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Proceedings of the 9th international conference on Model Driven Engineering Languages and Systems - MoDELS'06*, volume 4199 of *Lecture Notes in Computer Science*, pages 321–335. Springer Berlin Heidelberg, 2006.

[75] Christian Hein, Tom Ritter, and Michael Wagner. Model-Driven Tool Integration with ModelBus. *Workshop Future Trends of ModelDriven Development*, pages 35–39, 2009.

[76] Thomas A Henzinger and Joseph Sifakis. The Embedded Systems Design Challenge. *Foundations*, 4085:1–15, 2006.

[77] F. Herrera and E. Villar. A framework for embedded system specification under different models of computation in SystemC. *2006 43rd ACM/IEEE Design Automation Conference*, 2006.

[78] F Herrera, E Villar, C Grimm, M Damm, and J Haase. Heterogeneous Specification with HetSC and SystemC-AMS : Widening the Support of MoCs in SystemC. In Eugenio Villar, editor, *Embedded Systems Specification and Design Languages*, chapter 8. Springer Netherlands, 2008.

[79] Fernando Herrera, Pablo Sánchez, and Eugenio Villar. Modeling of csp, kpn and sr systems with systemc. pages 133–148, 2004.

[80] Fernando Herrera and Eugenio Villar. A framework for heterogeneous specification and design of electronic embedded systems in systemc. *ACM Trans. Des. Autom. Electron. Syst.*, 12:22:1–22:31, May 2008.

[81] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[82] C. A. R. Hoare. Communicating sequential processes, 1978.

[83] John E. Hopcroft. *Introduction to Automata Theory, Languages, and Computation*. Pearson Addison Wesley, 3rd edition, 2007.

[84] Gérald Huet. An algorithm to generate the basis of solutions to homogeneous linear diophantine equations. *Information Processing Letters*, 7(3):144–147, 1978.

[85] IBM Telelogic. Rational Rhapsody UML modeler. `http://www.telelogic.com/products/rhapsody/index.cfm`.

[86] iFEST ARTEMIS Joint Undertaking (JU). Industrial Framework for Embedded Systems Tools. `http://www.artemis-ifest.eu/`.

[87] Axel Jantsch. *Modeling Embedded Systems and SoCs - Concurrency and Time in Models of Computation*. Systems on Silicon. Morgan Kaufmann Publishers, June 2003.

[88] Axel Jantsch. Models of embedded computation. In Richard Zurawski, editor, *Embedded Systems Handbook*. CRC Press, 2005. Invited contribution.

[89] Axel Jantsch and Ingo Sander. Models of computation in the design process. *System-on-Chip*, page 161, 2005.

[90] Tor Jeremiassen, Grant Martin, Tim Kogel, Adam Donlin, Andres Takach, and Karam Chatha. From ESL 2010 to ESL 2015. *2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 61–62, 2010.

[91] Jean-Marc Jézéquel, Olivier Barais, and Franck Fleurey. Model driven language engineering with kermeta. In *Proceedings of the 3rd international summer school conference on Generative and transformational techniques in software engineering III*, GTTSE'09, pages 201–221, Berlin, Heidelberg, 2011. Springer-Verlag.

[92] Jean-Marc J'ez'equel, Benoit Combemale, and Didier Vojtisek. *Ing'enierie Dirig'ee par les Mod'eles : des concepts 'a la pratique...* R'ef'erences sciences. Ellipses, February 2012.

[93] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, March 2004.

[94] Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In *Satellite Events at the MoDELS 2005 Conference*, pages 128–138. Springer, 2006.

[95] CESAR ARTEMIS Joint Undertaking (JU). Cost-efficient methods and processes for safety relevant embedded systems. `http://www.cesarproject.eu/`.

[96] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, NY, 1974.

[97] S Karris. Introduction to Simulink® with engineering applications. In *Mathematics*, page 572. Orchard Publications, 2006.

[98] Shirl Kennedy. Resource Description Framework (RDF). *Computers in Libraries*, 24(2):27, 2004.

[99] Birman Kenneth. The Common Object Request Broker Architecture. *Guide to Reliable Distrubuted System, Springer London*, pages 249–269, 2012.

[100] Brian W Kernighan and Dennis M Ritchie. *The C programming language*, volume 78. Prentice Hall, 1988.

[101] K Keutzer, S Malik, R Newton, and J Rabaey. System Level Design : Orthogonalization of Concerns and Platform-Based Design. 19(12), 2000.

[102] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. *ACM Computing Surveys*, 28(June):220–242, 1997.

[103] Bart Kienhuis, EdF. Deprettere, Pieter Wolf, and Kees Vissers. A methodology to design programmable embedded systems. In EdF. Deprettere, Jürgen Teich, and Stamatis Vassiliadis, editors, *Embedded Processor Design Challenges*, volume 2268 of *Lecture Notes in Computer Science*, pages 18–37. Springer Berlin Heidelberg, 2002.

[104] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*, volume 83 of *Object Technology Series*. Addison-Wesley, 2003.

[105] Tim Kogel, Malte Doerper, Torsten Kempf, Andreas Wieferink, Rainer Leupers, and Heinrich Meyr. Virtual architecture mapping: a systemc based methodology for architectural exploration of system-on-chips. *IJES*, 3(3):150–159, 2008.

[106] Ali Koudri, Joël Champeau, Jean-Christophe Le Lann, and Vincent Leilde. Mopcom methodology: Focus on models of computation. In *Proceedings of the 6th European Conference on Modelling Foundations and Applications*, ECMFA'10, pages 189–200, Berlin, Heidelberg, 2010. Springer-Verlag.

[107] G. Kramler, G. Kappel, T. Reiter, E. Kapsammer, W. Retschitzegger, and W. Schwinger. Towards a semantic infrastructure supporting model-based tool integration. In *Proceedings of the 2006 international workshop on Global integrated model management*, GaMMa '06, pages 43–46, New York, NY, USA, 2006. ACM.

[108] Agnes Lanusse, Yann Tanguy, Huascar Espinoza, Chokri Mraidha, Sebastien Gerard, Patrick Tessier, Remi Schnekenburger, Hubert Dubois, and François Terrier. Papyrus uml: an open source toolset for mda. In Richard F. Paige, A. Hartman, and Arend Rensink, editors, *ECMDA-FA 09: Model driven architecture - foundations and applications: 5th European conference, ECMDA-FA 2009, Enschede, the Netherlands, June 23-26, 2009 ; proceedings*, volume 5562 of *Lecture Notes in Computer Science*, page 14, Berlin and New York, 2009. Springer.

[109] Diego Latella, Istvan Majzik, and Mieke Massink. Towards a formal operational semantics of uml statechart diagrams. In *Formal Methods for Open Object-Based Distributed Systems*, pages 331–347. Springer, 1999.

[110] Jirí Lebl. Ordinary differential equations. *Methods in molecular biology (Clifton, N.J.)*, 930(January):475–98, 2013.

[111] Henry Ledgard. *Reference Manual for the ADA Programming Language*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1983.

[112] Edward A Lee. Finite State Machines and Modal Models in Ptolemy II. Technical report, University of California at Berkeley, 2009.

[113] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75:1235–1245, September 1987.

[114] Edward A Lee and Thomas M Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[115] Edward A Lee and Alberto Sangiovanni-Vincentelli. A framework for comparing models of computation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(12):1217–1229, 1998.

[116] E Lenormand and G Edelin. An Industrial Perspective : A pragmatic High end Signal processing Design Environment at Thales, 2003.

[117] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1997.

[118] P. Liggesmeyer and M. Trapp. Trends in Embedded Software Engineering. *IEEE Software*, 26(3), 2009.

[119] Jane W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[120] Jie Liu. Continuous time and mixed-signal simulation in ptolemy ii. Technical report, Dept. of EECS, University of California, Berkeley, CA, 1998.

[121] Jie Liu, Xiaojun Liu, and Edward A. Lee. Modeling distributed hybrid systems in Ptolemy ii. In *In Proceedings of the American Control Conference*, pages 4984–4985, 2001.

[122] Marc Lobelle. VME bus interfacing: A case study, 1983.

[123] Janne Luoma, Steven Kelly, and Juha pekka Tolvanen. Defining domain-specific modeling languages: Collected experiences. In *In Proceedings of the 4th OOPSLA Workshop on Domain-Specific Modeling (DSM04*, 2004.

[124] Jean marie Favre. Towards a basic theory to model model driven engineering. In *In Proc. of the UML2004 Int. Workshop on Software Model Engineering*, 2004.

[125] Simon Marlow. Haskell 2010 Language Report. *Language*, page 329, 2010.

[126] Grant Martin. Overview of the mpsoc design challenge. In *Proceedings of the 43rd Annual Design Automation Conference*, DAC '06, pages 274–279, New York, NY, USA, 2006. ACM.

[127] Grant Martin, Brian Bailey, and Andrew Piziali. *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[128] Grant Martin and Gary Smith. High-Level Synthesis: Past, Present, and Future. *IEEE Design & Test of Computers*, 26(4):18–25, 2009.

[129] ARTEMIS Project MBAT. Combined Model-based Analysis and Testing of Embedded Systems. `https://www.mbat-artemis.eu/home/`.

[130] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, January 2000.

[131] Stephen J. Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. Model-Driven Architecture. In Jean-Michel Bruel and Zohra Bellahsene, editors, *Advances in Object-Oriented Information Systems*, volume 2426 of *Lecture Notes in Computer Science*, pages 290–297. Springer Berlin Heidelberg, 2002.

[132] StephenJ. Mellor, Stephen Tockey, Rodolphe Arthaud, and Philippe Leblanc. An action language for uml: Proposal for a precise execution semantics. In Jean Bézivin and Pierre-Alain Muller, editors, *The Unified Modeling Language. UML98: Beyond the Notation*, volume 1618 of *Lecture Notes in Computer Science*, pages 307–318. Springer Berlin Heidelberg, 1999.

[133] Aimé Mokhoo Mbobi, Frédéric Boulanger, and Mohamed Feredj. An Approach of Flat Heterogeneous Modeling based on Heterogeneous Interface Components. *International Review on Computers and Software (IRECOS)*, 2(2):179–189, March 2007.

[134] Wolfgang Mueller, Rainer Dömer, and Andreas Gerstlauer. The formal execution semantics of specc. In *IN PROCEEDINGS OF THE INTERNATIONAL SYMPOSIUM ON SYSTEM SYNTHESIS*, 2002.

[135] Lukito Muliadi. Discrete event modeling in Ptolemy II. Master's report, Dept. of EECS, University of California, Berkeley, CA, 1999.

[136] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In *Proc. of MODELS/UML*, LNCS, Montego Bay, Jamaica, 2005. Springer.

[137] Jonathan Musset, Étienne Juliot, Stéphane Lacrampe, William Piers, Cédric Brun, Laurent Goubet, Yvan Lussaud, and Freddy Allilaire. Acceleo user guide, 2006.

[138] R. Nikhil. Bluespec System Verilog: efficient, correct RTL from high level specifications. *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.*, 2004.

[139] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere. Daedalus: Toward composable multimedia MP-SoC design. *2008 45th ACM/IEEE Design Automation Conference*, 2008.

[140] O M G Document Number and Superstructure Associated Files. OMG Unified Modeling Language TM (OMG UML), Superstructure. *InformatikSpektrum*, 21:758, 2010.

[141] Roman Obermaisser, Christian El Salloum, Bernhard Huber, and Hermann Kopetz. Modeling and verification of distributed real-time systems using periodic finite state machines. *Comput. Syst. Sci. Eng.*, 23(4), 2008.

[142] Object Management Group. Meta object facility (MOF) 2.0 core specification. Technical Report formal/06-01-01, Object Management Group, 2001. OMG Available Specification.

[143] Object Management Group. Interface definition language, version 2.0. `http://www.omg.org/gettingstarted/omg_idl.htm`, 2003.

[144] Object Management Group. Spem 1.1. Technical Report ptc/05-01-06, Object Management Group, 2005.

[145] KTH Royal Institute of Technology. Introduction to forsyde. `http://www.ict.kth.se/forsyde/files/tutorial/ar01s03.html`.

[146] OMG. Object Management Group. `http://www.omg.org/`.

[147] OMG. Common Warehouse Metamodel (CWM) Specification Volume 2 . Extensions, 2001.

[148] OMG. UML Profile for MARTE, Beta 1. `http://www.omg.org/omgmarte/Documents/Specifications/08-06-09.pdf`, 2007.

[149] OMG. *XML Metadata Interchange (XMI)*. OMG, 2007.

[150] OMG. Mofm2t 1.0. `http://www.omg.org/spec/MOFM2T/1.0/`, 2012.

[151] Object Management Group OMG. Mof 2.0 core final adopted specification. `http://www.omg.org/cgi-bin/doc?ptc/03-10-04`, 2004.

[152] Object Management Group OMG. Spt 1.1 uml profile for schedulability, performance, and time. `http://www.omg.org/spec/SPTP/`, 2005.

[153] Object Management Group OMG. OCL : Object Constraint Language. *Language*, 36:1–11, 2012.

[154] Object Management Group OMG. Alf 1.0.1 action language for foundational uml (alf). `http://www.omg.org/spec/ALF/`, 2013.

[155] Object Management Group OMG. fuml 1.1 semantics of a foundational subset for executable uml models. `http://www.omg.org/spec/FUML/`, 2013.

[156] Qvt Omg. Meta Object Facility (MOF) 2 . 0 Query / View / Transformation Specification. *Transformation*, (January):1–230, 2008.

[157] P.R. Panda. SystemC - a modeling platform supporting multiple design abstractions. *International Symposium on System Synthesis (IEEE Cat. No.01EX526)*, 2001.

[158] Joann M. Paul, Donald E. Thomas, and Andrew S. Cassidy. High-level modeling and simulation of single-chip programmable heterogeneous multiprocessors. *ACM Trans. Des. Autom. Electron. Syst.*, 10(3):431–461, July 2005.

[159] Vincent Perrier. A look inside electronic system level (ESL) design. `http://www.eetimes.com/document.asp?doc_id=1276969`.

[160] Douglas L. Perry. *VHDL*. McGraw-Hill, Inc., New York, NY, USA, 3rd edition, 1998.

[161] James L. Peterson. Petri nets. *ACM Comput. Surv.*, 9(3):223–252, September 1977.

[162] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Universität Hamburg, 1962.

[163] Andy D. Pimentel, Todor Stefanov, Hristo Nikolov, Mark Thompson, Simon Polstra, and Ed F. Deprettere. Tool integration and interoperability challenges of a system-level design flow: A case study. In *SAMOS*, pages 167–176, 2008.

[164] GD Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61(January):17–139, 1981.

[165] Gordon D. Plotkin. The origins of structural operational semantics, 2004.

[166] JD Poole. Model-driven architecture: Vision, standards and emerging technologies. *... on Metamodeling and Adaptive Object Models ...*, (April):1–15, 2001.

[167] INTERESTED EU Project. Interoperable embedded systems Toolchain for enhanced rapid design, prototyping and code generation. `http://www.esterel-technologies.com/news-events/press-releases/eus-interested-project-concludes-goals-achieved/`.

[168] J. Rader, E.J. Morris, and A.W. Brown. An investigation into the state-of-the-practice of CASE tool integration. *1993 Software Engineering Environments*, 1993.

[169] Clemens Reichmann, Markus Kühl, Philipp Graf, and Klaus D. Müller-Glaser. Generalstore - a case-tool integration platform enabling model level coupling of heterogeneous designs for embedded electronic systems. In *ECBS*, pages 225–232. IEEE Computer Society, 2004.

[170] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio. A model-driven design environment for embedded systems. *2006 43rd ACM/IEEE Design Automation Conference*, 2006.

[171] Scott Rich. Ibm's jazz integration architecture: building a tools integration architecture and community inspired by the web. In Michael Rappa, Paul Jones, Juliana Freire, and Soumen Chakrabarti, editors, *WWW*, pages 1379–1382. ACM, 2010.

[172] Mark Richters and Martin Gogolla. OCL: Syntax, Semantics, and Tools. *Object Modeling with the OCL*, 2263(July 1997):42–68, 2002.

[173] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[174] Ingo Sander and Axel Jantsch. System modeling and transformational design refinement in ForSyDe. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(1):17–32, January 2004.

[175] A. Sangiovanni-Vincentelli. Quo Vadis, SLD? Reasoning About the Trends and Challenges of System Level Design. *Proceedings of the IEEE*, 95(3), 2007.

[176] Alberto L. Sangiovanni-Vincentelli, Sandeep K. Shukla, Janos Sztipanovits, Guang Yang, and Deepak Mathaikutty. Metamodeling: An emerging representation paradigm for system-level design. *IEEE Design & Test of Computers*, 26(3):54–69, 2009.

[177] Douglas C Schmidt. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, 39(2):25, 2006.

[178] Dana Scott and Christopher Strachey. Toward A Mathematical Semantics for Computer Languages. In Jerome Fox, editor, *Proceedings of the Symposium on Computers and Automata*, volume XXI, pages 19–46. Polytechnic Press, 1971.

[179] Dana S. Scott. Domains for denotational semantics. *Automata, Languages and Programming*, 140:577–610, 1982.

[180] Sandeep K. Shukla. Model-Driven Engineering and Safety-Critical Embedded Software, 2009.

[181] Carl Smith. *A Recursive Introduction to the Theory of Computation*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1994.

[182] Neil Smyth. *Communicating Sequential Processes Domain in Ptolemy II*. Electronics Research Laboratory, College of Engineering, University of California, 1998.

[183] SODIUS. Mdworkbench platform. `http://www.mdworkbench.com`.

[184] J.A. Stankovic. Misconceptions about real-time computing: a serious problem for next-generation systems, 1988.

[185] Guy L. Steele and Richard P. Gabriel. The evolution of Lisp, 1993.

[186] Ensta Bretagne STIC-IDM. Mdworkbench transformation rules defined with cometa. `https://gforge.ensieta.ecole/svn/cometa`, 2011-2013.

[187] Neil R. Storey. *Safety Critical Computer Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[188] B Stoustrup. *The C++ programming language*, volume 78. Prentice Hall, 1997.

[189] J. Sztipanovits and G. Karsai. Model-integrated computing. *Computer*, 30(4), 1997.

[190] Donald E. Thomas and Philip R. Moorby. *The VERILOG Hardware Description Language*. Kluwer Academic Publishers, Norwell, MA, USA, 3rd edition, 1996.

[191] I. Thomas and B.A. Nejmeh. Definitions of tool integration for environments. *IEEE Software*, 9(2), 1992.

[192] M Timmerman. Embedded Systems: Definitions, Taxonomies, Field, 2007.

[193] Andreas Tolk and James Muguira. The Levels of Conceptual Interoperability Model. In *Fall Simulation Interoperability Workshop*, pages 1–9, 2003.

[194] Frank Truyen. The fast guide to model driven architecture: The basics of model driven architecture. *URL: http://www. omg. org/mda/presentations. htm, January*, 2006.

[195] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. 42:230–265, 1936.

[196] A. Vachoux, C. Grimm, and K. Einwich. SystemC-AMS requirements, design objectives and rationale. *2003 Design, Automation and Test in Europe Conference and Exhibition*, 2003.

[197] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000.

[198] Dániel Varró and András Pataricza. Metamodeling mathematics: A precise and visual framework for describing semantics domains of UML models. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, UML '02, pages 18–33, London, UK, UK, 2002. Springer-Verlag.

[199] Simulink Design Verifier. Simulink ® Verification and Validation . *Update*, pages 1–16, 2011.

[200] J. Vidal, F. de Lamotte, G. Gogniat, P. Soulard, and J.-P. Diguet. A co-design approach for embedded system modeling and code generation with UML and MARTE. *2009 Design, Automation & Test in Europe Conference & Exhibition*, 2009.

[201] A.S. Vincentelli and J. Cohn. Platform-based design. *IEEE Design e Test*, 18(6):23–33, 2001.

[202] Wenguang Wang, Andreas Tolk, and Weiping Wang. The levels of conceptual interoperability model: Applying systems engineering principles to m&s. In *Proceedings of the 2009 Spring Simulation Multiconference*, SpringSim '09, pages 168:1–168:9, San Diego, CA, USA, 2009. Society for Computer Simulation International.

[203] AnthonyI. Wasserman. Tool integration in software engineering environments. In Fred Long, editor, *Software Engineering Environments*, volume 467 of *Lecture Notes in Computer Science*, pages 137–149. Springer Berlin Heidelberg, 1990.

[204] Stephen A White. Introduction to BPMN. *BPTrends*, (July):1–11, 2004.

[205] Roel Wieringa. *Design Methods for Reactive Systems: Yourdan, Statemate, and the UML*. Morgan Kaufmann Publishers, Boston, 2003.

[206] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA, 1993.

[207] Zhiru Zhang and Deming Chen. Challenges and opportunities of esl design automation. In *Solid-State and Integrated Circuit Technology (ICSICT), 2012 IEEE 11th International Conference on*, pages 1–4. IEEE, 2012.

[208] Richard Zurawski, editor. *Embedded Systems Handbook*. CRC Press, 2005. Industrial Information Technology.

Appendices

A

Appendix

A.1 Metamodel Excerpts

A.1.1 Spear Excerpt

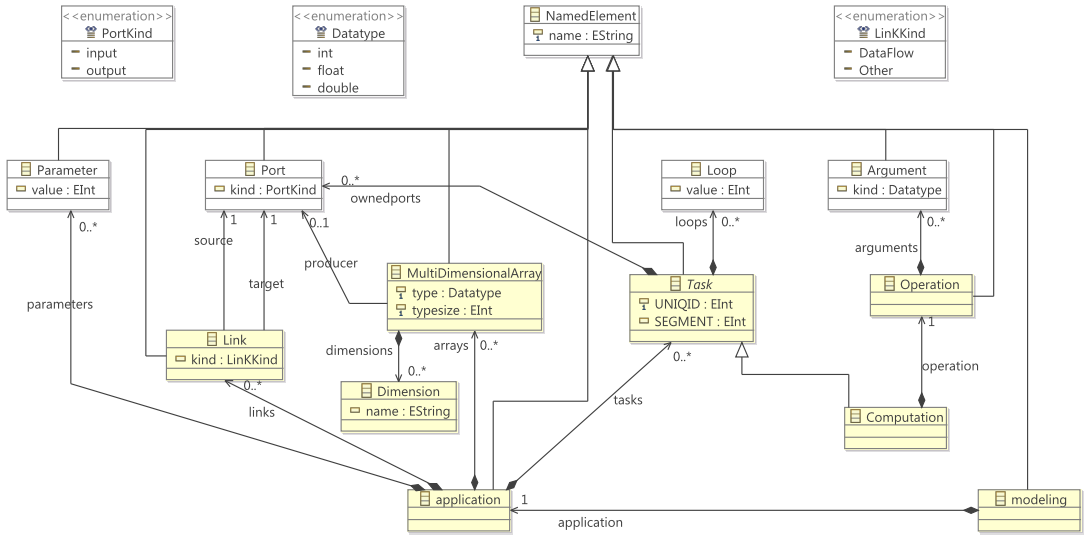

Figure A.1: Excerpt of the Spear Metamodel used for Model Transformation with Cometa

A.1.2 ForSyDe front-end Metamodel Excerpt

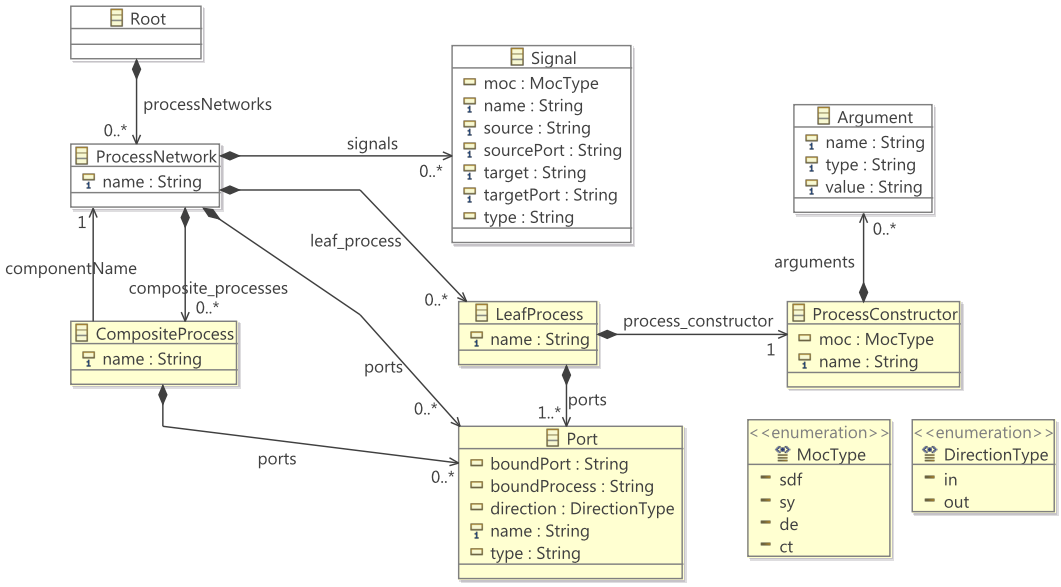

Figure A.2: Excerpt of the ForSyDe Metamodel used for Model Transformation from Cometa to ForSyDe-SystemC

A.1.3 NoC Mapping Metamodel Excerpt

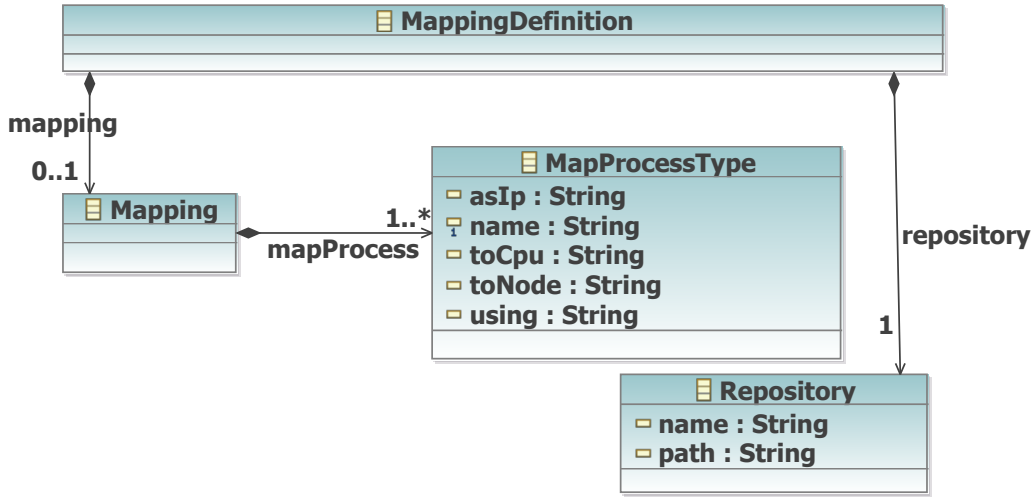

Figure A.3: Excerpt of the Metamodel used for the mapping of SW Processes into HW Architecture

A.1.4 NoC Generator Metamodel Excerpt

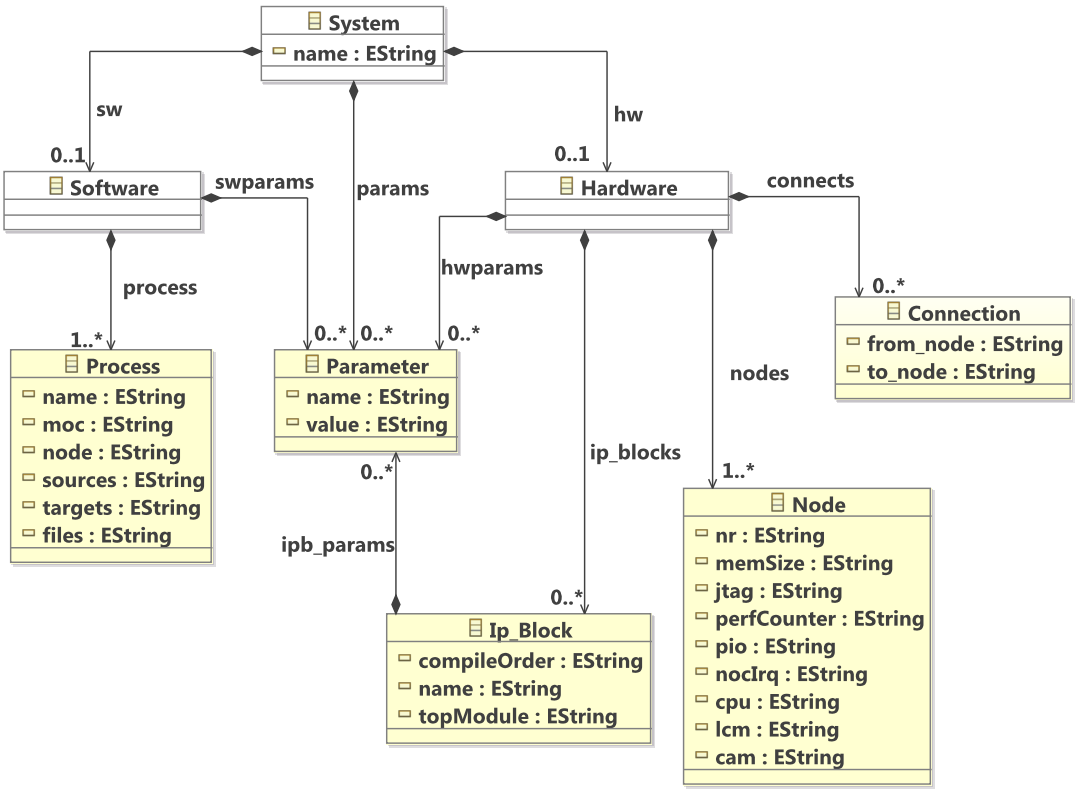

Figure A.4: Excerpt of the Metamodel used for the description of the NoC Generator

A.2 Sample of Models

A.2.1 Sample of Mapping Model for the UseCase

MDW: ForSyde BackEnd Mapping Model

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.1" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmlns:mapping="
    http://www.mdworkbench.com/mapping">
  <mapping:Mapping>
    <mapProcess name="CalibrationCorrectionAlloc" toNode="0"/>
    <mapProcess name="PulseCompressionAlloc" toNode="0"/>
    <mapProcess name="CFAR_ProcessingAlloc" toNode="0"/>
    <mapProcess name="DopplerMeasureAlloc" toNode="1"/>
    <mapProcess name="BeamFormingAlloc" toNode="1"/>
    <mapProcess name="AntennaAlloc" toNode="2"/>
    <mapProcess name="DigitalAnalogConverterAlloc" toNode="2"/>
    <mapProcess name="AnalogDigitalConverterAlloc" toNode="2"/>
    <mapProcess name="TransmitterAlloc" toNode="2"/>
  </mapping:Mapping>
  <mapping:Repository name="Repository" path="C:/NoC_v52/Examples/ENSTA"/>
</xmi:XMI>
```

Listing A.1: Sample of Mapping Model for the UseCase