Simulation of dynamically structured systems in a model-based system engineering approach applied to reconfigurable hardware

Min Zhu

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ FÉDÉRALE TOULOUSE MIDI-PYRÉNÉES

Délivré par :

l'Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)

Présentée et soutenue le *1/10/2018* **par :**

MIN ZHU

Simulation de systèmes à structure dynamique dans une approche d'ingénierie système basée modèles appliquée au matériel reconfigurable

---

JURY

M. VINCENT ALBERT	Université Toulouse III	Examinateur
M. CLÉMENT FOUCHER	Université Toulouse III	Directeur de thèse
MME. CLAUDIA FRYDMAN	Université Aix-Marseille III	Rapporteuse
M. FABRICE MULLER	Univesité Nice Sophia Antipolis	Examinateur
M. ALEXANDRE NKETSA	Université Toulouse III	Directeur de thèse
M. SÉBASTIEN PILLEMENT	Université de Nantes	Rapporteur

École doctorale et spécialité :

EDSYS : Informatique 4200018

Double mention :

EDSYS : Systèmes embarqués 4200046

Unité de Recherche :

Laboratoire d'analyse et d'architecture des systèmes

Directeurs de Thèse :

Monsieur Clément FOUCHER et *Monsieur Alexandre NKETSA*

Rapporteurs :

Madame Claudia FRYDMAN et *Monsieur Sébastien PILLEMENT*

Scientists are not dependent on the ideas of a single man, but on the combined wisdom of thousands of men, all thinking of the same problem, and each doing his little bit to add to the great structure of knowledge which is gradually being erected.

Ernest Rutherford

Acknowledgements

This thesis represents the final report of my time spent at *System Engineering and Integration* team of *Crucial Computing* department at *Laboratory for Analysis and Architecture of Systems*, Toulouse. I would like to take the opportunity to thank many of the people who have supported this thesis and influenced the creation of this work.

I am deeply grateful to Prof. Alexandre NKETSA, for his open-mindedness and critical advice during the discussions. I am very thankful to Clément FOUCHER, for giving constructive guidance thoughout and being an exemplar of a professional researcher. I also would like to specially thank Vincent ALBERT, for passing on his wisdom during the model construction. Their help and supervision made this piece of work possible.

I give full appreciation to Prof. Claudia FRYDMAN and Prof. Sébastien PILLEMENT, for their interest in my work and for refereeing this thesis.

I would like to thank to Philippe ESTEBAN, who took me into this adventure of scientific research after my Master studies of real-time system engineering and stood by me during my three years teaching assistance work. Thank to Hamid DEMMOU for his team leads which make my international presentation possible. Thanks also to Claude BARON for her excellent team building.

I would like to thank all my colleagues at faculté sciences et ingenierie at université Paul Sabatier. Thanks to Emmanuel Montseny for all the preparation made for the practice of Matlab/Simulink. I have shared lots of good moment with the others, especially thanks to the every year Circus.

I would like to give a special thanks to Hélène THIRION, Christèle MOUCLIER, Layla MOURCHID and Catherine GUERIN for their ongoing administrative support and for always giving a helping hand.

This thesis would not be possible without the support of families and friends, without the happiness they brought, sharing cafe culture, music, and good times spent outside of research in Finland, in Mexico, in Portugal, in China. Thank you for your great efforts people around me! Sylvain, David, Diego, Xue Rui, Karla, Guillaume, Yi Xin, Lily, Sangeeth, Yassine, Adina, Daniel et Julie, Violaine et Rémi, He Yun, Wang Rui, for all your emotional support and just being there when I need you.

In the end, I would like to give some very special thanks to the international exchange program ERASMUS, with whom I finished my undergraduate studies in France, and without whom I could not have achieved what I have today.

Min, 21 June 2018, at Toulouse

Abstract:

As partially reconfigurable technologies develop for embedded systems, the need for a proper model to describe its behavior emerges. Most academic and industrial tools available on the market does not address dynamic structure modeling. The arising of discrete-event modeling, in particular, Discrete Event System Specification (DEVS), propose formal tools for representing and simulating models. DEVS has already extension which handles the dynamic structure modeling. However, the capacities of these existing formalism have limitations. Notably, they do not address the components context aspect.

Also, the existing formalisms have not integrated the system engineering approach. System engineering brings beneficial procedures, notably model-driven architecture which proposes to separate the system description from its execution target. A platform-specific model is formed from a platform-description model coupled with a platform independent model.

To address these needs, we propose a model description formalism which takes into consideration these two aspects: dynamic structure modeling and system engineering. This formalism is based on DEVS and called Partially Reconfigurable Discrete Event System Specification (PRDEVS). PRDEVS allows to represent dynamic-structure models independently from the simulation platform.

The presented approach can be applied to different types of targets, such as software and reconfigurable hardware. This thesis addresses these two kinds of platforms, demonstrating the suitability of the abstract formalism to actual platforms.

Keywords: Reconfigurable hardware systems, Modeling, Discrete event simulation, Model-based system engineering, Model-driven architecture

Résumé :

Avec l'évolution des techniques de reconfiguration partielle pour les systèmes embarqués, le besoin d'un modèle de description capable de représenter ces comportements émerge. La plupart des outils disponibles sur le marché, tant académiques qu'industriels, ne prennent pas en compte la modélisation des systèmes à structure dynamique. L'émergence de la modélisation à évènements discrets, notamment Discrete Event System Specification (DEVS), propose des outils formels pour représenter et simuler des modèles. DEVS propose déjà des extensions capable de prendre en compte la modélisation à structure dynamique. Néanmoins, les possibilités offertes par ces extensions rencontrent certaines limites. En particulier, elles ne proposent pas de moyen de gérer l'aspect contexte des composants.

De plus, les formalismes existants n'ont pas intégré l'approche ingénierie système. L'ingénierie système met en place des procédures intéressantes, notamment l'architecture dirigée par les modèles, qui propose de séparer la description du système de sa plateforme d'exécution. Un modèle spécifique à une plateforme est ainsi la résultante d'un modèle de description de la plateforme combiné avec un modèle d'application indépendant de toute plateforme.

Pour répondre à ces besoins, nous proposons un formalisme de description de modèles prenant en compte ces deux aspects : la modélisation à structure dynamique, et l'ingénierie système. Ce formalisme est basé sur DEVS, et nommé Partially Reconfigurable Discrete Event System Specification (PRDEVS). PRDEVS permet de représenter les modèles à structure dynamique indépendamment de la plateforme de simulation.

L'approche présentée peut être appliquée à différents types de cibles, tels le logiciel et le matériel reconfigurable. Cette thèse présente des mises en œuvre du formalisme abstrait sur ces deux types de plateformes, démontrant ainsi sa capacité à être déployé sur des plateformes réelles.

Mots clés : Systèmes matériels reconfigurables, Modélisation, Simulation à évènements discrets, Ingénierie système basée modèles, Architecture dirigée par les modèles

Contents

List of Figures

Introduction

Mankind has created tools of all kinds to accomplish those tasks which are difficult to achieve directly by our own hands. The ambitions of human beings, together with these technologies, make it possible to fly and travel into space. However, the creation of a new system is generally not as straightforward as it might seem.

Human-made systems, from vehicles to satellites, interact with their environments. The interactions between a system and its surrounding lead to different additional performance factors. The environment impacts the system properties. At the same time, the system can change its environment. For example, aircrafts lift comes from the air and the acceleration of the aircraft is limited by the air resistance at the same time. Building a system requires a full-scale consideration of all such possible interactions.

Nowadays, the creation of an artificial system is sometimes so complex that it involves interdisciplinary expert cooperation. Such a complexity makes it difficult to estimate the delays, the costs, and even feasibility. Moreover, the development of a new product can take several years. A bigger team and a longer process add complexity to a project. New methods are required to adapt to this situation, notably to identify and improve common practices that exist across the development of a wide variety of systems. A double verification or a cross-validation alone is not enough to guarantee the success of a complex project.

Verification and validation are done throughout the creation of a new product. However, there can be multiple iterations over the process, and creating a product prototype each time for the verification adds to the cost and timescale. For example, a printed circuit board can take more than a week to be delivered. A verification of the virtual product using computer simulation has advantages in terms of cost and time. Lower costs and/or faster results make simulation an interesting tool for engineers. Thus, simulation is a major verification and validation method applied to the product lifecycle.

A simulation is generally done on a digital processing unit, e.g. a processor, and thus requires a digital model. A model is a representation of an actual object with a certain level of precision. Following the product engineering process, the simulation model is then an abstraction of the functions which are concerned. Moreover, a model level of representation depends on the underlying formalism. Using a very formal meta-model allows for precise and verifiable models.

Motivations

Formal meta-models consist in providing a mathematically-based syntax, a comprehensive semantic. Doing so enforces replicable simulations given the same models and an identical initial state.

Discrete event simulation is a paradigm in which the temporal evolution of the simulation is led by the events. In such a simulation, events are identified by the time in future at which they are meant to happen. This contrasts with discrete time simulations, in which the time advance is fixed and steady.

Discrete event simulation has advantages in certain cases over discrete time simulation. For example, for a system in which the time constant varies from long times to very short times. In such a case, the time step chosen can be very small to be able to account for fast variations, but this will trigger unnecessary computations for steady periods. On the other hand, if the time step chosen is longer to accelerate the simulation, one can miss fast variations which can occur between two computation slots. A discrete event simulation is able to use the derivative of the variable to determine the time of next event, so intervals are set according to the rate of change.

Formalisms like Discrete Event System Specification (DEVS) allow us to modeling discrete event systems. DEVS has a formal syntax, mathematically verifiable, and strict algorithms for simulating these models. Moreover, DEVS allows for model assemblies, which are required to build complex systems and is closed under coupling, i.e. a component made of sub-components externally exhibits a behaviour which can be represented by an atomic component. DEVS is thus a strong formalism for building simulations of complex systems using a discrete event paradigm.

But when it comes to representing dynamic structure systems, DEVS extensions related to such systems have limitations. DEVS formalisms for dynamic structure systems mostly rely on predetermined architectural states. Each of these states represents a model architecture, containing components and links between components. When a transition function is triggered, the model switches from one architectural state to another. The simulation environment is then responsible for inferring the actual simulation architecture from the state evolution.

While in many cases this will be enough to represent a dynamic structural system, there are other cases in which this will be a limitation. An example of this might be multi-agent systems, such as population simulation, in which agents can be born or die at some point in the simulation. In such cases, there is no way of knowing all possible architectural states of the system before simulation. We rather require a way of representing agent creation and deletion from the system, as well as relationship evolution between agents, directly in the formalism. Another field of application is complex adaptive systems, in which the network of interactions between components of the system evolve and is able to reach net structures impossible to predict.

Moreover, having a strict approach always leads to better results. The system engineering approach provides several tools for better handling system creation, including the simulation steps. One interesting concept of system engineering is the model-driven architecture approach. Initially created for software design, model-driven architecture consists of separating the application design from the execution platform. It means that the application must be written using a platform-independent representation, and must incorporate a transformation tool able to generate an executable application for various platforms. This requires a high-level, platform-agnostic, model of the system and models of the platforms that will constitute the execution target. Then, by establishing a correspondence of the models, we establish a model that is able to run on a specific platform.

Among execution platforms, we usually find the classic software processors, nowadays constituted of multiple execution cores. Recently, enthusiasm for even more parallel platforms such as graphics processing units has arisen. This kind of device, initially designed for graphics applications where the same treatment must be operated on multiple pixels, has gained a general purpose use over recent years. When application speed and/or current consumption is critical, building an Application-Specific Integrated Circuit (ASIC) is also worth considering. ASICs consist in creating a logic circuit which realizes the exact function required, allowing for very important optimizations compared to software, which requires generalisation. However, the significant of fixed costs of such devices only makes them suitable for mass production.

In the middle of all theses lie Field-Programmable Gate Arrays (FPGAs). Not as fast as ASICs, not as flexible as processors, but faster than processors and more flexible than ASICs. The FPGA represents a very interesting execution target for applications that can benefit on-demand parallelism. FPGAs being reprogrammable logic circuits, they also benefit from unique features such as partial dynamic reconfiguration. This features consists in modifying a portion of the circuit while the remaining part continues to operate, incarnating the concept of dynamic structure change. However, partial reconfiguration of FPGAs is still lacking formal methods for handling it.

Definition of objectives

At the time of writing, dynamic structure systems are new to modeling and simulation, and often treated without a clear methodology. They could benefit from of a system engineering approach. In this work, we are aiming to define a formal modeling approach to represent and simulate dynamic structure systems. To do so, we are going to formalize the representation of such systems into a meta-model. We will particularly focus on discrete-event models, and adopt a model-driven architecture approach.

As a first step, the meta-model, in other words, the model of the models must be defined. We need the meta-model to use the strong fondation of mathematical formalism. Moreover, we want the models we define to be free from constraints related to the execution platform. However, the definition must include the capability for dynamic structure change, which will restrict the available target platforms.

We also need to distinguish the dynamic structure behaviour from a model state evolving or being changed over time. The structure change takes place at a different level than the state change. Different components can exist in a system at one time, and exhibit the exact same internal behaviour while being in a different state. The state of the components thus represents an independent information set, which must be treated separately from the structure change.

The structure itself contains two levels of possible changes. A model structure is composed of components, which have relationships with each other. The change can be operated either on components or on connections. We will see that this differentiation has an impact on the way it is handled by the execution platform.

Concerning the components state management, it will be useful to develop the capacity to save the state of a component during simulation, or to force it into a specific state. For example, we should be able to save the current state of a component about to be deleted so that it can be restored later. Or, a component can be instantiated with different initial states depending on the general simulation state at the simulation time.

Thus, we should study different target platforms in order to understand their execution behaviour. This will enable us to make the connection between models expressed in the developed meta-model and the platforms. We will begin with a software platform as a first target. On a software platform, many constraints are flexible, such as the memory size which, to some extent, can be ignored. During this integration period, the system engineering approach will be validated for our meta-model.

As a final objective, the meta-model should be integrated into a dynamic hardware platform. With this specific platform, the constraints (space and time) of the platform should be defined. On such a level, the communication protocol and scheduling behaviour of the simulation should have been considered. The parallel nature of the platform must have been developed within our meta-model.

Structure of the document

In this thesis, we propose an interdisciplinary work between system engineering and modeling & simulation. The scientific context and state of the art are presented in the first chapter. It details the approach of system engineering and modeling, especially for discrete event simulation and dynamically reconfigurable computing systems.

In chapter two, we propose a dynamic structure meta-model: Partial Reconfigurable Discrete Event System Specification. We detail the meta-model syntax and semantics. We present the syntax using a mathematic description at a theoretical level. Then a graphic representation of the syntax is discussed to facilitate model representation.

In the third chapter, a software platform, with object-oriented programming, is presented to support the execution of models. We introduce a simulator which is able to run models described using the meta-model defined in chapter two.

In the fourth chapter, a hardware platform specification based on FPGA is presented. The platform architecture is defined in order to match the FPGA constraints.

Finally, we draw conclusions from the results and bring future perspectives into consideration.

Chapter 1

Scientific context & state of the art

Contents

System engineering consists in a process with several steps. The final goal of system engineering is to avoid failures while building a system. A complex system is usually decomposed into sub-systems, which are developed separately. The well-known V-model system engineering cycle, for example, begins with the feasibility study and concept exploration. It progressively goes down by breaking the system into sub-systems, which are fully specified individually. Finally, it ascends by testing system components and assembling them to form a complete system.

During the design of a system, modeling and simulation takes an important place. When a system is made of various sub-systems, all of them must be modeled as stand-alone pieces and tested individually. When bringing them together to form the whole system, how to co-simulate the independent models together is a really complex task. Co-simulation can happen between different types of models: hardware/software, continuous/discrete, electronic/digital, etc. It can also be done between different abstraction levels.

For classic systems, where the physical materials can be modeled, a software component can represent formally the physical materials. Before the simulation, an offline verification (static program analysis) is possible for the software components. The offline verification can be done using formal languages syntactically representing the system. Before the simulation, formal methods are used as critical tools for system verification, which guarantee mathematically the system is correct. If an error is detected during the static analysis with formal methods and debug, an accident can be avoided.

The simulation is not limited to dig into the inner behavior of the system. It also looks at the interaction with a surrounding environment. In order to correctly test the system, we need to put it into its environment. It can be done by modeling the environment and simulating it together with the system. Or more directly, we can put the simulated system directly in contact with the real environment. When the system is in the real environment, the simulation must respect real-time.

For a real-time system, the actuators, physical system, and its sensors can be involved as simulated elements or as real elements in the model. As an example an aircraft stress test for its wings can contain both virtual subsystems and real subsystems. The virtual subsystem may include the control system, and the real subsystem is the airplane wing. A common practice is to model the real actuators together with the simulated physic system and sensors.

What we are interested in is dynamic structure systems, where the composition can change over time. FPGA is an integrated circuit which can change its configuration to form various logic circuits. A custom design is thus possible depending on the project. Moreover, some classes of FPGA are capable of run-time re-configuration during the execution.

In this chapter, we are going to discuss three main topics: system engineering together with model-based system engineering in section 1.1, Discrete Event System Specifications and its extensions in section 1.2, and reconfigurable computing systems, especially Field-Programmable Gate Arrays (FPGAs) in section 1.3.

1.1 System engineering and simulation

In this section, we will define topics of expertise which are required for the further developments. First, we will give a definition of what is a system and clarify the system engineering process. Then modeling and simulation as a part of the system engineering process will be presented. Model-driven architecture is detailed later as a system engineering approach for the software development. At last, the syntax and semantics modeling notions are defined.

1.1.1 Systems and system engineering

There are lots of definitions of what is a system. An earlier definition was done in the 50s by Ludwig von Bertalanffy [Von Bertalanffy 1956], who contributed to general systems theory. Von Bertalanffy outlines systems inquiry into three major domains: Philosophy, science, and technology:

“The systems view is a world-view that is based on the discipline of SYSTEM INQUIRY. Central to systems inquiry is the concept of SYSTEM. In the most general sense, a system means a configuration of parts connected and joined together by a web of relationships. The Primer Group defines the system as a family of relationships among the members acting as a whole.” Here, a system is defined as elements in standing relationship.

The International Council on Systems Engineering (INCOSE) is an organization aiming at improving the systems engineering practices and education developed in 1995. Their definition is straightforward:

“A system is a construct or collection of different elements that together produce results not obtainable by the elements alone.”

A system is never with only one element inside, which mean there is a possibility to decompose the system into subsystems or include a system into a larger system.

For systems engineering, the INCOSE concept is practical and includes the concept of business:

“Systems Engineering integrates all the disciplines and specialty groups into a team effort forming a structured development process that proceeds from concept to production to operation. Systems Engineering considers both the business and the technical needs of all customers with the goal of providing a quality product that meets the user needs."

The French association of systems engineering (AFIS) gives a definition oriented toward industrial manufacture:

"The control of complex systems by manufacturers is essential to maintain and improve the positions of French and European industry in the global market for large systems, whatever the field: transport, space, defense, finance, security, health, energy... These systems involve many disciplines: mechanical engineering, electrical engineering, automatic engineering, civil engineering, software engineering, electronic engineering, chemical engineering, industrial engineering, subcontracting, production, maintenance, security... but also trade, marketing, customer relations, human factors sustainable development."

For sure, applying the system engineering concept to a complex system is interesting to ensure the success of the project. Rather than the quality, cost, delivery (QCD) approach which evaluate the results, system engineering focuses on the whole project procedure, internal communication and organization.

The well-known V-model development got this name by its V form and the final steps of its procedure: verification and validation. Even though recently a lot of projects are applying other methods such as waterfall model [Balaji 2012], spiral model [Boehm 1988], agile methods [Ambler 2004], etc.

Within the V-model development, reliability of the system is often done by modeling and verified by simulation. It can be a general model without coding, in some case done by experience and success stories. During the requirements and architecture step, a system with several subsystems is built. Detailed designs with each subsystem's specification are defined. Before the project moves to implementation, high-level modeling is already done.

After the implementation, during test and integration, simulation is done with different methods to ensure the subsystem is functional. Under system level verification and validation, the entire system is executed, analyzed and simulation is done on the entire system level.

The Model-Based System Engineering (MBSE) [Estefan 2007] concept was introduced by AW Wymore in 1993. It was popularized by INCOSE when it kicked off its MBSE Initiative in January 2007 [Friedenthal 2007]. The main idea is to replace the document exchange during systems engineering by creating and exploiting domain models [Friedenthal 2007].

1.1.2 Modeling and simulation

Modeling is building an abstraction of the real world, where changes of certain parameters are used to learn the way the system behaves. Simulation is based on a model which is built based on a real system, already existing or in the design phase. It shows the results by observing the model changes over time or in response to its environment.

As for the methodology for modeling and simulation, originally presented in 1976 [Zeigler 2000], it consists of two principal aspects:

Level of system specification - These are the levels at which we can describe how systems behave and the mechanisms that make them work the way they do.

M&S sets four levels of system knowledge :

Level	Name	What we know at this level
0	Source	What variables to measure and how to observe them
1	Data	Data collected from a source system
2	Generative	Means to generate data in a data system
3	Structure	Components coupled together to form a generative system

Systems specification formalisms - These are the types of modeling styles, such as continuous or discrete, that modelers can use to build system models.

A discrete system is one in which the state variables change only at a discrete set of points in time. A continuous system is one in which the state variables change continuously over time.

Under system theory [Zeigler 2000], *structure* – the inner constitution of a system – and *behavior* – its outer manifestations – are considered separately. For a time-based system, *behavior* and *structure* are connected by time-related parameters: the internal *structure* of a system includes its state, how one state transits to another state and the mapping between state and output. A well-defined *structure* helps to analyze and simulate its behavior.

Basic system concept considers the system as a black box, where we observe the output changing in reaction to the input event. After analysis, the output can be used to correct system input.

In this case, we can see the simulation as a closed loop system in the control theory, where results of one simulation cycle can impact the next simulation cycle.

Then, four classifications of looped simulation were proposed by Jens Eickhoff [Eickhoff 2009]: Model in the loop (MIL), Software in the loop (SIL), Processor in the loop (PIL) and Hardware in the loop (HIL). The model in the loop consists in building a model describing the behavior of the system to validate it by simulation. The model of the system is coupled to an environmental model to see if the system model meets the main requirements of the system. The software in the loop principle is to create a program that implements the model in a target language. This implementation can lead to bias due to implementation constraints of the model and language limitations. A new phase of validation and/or verification is needed to ensure semantic equivalence. A processor in the loop consists in validating the behavioral equivalence of the program after integration into the target processor. The environment remains simulated. Hardware in the loop is to use the final physical controller. The environment remains simulated but now responds in real time. The difference between PIL and HIL is [Mina 2016]: PIL is a test technique that allows designers to evaluate a controller, running in a dedicated processor or a plant which runs in an offline simulation platform. On the other side, HIL is an approach to test a plant or controller running on a digital platform which interacts with the real controller or plant.

1.1.3 Model-Driven Architecture

The Model-Driven Architecture (MDA) approach by Object Management Group [Object Management Group 2016], derived from Model-Driven Engineering (MDE), consists in separating the application model description from the execution platform.

A good software engineering flow offers lots of advantages. The most important advantage is to define different abstraction levels, for managing the complex applications. This brings various benefits, such as allowing the teams working on an application to be independent of the ones working on the platform, or enabling deploying an application built from a single model on various platforms.

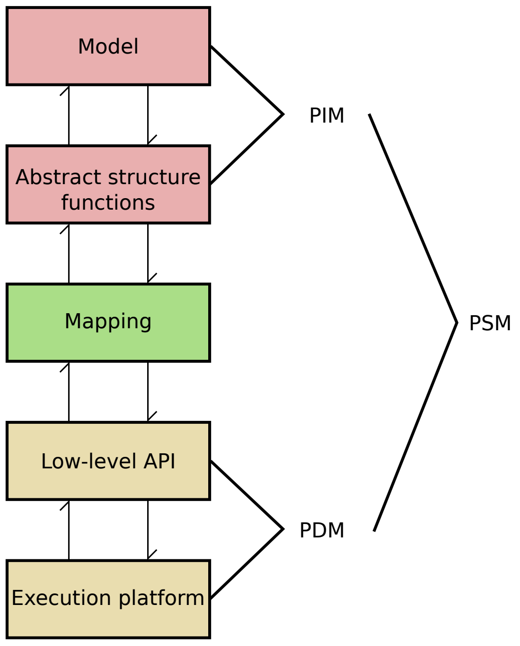


Figure 1.1: MDA structure of a meta-model

A complete MDA specification consists in a Platform-Independent Model (PIM), one or several Platform-Dependent Models (PDM), and sets of interfaces correspondence to allow building a Platform-Specific Model (PSM) by matching a PIM with a

PDM, as depicted in figure 1.1.

MDA is based on the massive use of models in all phases of the application life cycle. Standard MOF (Meta Object Facility) [Iyengar 2005] is the support for modeling formalisms under metamodel. MOF is used as the meta-metamodel not only for Unified Modeling Language (UML) but also for other languages, such as Common Warehouse Metamodel (CWM) [Poole 2003]. UML is defined as a model that is based on MOF. Every model element of UML is an instance of exactly one model element in MOF. A model is an instance of a metamodel. UML is a language specification (metamodel) from which users can define their own models.

With this architecture, the model can be built using different languages and translate into a unified model by mapping. In principle, an MDA is a framework for visualizing, storing and exchanging software designs and model [Kleppe 2003]. Since the framework separates the development in the first place, the PIM developers do not need to consider the platform details.

Within MDA, portability is realized by PIMs. One PIM can be deployed on multiple PSMs using different platforms PDMs. The MDA does not require a specific processes or languages for software development.

1.1.4 Syntax and semantics

The information and its meaning behind represent the different between syntax and semantics [Miller 1985, Dalrymple 1999, Harel 2000]. The information is represented as data while the data is the medium used to transport and store the information. There is general agreement in the literature that data is used to communicate and needs an interpretation to extract the information behind it [Harel 2000]. An interpretation is always a mapping assigning a meaning to each piece of data.

To correctly define a model, we need three levels of specification:

⋆ An abstract syntax, or meta-model, which is the formal definition used by the modeler to define its models.

The abstract syntax is used by the modeler to build and specify the model. Model description can be done theoretically, using mathematics tools such as sets and algorithms. But the actual model description provided to the simulator is often done using a Graphical User Interface (GUI), such as the ProDEVS environment [Vu 2015], to ease the process.

⋆ A concrete syntax, which is the actual model description matching the abstract syntax.

The concrete syntax is the way the model structure and its contents are stored and manipulated during the simulation. It must observe the abstract syntax, but its way of representing the model must be adapted to a digital representation and manipulation by the simulator. Note that this is the syntax representation which should comply with these constraints, not the model itself.

⋆ A semantics which specifies the model execution behavior by the mean of an abstract simulator.

The abstract simulator, indicating the semantics of how the model description is to be manipulated, and how the simulator should behave when a simulation event occurs.

1.2 DEVS and its extensions

Discrete Event System Specification (DEVS) is a formalism to describe discrete-event models and simulate them by proposing a syntax and a semantic. A DEVS model is a hierarchical set of components of two kinds: atomic components define a behavior while coupled components gather and link other components, either atomic or coupled. The original DEVS formalism was not designed to handle structure changes, either in model composition or communication. Dynamic structure behavior can only be emulated, e.g. using a selector to enable or disable models over time. Several extensions have been proposed addressing dynamic adaptation of the models structure during the simulation.

In this section, some generic definitions and background information about Discrete Event System Specification (DEVS) and its extensions from the industry and academia are discussed.

1.2.1 Discrete Event System Specification

DEVS formalism introduced by Zeigler [Zeigler 2000] is a strong mathematical foundation for specifying hierarchical and modular models. The DEVS formalism allows to build discrete event systems and provides algorithms for simulation. DEVS models are made of atomic components, which define a behavior, and coupled components which can hold several other components and describe the way they are connected.

DEVS was later extended with Parallel DEVS (PDEVS), and we now reference the initial DEVS formalism as Classic DEVS (CDEVS). In this thesis, the acronym DEVS thus refers to the general DEVS ecosystem rather than to the original CDEVS formalism.

1.2.1.1 Classic DEVS

Zeigler [Zeigler 2000] initially introduced the DEVS formalism in the late 70's as a way to build models with a discrete-event approach using a mathematically defined formalism.

CDEVS atomic components syntax CDEVS defines an atomic component as an indivisible unit implementing a behavior. It can evolve in reaction to an external event (external transition), or when a timeout occurs (internal transition). The formal definition is as follows:

$$M =< X, Y, S, \delta_{ext}, \delta_{int}, \lambda, \tau > \text{ where}$$

X $= \{(p, v) \mid p \in InPorts, v \in X_p\}$ is the set of input ports and values, where

- $InPorts$ is the set of input ports
- X_p is the set of allowed input values for port p

Y $= \{(p, v) \mid p \in OutPorts, v \in Y_p\}$ is the set of output ports and values, where

- $OutPorts$ is the set of output ports
- Y_p is the set of possible output values for p

S is the set of sequential states

δ_{ext} $: Q \times X \to S$ is the external state transition function, where

- $Q = \{(s, e) \mid s \in S, 0 \leq e \leq \tau(s)\}$ is the set of total states, with e the time elapsed since latest transition

δ_{int} $: S \to S$ is the internal state transition function

λ $: S \to Y$ is the output function

τ $: S \to \mathbb{R}^+_{0,\infty}$ is the time advance function (sometimes refered to as ta)

As simulated time advances, the e variable growths. An atomic component is said to be *imminent* when its remaining time in the current state $tr = \tau(s) - e$ is minimal among all the components in the simulation. Zeigler defines tl to be the time of the latest event that occured in the component, and $tn = tl + \tau(s)$ the scheduled time for the next event.

It is also of use to know the component initial state, which is usually refered to as s_0.

CDEVS coupled components syntax A coupled component is a way of linking other components. Externally, it behaves like an atomic component and thus can be used in another coupled to form a hierarchical model.

$$N =< X, Y, D, \{M_d\}, EIC, EOC, IC, Select > \text{ where}$$

X, Y as defined for atomics

D is the set of components names

$\{M_d\}$ is the set of components in this coupled, with $d \in D$

EIC is the external input coupling function

EOC is the external output coupling function

IC is the internal coupling function.

$Select$: $2^D - \{\} \rightarrow D$, the tie-breaking function

The coupling functions define links between different components ports. They are defined as:

EIC links $p_N \in InPorts_N$ to $p_d \in InPorts_d, d \in D$

EOC links $p_d \in OutPorts_d, d \in D$ to $p_N \in OutPorts_N$

IC links $p_a \in OutPorts_a$, $a \in D$ to $p_b \in InPorts_b$, $b \in D, a \neq b$

We notice here that no direct feedback loops are allowed, i.e. no output port of a component may be connected to an input port of the same component.

The *Select* function is used when various components are simultaneously imminent. In this case, the *Select* function defines a priority for the order in which the components will be processed.

CDEVS semantics Zeigler defines a complete semantics for CDEVS models. An atomic component is managed by a simulator, while a coupled component is managed by a coordinator.

First, the imminence of all components is checked by the root coordinator, and a list of imminent components is established. The *Select* function is used to determine the next component to be processed if the list contains more than one component.

An $*-message$ is sent to the most imminent component simulator, which activates it. A simulator receiving an $*-message$ will first perform its component λ emission then applies its δ_{int} internal transition. λ emission results in the creation of an $y-message$ which is transmitted to its parent coordinator. Finally, the simulator updates its tl and tn internal variables.

When a coordinator receives an $y-message$, it searches in its coupling lists the component to which it should be transmitted. If the component is contained in the coupled, it is converted to an $x-message$ and given to the component simulator. If the message is to be transmitted to an output port of the coupled, it is then sent to its parent.

When a simulator receives an $x-message$, it applies the δ_{ext} external transition of the component.

When there is no more action to perform, we move to the next iteration of the loop.

1.2.1.2 Parallel DEVS

Parallel DEVS (PDEVS) [Zeigler 2000] is the root formalism for DEVS extensions dealing with parallelism. It is strongly based on CDEVS, but it removes the *Select* function and replaces it with other mechanisms for handling simultaneous events. PDEVS thus allows different components to evolve simultaneously and provide resolution mechanisms to deal with conflicting simultaneous events.

PDEVS atomic components syntax PDEVS atomic component definition is as follows:

$$M =< X^b, Y^b, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, \tau > \text{ where}$$

The only addition to CDEVS is the δ_{con} function, defined as:

δ_{con} : $Q \times X \to S$ is the confluent transition function

When simultaneous external and internal events occur, the confluent function δ_{con} is called instead of δ_{int} or δ_{ext} to solve the conflict. δ_{con} can be as simple as calling δ_{int} or δ_{ext}, which is a way of prioritizing between these two functions or be a totally different function.

Moreover, the X and Y elements contain now not only simple values but bags of values, since several input or outputs can be received or sent simultaneously. This is why there are here noted as X^b and Y^b.

PDEVS coupled components syntax PDEVS coupled component is changed to:

$$N =< X, Y, D, \{M_d\}, EIC, EOC, IC > \text{ where}$$

The only difference with CDEVS is that the *Select* function has been removed, being no longer necessary as the parallelism is accepted.

1.2.2 Dynamic Structure DEVS and its parallel version

Dynamic Structure DEVS (DSDEVS), defined in [Barros 1997a], is based on a 4-tuple network structure where atomic components can connect directly with other atomic components by a set of influencers I. The network executive χ is a specific component whose state represents the network structure. χ thus takes responsibility for all changes of model structure, meaning that components in the model cannot take the decision on structural adaptation. Parallel Dynamic Structure DEVS (DSDE) [Barros 1998] is a parallel version of DSDEVS.

1.2.2.1 Dynamic Structure DEVS

Dynamic Structure DEVS allows changes in model structure during execution. It has the same basic model as CDEVS, but the structure of coupled models can change over time. The atomic component of DSDEVS inherit all definitions of CDEVS, while the coupled components have a different syntax adapting to dynamic structure.

DSDEVS atomic components syntax In DSDEVS, the atomic components are definied as CDEVS atomic components.

DSDEVS coupled components syntax In DSDEVS, the coupled models can be seen as sets of standard coupled models, each coupled representing a possible configuration of the network. A DSDEVS coupled model is called a dynamic structure network, and is defined by Barros as:

$$DSDEVN_{\Delta} =< X_{\Delta}, Y_{\Delta}, \chi, M_{\chi} > \text{ where}$$

Δ is the network name

$X_{\Delta}, Y_{\Delta} \equiv X, Y$ in DEVS

χ is the DSDEVS network executive name

M_{χ} is the model of the executive χ

The DSDEVS networks is defined with a special component, the network executive χ. M_{χ} the model of the executive, is a DEVS basic model and is defined by the structure:

$$M_{\chi} =< X_{\chi}, S_{\chi}, Y_{\chi}, \delta_{int_{\chi}}, \delta_{ext_{\chi}} \lambda_{\chi}, ta_{\chi} >$$

Differing from a traditional CDEVS atomic component, a state $s_\chi \in S_\chi$ contains the information about the structure of the DSDEVS network. To represent the structure, the state notably contains information about the components, the connections and other variables.

$$s_\chi =< D^\chi, \{M_i^\chi\}, \{I_i^\chi\}, \{Z_{i,j}^\chi\}, SELECT^\chi, V^\chi >$$

where

D^χ	is the set of components
M_i^χ	is the model of component i, $\forall\ i \in D^\chi$
I_i^χ	is the influence of i, $\forall\ i \in D^\chi \cup \{\chi, \Delta\}$
$Z_{i,j}^\chi$	is the i-to-j output to input function, $\forall\ j \in I_i^\chi$
$SELECT^\chi$	is the *Select* function
V^χ	contains other variables required by the executive for decision making

Presenting the system that way leads to two possible approaches. Either the entire set of possible model structures are known initially, or they exist unpredictable architectures. In the first case, all structures can be encoded into the S_χ set of states, which is suitable when the number of architectures is reduced. However, with this formalism, treating cases in which the reachable architectural states are not predetermined is difficult. Indeed, Barros himself details in [Barros 1997b] the helper functions exposed by the simulator to add and remove components and links.

In order to understand the DSDEVS syntax, we present an example proposed by Zeigler [Zeigler 2000].

the figure 1.2 presents a system which has the ability to change its structure, by changing its coupled component into a 1-server or 2-server structure. If the server1 in this node is idle, it can be removed and transferred to another node. However, if a 3-server structure is not anticipated and integrated into s_χ, the node cannot have a 3-server structure. In a complex use case where the combinatorial structure cannot be planned, DSDEVS is not suitable.

According to the definition, the set of components is defined, but their state is not. Barros thus defines the new components states after a χ transition to be equal to the same components state before transition (plus time advance) if the component existed or to be the initial state if the component didn't exist.

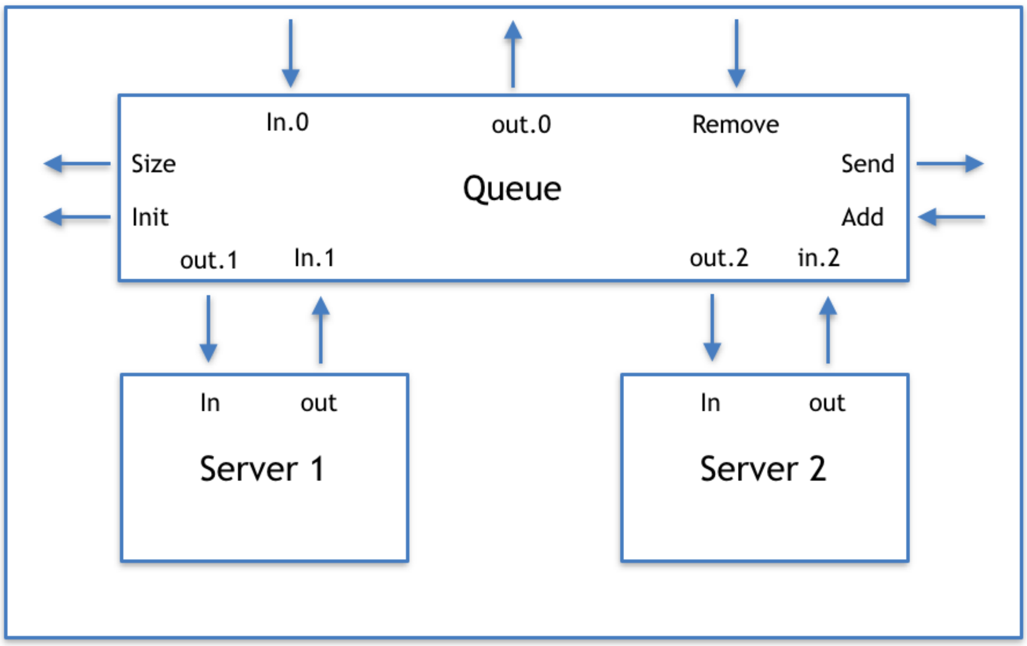


Figure 1.2: A DSDEVS exemple: block diagram of a node

1.2.2.2 Parallel Dynamic Structure DEVS

Parallel Dynamic Structure DEVS (DSDE) [Barros 1998] defines a specific component, χ, as for DSDEVS, whose state encodes the structure of the network. The current network structure can be obtained at any time from χ state using the structure function γ.

DSDE atomic components syntax defined as PDEVS atomic components

DSDE coupled components syntax The DSDE component is defined as for DSDEVS:

$$DSDE_N =< X_N, Y_N, \chi, M_\chi >$$

However, the model of the executive χ is an extended definition of an atomic model defined as:

$$M_\chi =< X_\chi, S_\chi, s_{0,\chi}, Y_\chi, \gamma, \Sigma^*, \delta_\chi, \lambda_\chi, \tau_\chi > \text{ where}$$

$\gamma \,: S_\chi \rightarrow \Sigma^*$ is the structure function

Σ^* is the set of network structures

In this definition, χ is the only component allowed to change the network structure. Moreover, the connections between the components of the network are also defined by χ state, i.e. a simple change of connector without affecting the atomic components themselves must be treated as a χ transition.

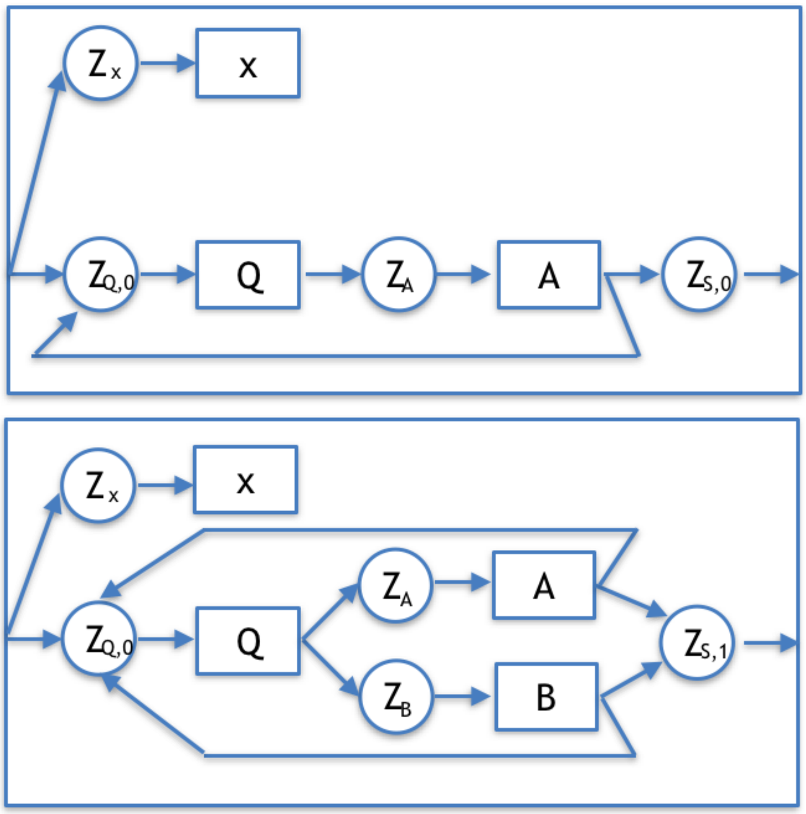


Figure 1.3: A DSDE exemple: one server network change to two sever network

An exemple of a buffered server [Barros 1998] is presented in figure 1.3. It is composed of the buffer Q and the single server A. When the waiting queue is long, it can hire a new server B.

The buffered server is defined by :

$$S =< X_S, Y_S, \chi, M_\chi > \text{ where}$$

$Y_S = \ J^*$

J = $\{c_0$, c_1,...,c_j,...$\}$ is a set of clients

J^* is a sequence of jobs

$X_S = \ J^* \times$ $\{$**change**, null$\}$ - $\{$(<>,null)$\}$

$M_\chi = = (X_\chi,\, s_{0,\chi},\, S_\chi,\, \delta_\chi,\, \tau_\chi)$

with

$X_\chi = \{\textbf{change}\}$

$S_\chi = \{s_{0,\chi},\, s_{1,\chi}\}$

$\tau_{s_{0,\chi}} = \tau_{s_{1,\chi}} = \infty$

$\delta_\chi(s_{0,\chi},\, e,\, \textbf{change}) = s_{1,\chi}$

$\delta_\chi(s_{1,\chi},\, e,\, \textbf{change}) = s_{0,\chi}$

$\gamma(s_{0,\chi}) = (D_0, \{M_{i,0}\}, \{I_{i,0}\}, \{Z_{i,0}\})$

$\gamma(s_{1,\chi}) = (D_1, \{M_{i,1}\}, \{I_{i,1}\}, \{Z_{i,1}\})$

where

$D_0 = \{Q, A\}$

$D_1 = \{Q, A, B\}$

$M_{Q,0} = M_{Q,1} = (X_Q,\, s_{0,Q},\, S_Q,\, Y_Q,\, \delta_Q,\, \lambda_Q,\, \tau_Q)$

$M_{A,0} = M_{A,1} = (X_A,\, s_{0,A},\, S_A,\, Y_A,\, \delta_A,\, \lambda_A,\, \tau_A)$

$M_{B,1} = (X_B,\, s_{0,B},\, S_B,\, Y_B,\, \delta_B,\, \lambda_B,\, \tau_B)$

$I_{A,0} = I_{A,1} = I_A = \{Q\}$

$I_{\chi,0} = I_{\chi,1} = I_\chi = \{S\}$

$I_{Q,0} = \{S,\, A\}$

$I_{S,0} = \{A\}$

$Z_{Q,0}\colon\; X_S \times X_A \rightarrow X_Q$

$Z_{A,0} = Z_{A,1} = Z_A$ and $Z_A\colon\; X_Q \rightarrow X_A$

$Z_{\chi,0} = Z_{\chi,1} = Z_\chi$ and $Z_\chi\colon\; X_S \rightarrow X_\chi$

$Z_{S,0}\colon\; Y_A \rightarrow Y_S$

$I_{B,1} = \{Q\}$, $I_{S,1} = \{A, B\}$, $I_{Q,1} = \{S, A, B\}$

$Z_{Q,1}\colon\; X_S \times X_A \times X_B \rightarrow X_Q$

Z_B: $X_Q \to X_B$

$Z_{S,1}$: $Y_A \times Y_B \to Y_S$

The network initial structure is presented at the top of figure 1.3 and when the executive receives an order to hire a new server it changes to the state represented in the botton of figure 1.3. The network can return to its initial structure after firing server B.

1.2.2.3 Conclusion on DSDEVS and DSDE

DSDEVS and its parallel version DSDE propose the first approach to dynamic structure systems representation. They describe a model based on different architectural states between which the components can switch.

However, the formalism given to represent these architectural states is either too general or too specific. It can list all the available architectural states by representing them as sets of sets, or provide a specific, limited list of available states. In the second case, the formalism can be adapted to represent a system in which a well-known set of states is reachable. But in the first case, the formalism lacks an expressiveness and relies on the simulation architecture for providing helper functions such as for adding or removing a component.

Moreover, DSDEVS way of taking into account component internal state change is limited. DSDEVS states that, when a new component appears in the simulation, it is automatically in its initial state, while all other components remain in their previous state. However, no traceability is provided to identify which components are new or not.

1.2.3 Dynamic DEVS and its dynamic port extension

The principle of Dynamic DEVS (dynDEVS) as described in [Uhrmacher 2001] is that each atomic component has its own model transitions function ρ_α which controls its own structural transformation. At coupled component level, the equivalent function is the network model transition function ρ_N. The model transition functions can pass from one structure to another. However, it does not consider the context management. Uhrmacher later developed $\rho-$DEVS, a dynDEVS variation supporting dynamic ports.

Under dynamic DEVS, the original definition of DEVS is changed to capture reflectivity by a recursive definition of models. Its transitions may possibly give rise to “new” models. A model changes its structure, i.e., its state space and its behavior pattern.

1.2.3.1 Dynamic DEVS (dynDEVS)

dynDEVS atomic components syntax A dynamic DEVS is a structure:

$$dynDEVS =_{df} < X, Y, m_{init}, \mathcal{M}(m_{init}) > \text{ where}$$

$$X, Y \text{ structured sets of inputs and outputs}$$

$$m_{init} \in \mathcal{M}(m_{init}) \text{ the initial model}$$

$$\mathcal{M}(m_{init}) \text{ is the least set having the structure}$$

$$\{< S, s_{init}, \delta_{ext}, \delta_{int}, \rho_\alpha, \lambda, ta > |$$

S the set of states $s_{init} \in S$ the initial state δ_{ext}:$Q \times X \to S$ the external transition functions with Q=$\{(s, e) : s \in S, 0 \leq e \leq ta(s)\}$ δ_{int}:$S \to S$ the internal transition function $\rho_\alpha : S \to \mathcal{M}(m_{init}$ the model transition function $\lambda : S \to Y$ the output function $ta : S \to \mathcal{R}_0^+ \cup \{\infty\}$ the time advance function }

and satisfying the property
$\forall n \in \mathcal{M}(m_{init}).\exists m \in \mathcal{M}(m_{init}).n = \rho_\alpha(s^m)\text{with}s^m \in S^m) \vee n = m_{init}$

dynDEVS coupled components syntax Networks (alias coupled models) do not add functionality to atomic models, since each network can be expressed as an atomic model (due to the DEVS closure under coupling property).

A dynamic network, dynNDEVS, is the structure:

$$dynNDEVS =_{df} < X, Y, n_{init}, \mathcal{N}(n_{init}) > \text{ where}$$

X, Y structured sets of inputs and outputs

$n_{init} \in \mathcal{N}(n_{init})$ the start configuration

$\mathcal{N}(n_{init})$ the least set having the structure$< D, \rho_\mathcal{N}, \{dynDEVS_i\}, \{I_i\}, \{Z_{i,j}\}, Select >$

D the set of component names

$\rho_\mathcal{N}$:$\mathcal{S} \to \mathcal{N}(n_{init})$ the network transition function with $\mathcal{S} = \times_{d \in D} \bigoplus_{m \in dynDEVS_d} S^m$

1.2.3.2 ρ-DEVS

This extension [Uhrmacher 2006] introduces dynamic ports and allows for multiple connections. It authorizes the models to enable or disable certain interactions at the same time.

ρ-DEVS atomic components syntax An atomic ρ-DEVS model is the structure:

$$\rho - DEVS =< m_{init}, \mathcal{M}, X_{SC}, Y_{SC}, m_{init} >$$

with $m_{init} \in \mathcal{M}$ the initial model, X_{SC}, Y_{SC} ports to communicate structural changes, $\mathcal{M}$ the least set with the following structure:
$< X, Y, S, s_0, \delta_{int}, \delta_{ext}, \delta_{con}, \rho_\alpha, \lambda_\rho, \lambda, ta >$

λ_ρ : $S \times X_{SC} \rightarrow \mathcal{M}$: model transition

ρ-DEVS coupled components syntax A reflective, higher order network, or coupled components is defined as:

$$\rho - DEVSN =< n_{init}, \mathcal{N}, X_{SC}, Y_{SC}, m_{init} >$$

$\mathcal{N}$ the least set with the following structure $< X, Y, C, MC, \rho_{\mathcal{N}}, \rho_\alpha >$ where

C set of components wich are of type ρ-DEVS

MC set of multi-couplings

ρ_λ : $S^n \rightarrow Y_{SC}$: structural output function

A multi-coupling $mc \in MC$ is defined as a tuple

$$mc =< \{(C_{src}.port)|C_{src} \in C\}, \{(C_{tar}.port|C_{tar} \in C\}, select >$$

1.2.3.3 Conclusion on DynDEVS and ρ-DEVS

DynDEVS and its extension deal with the atomic components reconfiguration which was not addressed in DSDE. It adds the atomics the capability to change their inner behavior with no changes on the coupled level. However, despite this interesting additional capability, DynDEVS still lacks details on how the state of the component is handled when facing a reconfiguration.

The ρ-DEVS extension deals with the external interface of atomics when an internal structure change occurs. This aspect is very interesting with regard to the dynamic behavior of components.

1.2.4 Synthesizable DEVS

Synthesizable DEVS (SynDEVS) is proposed by Molter [Molter 2012]. It is a hardware/software co-design flow based on DEVS model of computation. It proposes a co-simulation process with a DEVS-based formalism and distinguish hardware and software partitions during the simulation. It creates a VHDL-based hardware DEVS and a C++-based software DEVS. It considers the development process for a model of computation relying on simulation. However, it focuses neither on hardware nor on software. The definition of the hardware platform relies on a direct DEVS state-machine to Finite State Machine implementation. It considers the actual hardware clock to match the simulation clock.

Moreover, the model structure of SynDEVS is not dynamic. We have been inspired by its design flow approach, but not its formalism.

SynDEVS atomic components syntax SynDEVS atomic component is denoted by a 15-tuple

$SynDEVS_{atomic} =< P_{in}, P_{out}, X, Y, V, W, v_{init}, v, S, s_0, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, \tau >$

X	$Y, S, s_0, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, \tau$ defined as Zeigler's
V	variables
W	values for each variable
v_{init}	initial values for each values
v	variable assignment functions

SynDEVS coupled components syntax A coupled components, called as parallel components by Gregor Molter:

$SynDEVS_{parallel} =< P_{in}, P_{out}, X, Y, M, C_{in}, C_{out}, C_{inner} >$

P_{in}	input ports
P_{out}	output ports
X	event values for each input port
Y	event values for each output port
M	inner atomic or parallel components
C_{in}	connections between input ports and inner components
C_{out}	connections between inner components and output ports
C_{inner}	connections in-between inner components

1.2.4.1 Conclusion on SynDEVS

SynDEVS deals with hardware/software codesign. It proposes an interesting view of the codesign problem using the DEVS formalism. However, it tries to make the hardware architecture stick very closely to the DEVS state-machine and time representation. In SynDEVS, the hardware clock *is* the simulation clock. If this aspect brings interesting real-time applications, it restains the applicability and brings lots of constraints which can be avoided when working in, discrete, simulated time.

1.2.5 Reconfigurable DEVS

Reconfigurable DEVS (RecDEVS) [Madlener 2013] considers dynamic hardware MoC together with DEVS. RecDEVS proposes a model based on DEVS for final use on reconfigurable hardware like FPGA. In RecDEVS the system executive C_χ is in charge of the structure changes. However RecDEVS takes into account some hardware specificities from the beginning, like component communication relying on a bus structure with an address notion. This limits the model to an use on the target platform defined by RecDEVS. Thus, there is no separation between PIM and PDM. There are other limits on the meta-model itself, like the fact that a component deletion can only be triggered by the component itself. Eventually, there is no final implementation on FPGA, the workflow only goes to SystemC simulation.

$RecDEVS$ accounts for the various special properties of reconfigurable hardware architecture, like the bus architecture. Its concept is based on represeting the reconfigurable hardware blocks as DEVS components.

RecDEVS defines an unique identifier ID for each component.

RecDEVS atomic components syntax The difference between a RecDEVS atomic component and a PDEVS component is that the ports and messages are redefined.

$$C^{RecDEVS} =< X^{\#}, Y^{\#}, S, s_0, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, \tau >$$

$$\begin{aligned} S &\;, s_0, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, \tau \text{ as PDEVS} \\ X^{\#} &: I \times I \times Data \\ Y^{\#} &: I \times I \times Data \end{aligned}$$

$I = \{C_d^{ID} \mid d \in D, ID \in N\}$, a communication between two components is performed by sending a message onto a blobal communication system. Each message consists of a tuple $(sender, receiver, data)$, where $sender, receiver \in I$. The receiver can identify relevant messages by their address.

RecDEVS coupled components syntax

$$N_{Rec} =< X_{ext}, Y_{ext}, D, C_\chi > \text{ where}$$

D : Set of all available DEVS components

C_χ : is the network executive which is a DEVS atomic component

RecDEVS dynamic semantics The creation of new RecDEVS components consists of a fixed sequence of messages as follows:

∗ if the component C^{ID}_{orig} wants to create a new component of type $d \in D$, it sends a message (C^{ID}_{orig},C_χ,(new d)) to the network executive.

∗ C_χ receives the message and performs an external transition δ_{ext}. This will create a new RecDEVS component C^{id}_d and add it to the list of instantiated components

∗ A confirmation message (C_χ, C^{ID}_{orig}, (confirm C^{id}_d)) with the address of the new component is then sent to the originator.

∗ Starting from the reception of the confirmation message, the originator can address the newly created component.

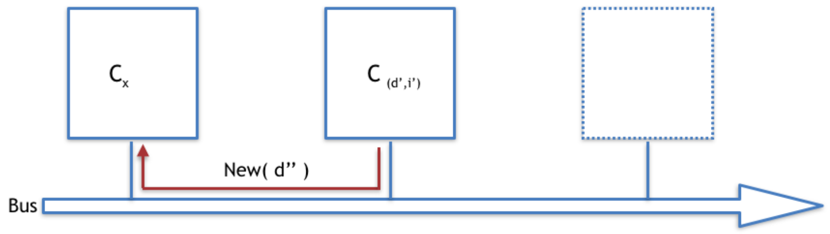


Figure 1.4: A RecDEVS exemple: add a new component

1.2.5.1 Conclusion on RecDEVS

RecDEVS proposes the first simulator aiming at supporting dynamic structure models on partially reconfigurable architectures. Moreover, many interesting notions, such as the identifier adjoined to the component in order to uniquely identify it.

Nevertheless, RecDEVS models have to include low-level, platform-related, description elements. This ties the models deeply with the execution platform, preventing them from being adapted to other simulators. This goes against the MDA approach in which the models are platform-independent.

Furthermore, the component context aspect is not treated. When dealing with dynamic component, the context is of prominent importance. For example, we may want to suspend a component at one time and resume it later from the same point. If the component has to be removed meanwhile, this can only be done by supporting context saving and reloading.

1.3 Dynamically reconfigurable computing systems

Under earlier Von Neumann architecture by 1950 [Iannucci 1988], as shown in figure 1.5, an electronic digital computer consists in a processing unit, a control unit, a memory, external mass storage and input and output mechanisms. The processing unit is made of an arithmetic and logic unit and processor registers. The control unit contains instruction registers and program counter. The memory stores both data and instructions.

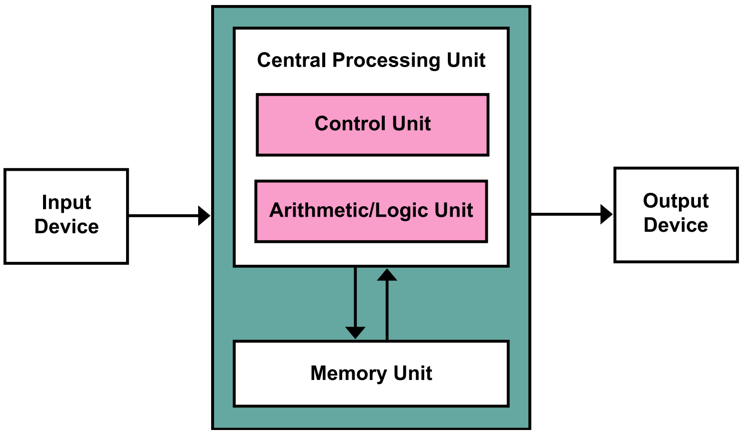


Figure 1.5: Von Neumann Architecture
(Image “Von Neumann Architecture”
by Kapooht, available on Wikimedia Commons, CC-BY-SA-3.0)

After 60 years of development, the ability of computing using a processing unit arrives at a high level: a system on chip [Xu 2005] has a maximum CPU clock rate up to 3 GHz [Deleganes 2002], compared to the first Intel 4004 at 740 kHz [Aspray 1997]. However, CPU’s are sequential processing devices, even if multi-cores CPUs are developed nowadays, allowing for multiple simultaneous processings. Compared to a CPU, the parallel nature [Adamski 2005] of Graphics Processing Units (GPU) are quite an interesting for multi-process application.

For high-performance, dedicated applications, it is possible to design Application-Specific Integrated Circuits (ASIC), containing optimized designs for a single application. However, in traditional integrated circuits fabrication, the design is long and expensive, and no modification is possible unless producing a new chip.

Between CPU and ASIC, there are flexible hardware materials, which can implement specific functionalities which can be modified.

In this section, we are presenting the dynamically reconfigurable computing systems: the programmable logic devices, especially the architecture of the FPGA, and the technologies of partial reconfiguration.

1.3.1 Programmable logic device

A programmable logic circuit (PLD), is an integrated circuit that can be configured after manufacture. In order to distinguish from software, we try to avoid using the word “programming”. Instead, we call it “configuring” since the hardware level changing is not due to executing any line of code. We modify connections or the behavior of the physical component: we connect logic gates between them. The word “programming” is still used under several circumstances in the hardware, but in the case where the configuration is done and the modifiable logical networks considered fixed.

The most common reconfigurable hardware is the FPGA, which makes it possible to build any logic circuit within the limits of available resources. Some offer partial dynamic reconfiguration [Lysaght 2006], which allows a modification of a part of the component without interfering with the operation of elements that are not modified.

This component will obviously be at the center of the execution platform. Although there are other aspects of PLD as Application-Specific Integrated Circuits (ASICs) or Application-Specific Standard Parts (ASSPs), in comparision to an FPGA, these devices contain a relatively limited number of logic gates, and the functions they can be used to implement are much smaller and simpler [Maxfield 2004].

FPGA is a separate stream of development of original programmable logic array which is based on gate array technology. FPGAs based on user-defined place (gate) and route (multiplexers) and register-rich (D flip-flops) architectures.

Hardware reconfiguration is based on a circuit with generic hardware resources which can be connected by a programmable network as shown in figure 1.6.

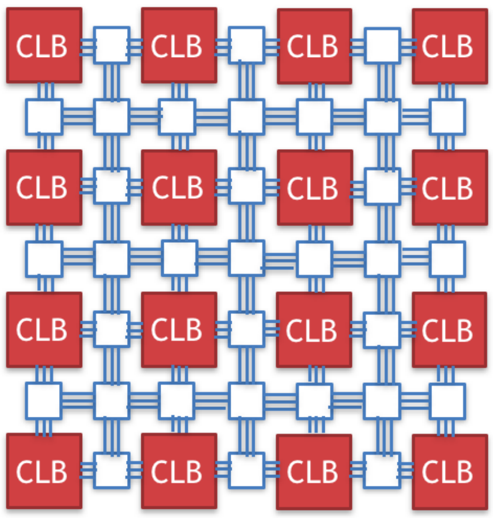


Figure 1.6: FPGA architecture

Configurable Logic Block (CLB) is usually made up of several cells. Within each cell, there are LookUp Tables (LUTs) associated with multiplexers and flip-flops. A LUT is a memory associating an output with an input, in which it is possible to record the result of a logic function, associating the desired value with each input combination. LUTs are associated with flip-flops and multiplexers to provide different configurations of the cell. The LUT is a truth table which can mimic gates (like AND, NOR, OR, etc) linked to form a final output.

The five main advantages of FPGA technology according to [Hauck 2010] are: performances, time to market, cost, reliability, long-term maintenance. That makes FPGAs used almost everywhere today [Maxfield 2004] including communication devices and software-defined radios; radar, image, and other digital signal processing (DSP) application; system-on-chip (SoC) component that contain both hardware and software elements.

1.3.2 Partial reconfiguration

Partial Reconfiguration [Platzner 2010] is the ability to dynamically modify blocks of logic while the remaining logic continues to operate without interruption.

Partial Reconfiguration (PR) takes FPGA flexibility one step further, allowing the modification of an operating FPGA design by loading a partial configuration file, sometimes referred to as a bitstream file. After a full bitstream file configures the FPGA, partial bitstream files can be downloaded to modify reconfigurable regions in the FPGA without compromising the integrity of the applications running on those parts of the device that are not being reconfigured.

In order to isolate the partial reconfiguration area from the static areas during the reconfiguration process, several tools are needed to ensure the calculation.

The use of Partial Reconfiguration can allow designers to move to fewer or smaller devices, reduce power, and improve system upgradability. Make more efficient use of the silicon by only loading in functionality that is needed at any point in time.

We will use here Xilinx tools as a base platform. There are tools integreated in Xilinx VIVADO platform which help to manage a partial reconfiguration design [Xilinx 2018], notably a Partial Reconfiguration Controller and a Partial Reconfiguration Decoupler, shown in figure 1.7. The partial reconfiguration controller is a low level hardware-based configuration controller. The bitstreams are delivered and arranged by the partial reconfiguration controller. The partial reconfiguration decoupler isolates the dynamic region from the static one while it is being reconfigured.

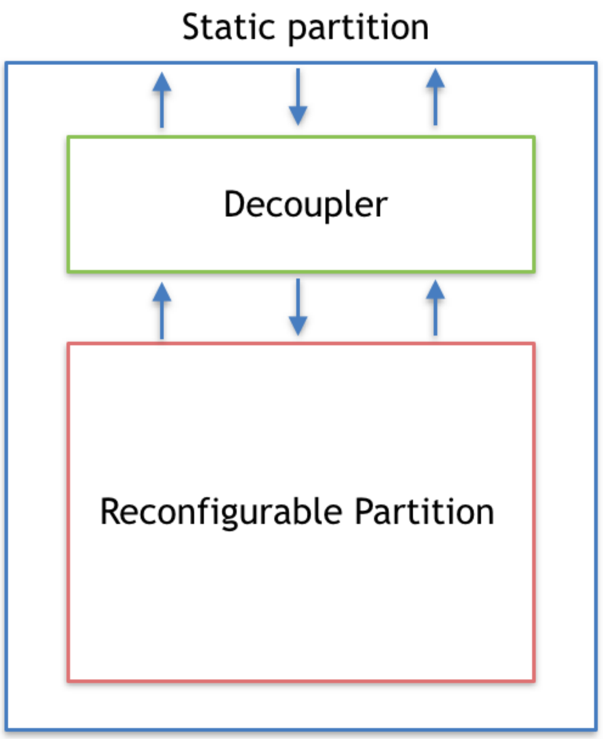


Figure 1.7: Virtual socket containing a decoupler

The Partial Reconfiguration Controller uses a notion of Virtual Sockets to identify a reconfigurable area along with its decoupling and reset logic. A virtual socket thus describes all the information required to do the reconfiguration and initialize the component in a safe state.

The controller contains virtual socket managers, each is in charge of controling one virtual socket. One socket manager contains a list of up to 128 partial bitstreams that can be used to configure the reconfigurable area of the socket. Each manager has an interface to the socket to control its isolation and reset logic. The manager also has information to know what are the operations to apply on the socket when a reconfiguration is triggered. E.g., when a reconfiguration is launched, the socket manager will first shut down the existing component within the socket, isolate it, apply the reconfiguration, reset the component then finally connect it.

1.4 Conclusion

This chapter has explained the basic knowledge in order to understand the problems in this thesis. Through the history of system engineering development, we could see the evolution of the system process of modeling. Involving modeling and simulating through all steps, system engineering makes it possible to build systems more and more complex.

We have seen that there already exist several meta-models based on discrete event system for simulation, and even extensions for dynamic systems. However, most of them do not apply to the approach of system engineering. These meta-models are not adapted to the partially reconfigurable systems.

Moreover, our point of view on existing formalisms which allow describing dynamic structure systems is that they are either too high level, making it difficult to apply to real systems, or too deeply linked to the execution platform. The formalisms mentioned in this section stay at theoretical level: they propose an abstract simulator and can be adapted to an actual execution environment.

The main DEVS-based formalisms structure-change aware are DSDEVS, DynDEVS and their respective extensions. They all provide a mathematic syntax based on architectural states which evolve during the simulation time. We identified a few lacks in these meta-models.

The first one is related to the syntax description of the structural states. Indeed, for systems whose all architectural states cannot be predicted, the formalism is too generic and requires helper functions to be provided by the simulator. Doing so reduces the replicability of the simulations by bringing the execution context in the semantics.

The context of the component is not considered neither. At some point in a dynamic structure simulation, it can be of use to force a change of a component state externally. For example, one may want to reload a specific context previously saved.

We also investigated at hardware-related DEVS extensions. SynDEVS integrated the idea of model transformation flow. It aims at form a software / hardware co-simulation but remains static. Finally, RecDEVS integrates the dynamic model notion on hardware, but the model is tightly bound to the low-level implementation platform and the context management is not considered.

Nowadays the development of dynamically reconfigurable computing systems requires a system engineering approach for modeling, which is the one that will be considered in this thesis.

Chapter 2
Partially Reconfigurable Discrete Event System Specification

Contents

In the previously studied formalisms, we identified some lacks in the capabilities offered.

The first one is that most formalisms rely on a pre-established list of structures that a model can adopt during its life. As an example, DSDE encodes the structure in the χ component state, and the γ function translates it to an actual model structure. This way of doing is not adapted to the current methodology adopted in auto-adaptive design, in which the live state may be not predictable at the design phase.

A second remark applies to RecDEVS, which is the only dynamic DEVS formalism to target dynamic hardware. In this case, the system engineering approach is not applied, in which the high-level model should be disconnected from the target, low-level, executing platform. RecDEVS integrates platform constraints directly in the high-level model. In order to consider those specifications, we propose a new meta-model which adopts the MDA approach by separating the abstract model from the executing platform. This meta-model is based on the foundation of PDEVS which is extended to structural changes. We name it Partially Reconfigurable Discrete Event System Specification (PRDEVS).

2.1 PRDEVS meta-model syntax

The meta-modeling approach is frequently done using UML as a definition tool. In our approach however, the PRDEVS meta-model will be based on mathematical definition as for the original DEVS formalism.

We propose a PIM syntax based on PDEVS and inspired by RecDEVS. The platform-independent model is, as stated from the name, a model which is built to represent an application, without necessarily knowing the simulation environment or the target platform that will run the application or the application simulation. Thus, our PIM syntax must be able to represent dynamic structure models, but shouldn't assume anything about how the structure changes will actually be applied. This is the main difference with RecDEVS, as the RecDEVS meta-model only considers PSM. The target architecture is assumed from the model definition, notably by using the address notion within the high-level models.

2.1.1 PRDEVS abstract syntax

A PRDEVS is a model which contains all required information about components, structure, and allows for structural changes. Though we use PDEVS and RecDEVS as a base reference, we slightly rewrite some definitions to clarify specific points. A major difference with RecDEVS is that we do not use a specific component like C_χ which stores the network structure in its state: we directly manipulate the sets, adding and removing elements. The first motivation for this approach is to be closer to the current engineering approach to describe dynamic systems. This is slightly equivalent to the software notion of new/delete instructions for object manipulation. Moreover, this way of doing offers the ability to reach structure states which may not have been predicted when first conceiving the system, allowing for auto-adapting systems to be more flexible.

Concerning the D set, on one hand, PDEVS defines D to be *the set of components names*, i.e. a list of all the names of the components *inside* the coupled. On the other hand, RecDEVS definition of set D is "a list of available component names", and this set is compared to a list of components *types*. They both define the set $\{M_d \mid d \in D\}$, which contains the components themselves. In our component sets-based description, we rather directly manipulate the components sets and think this description is redundant, as one can be obtained from the other. So we chose to merge these two sets so that the D set directly contains the components themselves. Instead of storing the names of the components, we rather use the notions of *identifiers* and *types*.

The *identifier* follows the notion introduced by RecDEVS where an identifier $ID \in \mathbb{N}$ is attributed to each component. For the *type* notion, we can make the connection with object-oriented programming, where there can be various instances of a *class*, we call *objects*. Here, the notion of *type* is equivalent to *class*, i.e. it defines a component structure and initial state, but there can be various components sharing the same type with a different state. The identifier is then used to differentiate the components. We use here a definition close to RecDEVS, but formalize the notation: we use T as the list of defined types, i.e. a list of components types which can be instantiated.

A PRDEVS component then has an identifier, which is unique and dynamically defined, and a type, which can be shared.

2.1.2 Root PRDEVS component

The dynamic structure ability relies on a library of available components, each being of a specific type, which can be added to the system. The library is defined as $L = \{C_t \mid t \in T\}$. Components in the library expose a null identifier, as it is defined on instantiation.

Components in use still have a type $t \in T$, but also an identifier id, and are noted as C_t^{id}. The notation can be simplified to C^{id}. Unlike RecDEVS, we do not restrict $id \in \mathbb{N}$: although a PSM implementation will probably have to impose such a restriction, the PIM doesn't require so. We thus define an arbitrary set ID which contains the allowed identifiers.

With this considerations, we then define a PRDEVS to be:

$$PRDEVS \; = < L, C^{Top} > \text{ with:}$$

$L \; = \{C_t \mid t \in T\}$, the library of available components

C^{Top} is a coupled component containing the application structure

C^{Top} has an initial configuration, which will evolve over simulation time.

To have a clear and simple definition, we did not include the sets T and ID in the $PRDEVS$ definition. Indeed, T can be retrieved from L using the reverse definition $T = \{t \mid C_t \in L\}$, while ID can be any arbitrary set.

2.1.3 Coupled components

A coupled component will be very alike PDEVS definition:

$$N^{nid} =< X, Y, D, EIC, EOC, IC >$$

All elements of this definition are the same as PDEVS definitions, except for D:

$D \; = \{C_t^{id} \mid t \in T, id \in ID\}$ the set of components contained in the coupled

A coupled component C^{nid} does not have a specific type, as the component structure can change during execution when a component is added to or removed from D^{nid}. From D, we can get the set of all identifiers of components in the coupled: $ID^{nid} = \{id \mid C^{id} \in D^{nid}\}$.

2.1.4 Atomic components

As our formalism deals with structure changes, the atomic components definition adds a structure change function to the PDEVS atomic formalism:

$$M_t^{mid} =< X, Y, S, s_0, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda_{SC}, \lambda, \tau >$$

$\lambda_{SC} \; : S \rightarrow SC$ the structure function with

$SC \; = \{addComponent(),\ removeComponent(),\ addPort(),\ removePort(),\ addConnection(),\ removeConnection()\}$ the list of structure change functions as detailed in section 2.2.

2.1.5 Ports

The ports of a component are implicitly defined by the couples (p, v) of X and Y sets. However, a specific definition can be derived to formally identify the ports as mathematical objects. This representation of ports as independent objects, i.e. not only as names belonging to a set, and whose allowed values are defined by the X and Y sets, is easier to manipulate.

The ports of the components must be uniquely identified, but only among a component. Thus, the name of the port on the component is sufficient to identify it uniquely, as long as the component itself has a unique identifier. The port name is then equivalent to the identifier notion. As for the component, the identifier can be part of any set, and thus can be a character string. Moreover, a port has a direction (input/output), and a set of available values. We then introduce a definition of what is a port:

$$P^{id} =< \mathit{Direction}, \mathit{DataType} > \text{ with}$$

id the identifier of the port

$\mathit{Direction} \in P_{dir} = \{in, out\}$

$\mathit{DataType} \in P_{type}$

The set of allowed types P_{type} can be defined as a set of allowed definition sets, and will most likely contain $\mathbb{N}$, $\mathbb{R}$, $\mathbb{B} = \{True, False\}$, etc.

Concerning the connection between two ports, it must define a source and a sink. The data type does not have to be recorded as part of the connection object, but type match between source and sink should be checked when the connection is created.

On a structure change point of view, the ports can be added and removed from a coupled component. An atomic component can not see its ports sets changed, because it would affect its behavior. This was addressed by ρ-DEVS, but we chose not to integrate it. Thus, an atomic component with different sets of ports is a different type of component.

2.1.6 Convenience sets and notations

All of the sets defined previously can be linked to a specific component by displaying its identifier, e.g. X^{nid} refers to the set of inputs X of coupled component C^{nid}. The same way, S^{mid} is the set of states S of component C^{mid}.

For convenience, we also define a few additional sets to gather elements regardless of the hierarchy.

Sets of components:

D_N the set of all coupled in the PRDEVS

D_M the set of all atomics in the PRDEVS

$D_{PRDEVS} = D_M \cup D_N$

Sets of identifiers:

$ID_N = \{id \mid C^{id} \in D_N\}$, the set of all coupled identifiers

$ID_M = \{id \mid C^{id} \in D_M\}$, the set of all atomics identifiers

$ID_{PRDEVS} = ID_N \cup ID_M$

$ID_P = \{id \mid P^{id} \in D_{PRDEVS}\}$, the set of ports identifiers

2.2 PRDEVS semantics

We rely on PDEVS semantics for most of the meta-model. The only addition is the structure change semantics. Structure change functions have a different priority level than other messages in the simulator: the simulator has a list of pending SC tasks and the list is executed at the end of an imminence cycle.

Unlike RecDEVS, we do not restrict which component can call a structural change function. Indeed, RecDEVS states that a component can only delete itself, not another component. The main justification provided is that it avoids accidentally deleting a component which is still in use. By only allowing self-deletion, the component can announce its own deletion to linked components before committing deletion. But this approach doesn't seem to be a good answer to this issue. Indeed, deleting a component which is still in use in an application may be a conception error. Restricting the remove call to the component itself does not solve the case where the component itself is badly defined, and forget announcing its deletion to some of the related components. Imposing such a constraint do not avoid errors, so the restriction is irrelevant. We believe the application correctness is up to the modeler, and to avoid such errors, applications should be checked for correctness, e.g. using formal methods.

By allowing any component to call structure change functions, we let the modeler decide how to handle its structure changes: all components can be autonomous and directly trigger the functions, or there can be one or more components in the model which are in charge of the structure, only them being able to call these functions.

In our definition, we will assume as a first approach that model correctness is done statically before the simulation is run. We thus do not include error-checking in the semantics.

We define 6 structure change functions, and a set of general functions that are used by the structure change functions.

2.2.1 General functions

The following functions are defined as reference functions in the semantic to mutualize common operations.

2.2.1.1 getAvailableId

getAvailableId : $\emptyset \to ID$

This function determines an available identifier from ID. For example, if $ID = \mathbb{N}$, a variable will determine the next unused identifier, which will be returned, and the variable will be incremented. The function actual implementation is up to the PDM, which must defines its own ID set.

2.2.1.2 getNewComponent

getNewComponent : $T \to L$

Returns a new component from the library matching the given type. It is the function that allows to retreive the component object from the component type. For $t \in T$, it will return $\{C_t \mid C_t \in L\}$.

2.2.1.3 getExistingComponent

getExistingComponent : $ID_{PRDEVS} \to D_{PRDEVS}$

This function returns an existing component from its identifier.

2.2.1.4 getParentComponent

getParentComponent : $D_{PRDEVS} \to D_N$

Returns the parent component of a component. It can be applied to an atomic or a coupled, but always return a coupled. It cannot be applied on C^{Top}.

2.2.1.5 getPort

getPort : $D_N \times ID_P \to Port$

Gets an existing port from a coupled component.

2.2.2 Structure change semantics

2.2.2.1 addComponent

addComponent : $T \times ID_N \rightarrow ID$

Adds a new component into an existing coupled component. The new component type and the hosting coupled ID are provided as parameters. The identifier of the new component is returned. The algorithm 1 depicts the function behavior.

```
input  : $t \in T$; $id_{host} \in ID_N$
output: $newId \in ID$
Data: $newC \in L; hostC \in D_N$
newId ← getAvailableId()
newC ← getNewComponent($t$)
newC.id ← newId
hostC ← getExistingComponent($id_{host}$)
hostC.D ← hostC.D ∪ {newC}
return newId
```

Algorithm 1: addComponent procedure

2.2.2.2 removeComponent

removeComponent : $ID_{PRDEVS} \rightarrow \emptyset$ Removes an existing component, whose identifier is passed as a parameter, from the PRDEVS. Its behavior is presented on algorithm 2.

```
input: $id_{removed} \in ID_{PRDEVS}$
Data: $rem_C \in D_{PRDEVS}, hostC \in D_N$
remC ← getExistingComponent($id_{removed}$)
hostC ← getParentComponent($remC$)
hostC.D ← hostC.D \ {remC}
```

Algorithm 2: removeComponent procedure

2.2.2.3 addPort

addPort : $ID_N \times P_{type} \times P_{dir} \rightarrow ID$

Adds a port to a coupled component whose identifier is provided, with the associated *type* and *direction*. The new port identifier is returned, as displayed on algorithm 3.

```
**input** : $id \in ID_N$; $p_t \in P_{type}$; $p_d \in P_{dir}$
**output:** $pid \in ID$
**Data:** $newId \in ID$; $new_{port} \in Port$; $hostC \in D_N$
newId $\leftarrow$ getAvailableId()
$new_{port} \leftarrow (p_t, p_d)$
$new_{port}$.id $\leftarrow$ newId
hostC $\leftarrow$ getExistingComponent($id$)
hostC.port $\leftarrow$ hostC.port $\cup$ $\{new_{port}\}$
return newId
```

Algorithm 3: addPort procedure

2.2.2.4 removePort

removePort : $ID_N \times ID_P \rightarrow \emptyset$

Removes a port from the component whose ID is provided, as displayed on algorithm 4.

```
**input** : $id \in ID_N$;$p_n \in ID_P$
**Data:** $hostC \in D_N$; $removed_{port} \in Port$
hostC $\leftarrow$ getExistingComponent($id$)
$removed_{port} \leftarrow$ getPort($hostC$,$p_n$)
hostC.port $\leftarrow$ hostC.port $\setminus$ $\{removed_{port}\}$
```

Algorithm 4: removePort procedure

2.2.2.5 addConnection

addConnection : $ID_{PRDEVES} \times ID_P \times ID_{PRDEVS} \times ID_P \rightarrow \emptyset$

Adds a connection between two ports. The ports are referred to using the combination of the component identifier and the port identifier. The $Z_{i,d}$ definition from Zeigler must be respected, i.e. the two components must be in the same coupled, or one of the two component must be a coupled and the other one a component inside the coupled, and one must be an input and the other an output. Moreover, the definition interval *type* must match between the two ports. The definition is displayed on algorithm 5.

2.2.2.6 removeConnection

removeConnection : $ID_{PRDEVES} \times ID_P \times ID_{PRDEVS} \times ID_P \rightarrow \emptyset$

Removes a connection between two ports using the same notation as the previous function, as displayed on algorithm 6.

```
input : $id_1 \in ID_{PRDEVS}$; $p_n1 \in ID_P$; $id_2 \in ID_{PRDEVS}$; $p_n2 \in ID_P$
Data: $hostC1 \in D_N$; $hostC2 \in D_N$; $connection \in PortConnection$

$hostC1 \leftarrow$ getParentComponent($id_1$)
$hostC2 \leftarrow$ getParentComponent($id_2$)
$connection \leftarrow \{(id_1, p_n1), (id_2, p_n2)\}$
if $hostC1.id = hostC2.id$ then
    $hostC1.IC = hostC1.IC \cup \{connection\}$
else if $hostC1.id = id_2$ then
    $hostC1.EOC = hostC1.EOC \cup \{connection\}$
else if $hostC2.id = id_1$ then
    $hostC2.EIC = hostC2.EIC \cup \{connection\}$
```

Algorithm 5: addConnection procedure

```
input : $id_1 \in ID_{PRDEVS}$; $p_n1 \in ID_P$; $id_2 \in ID_{PRDEVS}$; $p_n2 \in ID_P$
Data: $hostC1 \in D_N$; $hostC1 \in D_N$; $connection \in PortConnection$

$hostC1 \leftarrow$ getParentComponent($id_1$)
$hostC2 \leftarrow$ getParentComponent($id_2$)
$connection \leftarrow \{(id_1, p_n1), (id_2, p_n2)\}$
if $hostC1.id = hostC2.id$ then
    $hostC1.IC = hostC1.IC \setminus \{connection\}$
else if $hostC1.id = id_2$ then
    $hostC1.EOC = hostC1.EOC \setminus \{connection\}$
else if $hostC2.id = id_1$ then
    $hostC2.EIC = hostC2.EIC \setminus \{connection\}$
```

Algorithm 6: removeConnection procedure

2.2.3 Components context under dynamic behavior

A model dynamicity resides not only in adding or removing component. As seen previously, the components interconnections are also part of the structure. Moreover, an additional level of change can be seen in the components internal behavior evolution, denoted by its current state. In DSDE, when a new component is added, its state is set to the initial state, as detailed in [Barros 1998].

We name a component total state (Q) its context. The context is made of the component current state S_i and the elapsed time e. In some cases, it could be interesting to be able to set a component into a different context.

The first use of this capability can be to save the context of a component before removing it. This allows to later restore it and resume the computations from the point where there were stopped. This if for example of use in a time-slicing, preemptive context, where the computation resources are limited and distributed over time to different tasks. Another application for restoring a previously saved context are a suspension of a task to be resumed later, or the ability to restore a backed-up context if needed.

The second use for context restoration can be to allow choosing the starting state of a component. For example, a component can be initialized in different states depending on its situation. In such a case, DSDE limitation demands several independent models which have exactly the same structure, except for the initial state.

In either case, this ability requires to be able to store the components contexts. For context saving and restoring, the context library is initially empty, and progressively filled over simulation time. In the different initialization scenario, the context library has to contain various initial contexts at simulation start. In both cases, it requires a context library, and functions to interact with the components contexts.

A context is an element which contains all the required data to fully determine a component state. When placed in a library, it must be uniquely labeled by an identifier.

For such purpose, we must extend the PRDEVS definition to include the context library, which we name L_Φ. We then define the context-aware PRDEVS as follows:

$$PRDEVS_\Phi \ = < L, L_\Phi, C^{Top} > \text{ with:}$$

$L_\Phi \ = \{Q_t^{id} \mid id \in ID_\Phi, t \in T\}$, ID_Φ being the allowed context identifier set.

We also need two functions to allow saving a context from a component and changing a component context.

2.2.3.1 getContext

getContext : $ID_{PRDEVS} \rightarrow ID_\Phi$

Obtains a context from a component and saves it to the context library, as displayed on algorithm 7.

2.2.3.2 setContext

setContext : $ID_{PRDEVS} \times ID_\Phi \rightarrow \emptyset$

Applies a context from the context library to an existing component, as detailed on algorithm 8.

2.3 Graphic representation

Although we have already the syntax and semantics for PRDEVS, the mathematical notations are sometimes not optimal for model readability. The main

```
input  : $id \in ID_{PRDEVS}$
output: $newId \in ID_{\Phi}$
Data: $C \in D_{PRDEVS}; Q \in \Phi$
newId ← getAvailableId()
C ← getExistingComponent($id$)
Q ← C.Q
Q.id ← newId
Q.t ← C.t
$L_{\Phi} \leftarrow L_{\Phi} \cup \{Q\}$
return newId
```

Algorithm 7: getContext procedure

```
input  : $id \in ID_{PRDEVS}; id_{\Phi} \in ID_{\Phi}$
Data: $C \in D_{PRDEVS}; Q \in \Phi$
C ← getExistingComponent($id$)
$\Phi \leftarrow \{Q^i \mid i = id_{\Phi}\}$
$C.Q \leftarrow \Phi$
```

Algorithm 8: setContext procedure

goal of this section is to ease the definiton of PRDEVS models while keeping the preciseness of the mathematical notation. Other popular Models are usually exploited with visual programming paradigm, after the mathematical definitions, as Ptolemy [Ptolemaeus 2014], UML StateCharts [Latella 1999].

2.3.1 Components

To represent components graphically, we chose the square for atomic components and a rounded square for the coupled, as shown in 2.1. The input and output ports are presented as a triangle, inputs on the left side and output on the right side. The example of figure 2.1 shows an atomic component with two input ports and a coupled component with one output port.

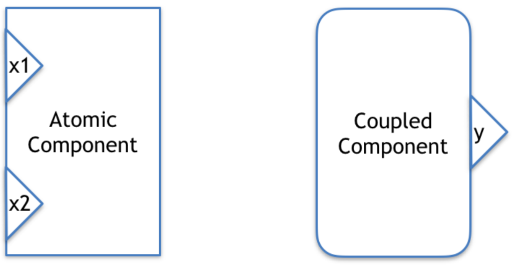


Figure 2.1: Graphic representation of components

2.3.2 States, transitions and actions

Inside the atomic components, there are states (S), transitions (δ) and actions (λ). As the total state can be uncontable, it can be cut into various internal variables. The main finite one will be called phase and represented graphically.

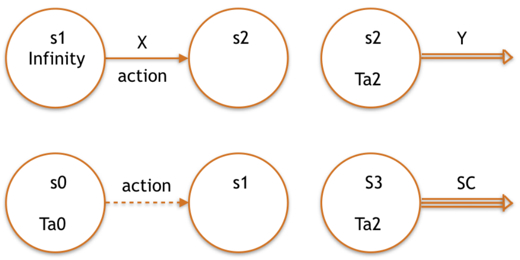


Figure 2.2: Graphic presentation of transitions

The figure 2.2 represents the atomic internal elements. We chose a circle to represent the phase, lines for the different transitions and arrows for the actions. A solid line represents an external transition, the input is noted with a question mark "?". A dotted line represents an internal transition, the condition of the transition being marked with brackets "[ ]". A double arrow represents an output emission, the output value is assigned to the output port. A triple arrow represents a structural change call, the dynamic function is marked around the triple line.

2.4 Example

In order to detail the structure of the PRDEVS meta-model, we give an example of a simple dynamic structure model.

Root model $ModelExample =< L, L_\Phi, C^{Top} >$ where

$$L = \{C_{counter}, C_{generator}, C_{empty\_coupled}\}$$

$$L_\Phi = \{Q^1_{generator}, Q^2_{generator}\}$$

$$C^{Top} = \{X, Y, D, EIC, EOC, IC\}$$

with

$$X^{Top} = Y^{Top} = EIC^{Top} = EOC^{Top} = IC^{Top} = \{\}$$

$$D^{Top} = \{C^{cpt}_{counter}\}$$

Atomic component - counter The counter contains the dynamic calls. It can be described as:

$C_{counter} =< X, Y, S, s_0, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda_{SC}, \lambda, \tau >$ with

$X = \{P^{EVENT}\}$, with $P^{EVENT} = (in, \mathbb{B})$

$Y = \{\}$

$S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7\} \times ID \times ID \times ID \times \mathbb{N}$
repectively: s, coupledID, generatorID, portID, count

$s_0 = (s_0, \varnothing, \varnothing, \varnothing, 0)$

$$\delta_{int}(s, count) = \begin{cases} s_1 & \text{if} \quad s = s_0 \\ s_2 & \text{if} \quad s = s_1 \\ s_3 & \text{if} \quad s = s_2 \\ s_4 & \text{if} \quad s = s_3 \\ s_5 & \text{if} \quad s = s_6 \ \wedge [count \neq 10]) \ or \ (s = s_4) \ or \ (s = s_7) \\ s_7 & \text{if} \quad s = s_6 \ \wedge [count = 10] \end{cases}$$

$$\delta_{ext}((s, count), in) = (s_6, count + 1) \text{ if } s = s_5 \wedge in \rightarrow \mathsf{TRUE}$$

$$\delta_{con}(s) = \{\}$$

$$\lambda_{SC}(s) = \begin{cases} coupledID = addComponent(empty\_coupled, top) & \text{if} \quad s = s_0 \\ generatorID = addComponent(generator, coupledID) & \text{if} \quad s = s_1 \\ portID = addPort(coupledID, boolean, out) & \text{if} \quad s = s_2 \\ addConnection(generatorID, out, coupledID, portID) & \text{if} \quad s = s_3 \\ addConnection(coupledID, portID, cpt, EVENT) & \text{if} \quad s = s_4 \\ setContext(generatorID, 2) & \text{if} \quad s = s_7 \end{cases}$$

$$\tau(s) = \begin{cases} \infty & \text{if} \quad s = s_5 \\ 0 & else \end{cases}$$

Atomic component - generator The generator is static which can be described as:

$C_{generator} =< X, Y, S, s_0, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda_{SC}, \lambda, \tau >$ with

$X = \{\}$

$Y = \{P^{EVENT}\}$, with $P^{EVENT} = (out, \mathbb{B})$

$S = \{s_0\} \times \mathbb{R}^+$
repectively s and speed

$s_0 = (s_0, 10)$

$$\delta_{int}(s_0) = \{s_0\}$$

$$\lambda(s_0) = (P_{EVENT} \leftarrow \mathsf{TRUE})$$

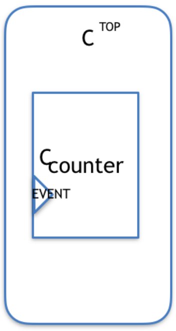
$$\tau(s_0) = speed$$

$$\delta_{ext}(s) = \delta_{con}(s) = \lambda_{SC}(s) = \{\}$$

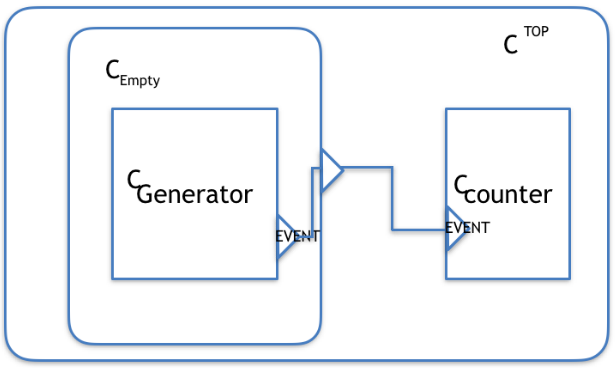
Context Library $L_\Phi = \{Q^1_{generator}, Q^2_{generator}\}$
$Q^1_{generator} = (s_0, 1)$
$Q^2_{generator} = (s_0, 2)$

Graphic representation - Top level Here is a graphic representation of this example. At C^{TOP} there is only counter at the beginning, as shown in 2.3a. At the end of the initial phase, a generator is added and connected to the counter, as detailed in 2.3b.

Initialy, the generator emits an event every 10 time units. After the counter reaches 10 for its variable count, the context Q^2 is loaded which changes the generator speed to 2 time units.



(a) Graphic presentation of the beginning



(b) Graphic presentation of the end

Figure 2.3: Graphic presentation of the top level component

Graphic representation - generator atomic In generator there is only one phase state, as shown in 2.4.

Graphic representation - counter atomic In counter there are eight phase states, as shown in 2.5

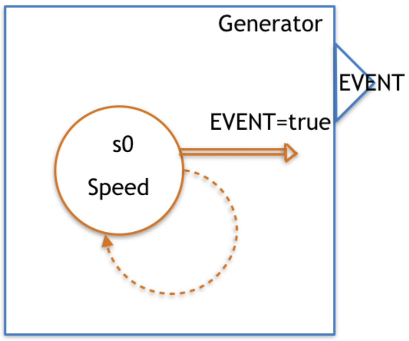


Figure 2.4: Graphic presentation of the generator

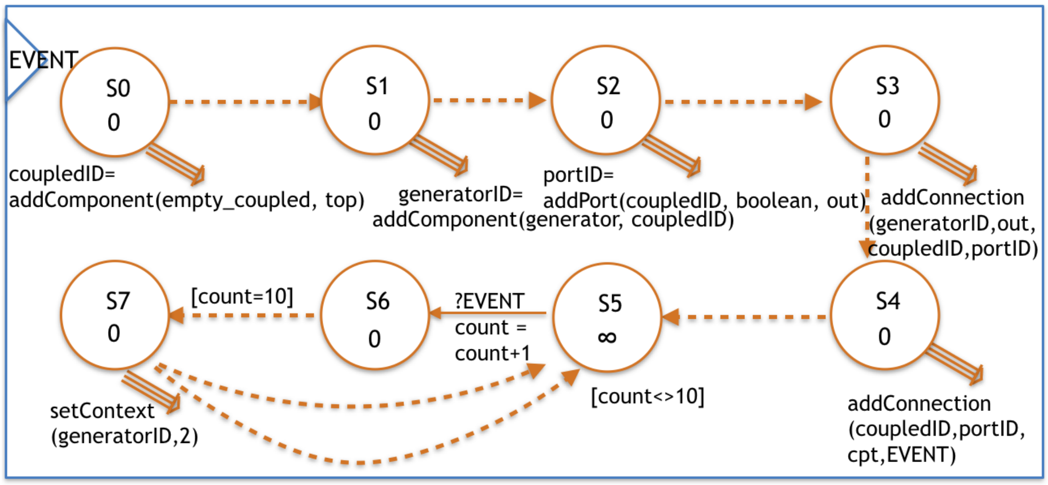


Figure 2.5: Graphic presentation of the counter

2.5 Conclusion

In this chapter, we presented a formal syntax and semantics for a partially reconfigurable DEVS. We also introduced a graphic presentation which helps for a clear interpretation of the formal definition of the model, at least for simple models.

Most functions defined in this chapter could possibly fail in some circumstances, but we do not want to handle exceptions or errors in this definition. In this meta-model, the error checking is up to the modeler. We could extend PRDEVS to handle error cases, such as illegal actions (e.g. connecting ports with different types). We can also think of implementation-related error cases, such as creating a new component while there is no room available for the new component.

Structure Change (SC) functions have a different priority level than other messages in the simulator: the simulator has a list of pending SC tasks and the list is executed only at the end of an imminence cycle. As a first approach, we chose to execute all SC functions in zero time relative to the simulator, i.e. the simulator is paused while the structural changes are carried on. In future work, however, it could be possible to allow structural changes to be applied while the simulation is running in order to allow taking full advantage of hardware partial reconfiguration technology. This will require structural checks on the model such as making sure that a newly added component will not be required for simulation until it is fully operational. This can be carried on by separating the SC function call from their return, obtaining the new identifier on a separate external transition of the component which called the add function.

More possible extensions include dependency handling for calls. As an example, deleting a component still connected to other components could automatically delete its connections prior to the component removal.

Chapter 3

A software PDM for the PRDEVS PIM

Contents

As a first execution platform, we will study a generic software architecture. From this, we will build a Platform Definition Model and match it with the previously defined PRDEVS meta-model.

Our final goal for the PRDEVS meta-model is to target partially reconfigurable hardware platforms. However, preliminary tests are required to verify our PIM syntax correctness and applicability. The dynamic functions can be also be tested more easily on an object-oriented software implementation, as hardware platforms are not as flexible as software.

A strong advantage of testing on software first is that the software concept is very close to our PIM. For example, the dynamic functions like *addComponent()* and *removeComponent()* directly match the *new()* and *delete()* operations on objects. Another strength of software is its flexibility, notably the memory use, virtually illimited, and the communication ease between objects, which can rely on function calls. Finally, software is inherently sequential, i.e. an operation is carried on only when the previous one is over, which avoids concurrency issues. We will not deal with threads and other multiple process capabilities.

In this chapter, the first step will be a general study of software platforms, which is detailed in section 3.1. It also will be of help to review an existing discrete-event simulation platform. Then, we will present the hierarchical architecture of the PDM in section 3.2. A mapping to PSM is presented in section 3.3, especially as we present the timing mechanism during the simulation. At the end of this chapter, we give a testing example of this software PSM in section 3.4.

3.1 Platform study

Most of the DEVS models are simulated under a software simulator. For example, PythonPDEVS [Van Tendeloo 2016] is a Python-based simulation kernel for both CDEVS and PDEVS. It is designed by The Modelling, Simulation and Design Lab (MSDL) at McGill University in Montreal and the University of Antwerp. DEVS-Suite [Kim 2009] is a Parallel DEVS simulator. It is a tool created at Arizona center for integrative modeling and simulation. ProDEVS [Vu 2015] is a DEVS simulation tool for hybrid systems using state diagrams. The advantage to study ProDEVS is that it is an in-house made tool at LAAS, where the source of the project is accessible. Other simulation software frameworks using Discrete Event System Specification can be found under survey [Franceschini 2014b], as CD++ [Wainer 2002], DEVS-Java [Hu 2008], James II [Himmelspach 2009], PowerDEVS [Bergero 2011] or DEVS-Ruby [Franceschini 2014a].

3.1.1 Software platform and object-oriented programming

Software applications are sets of instructions executed sequentially by a processor. Multi-core and multi-processor platforms allow treating several tasks at the same time. However, we decide, as a first approach, to only consider single-threaded execution.

A widespread approach to software development is the object-oriented method. Objects are sets of variables, along with methods, defined in classes. The variables are stored in memory, and the methods are sets of instructions that can be applied to the object to alter the variables content. The methods range from single accessors (getters and setters) to large pools of code carrying out complex operations. The object approach suits our goals, as a component can be represented as a class.

Moreover, objects are blocks which we can create and delete easily, which match our needs for structural change. The methods in objects can be called and the affected behaviors are applied. The variables of each object are kept as properties. Methods can create and change others properties. Methods of different objects can share the same name but distinct behaviors. Several objects can also share public properties.

3.1.2 A quick look into a software simulator core

To have a good understanding of how a simulation engine must behave, we chose to take a look at the source code of an existing DEVS simulator.

ProDEVS is an event-driven modeling and simulation tool for hybrid systems (continuous and discrete time systems). It is a team internally developed tool and we have access to the source code. ProDEVS applies a model-view-controller (MVC) design pattern, as shown in figure 3.1.

★ View: User builds the model using the graphical interface. For each visual component, the tool creates a corresponding internal object. The user defines the components hierarchy, connections, and behaviors. A model check is done prior to the simulation. During the simulation phase, the controller is in charge of the computation. After the simulation ends, the results of the simulation are shown in a graphic interface, displaying the variables selected by the user.

★ Model: *DEVSModel* is the class which records the whole DEVS model, as shown in figure 3.2, together with the next identifiers of atomic component, coupled component, input port, output port, transition, state and connector. The *Component* class can either specify an atomic component, or a coupled component. The common attributes such as the id, component name and

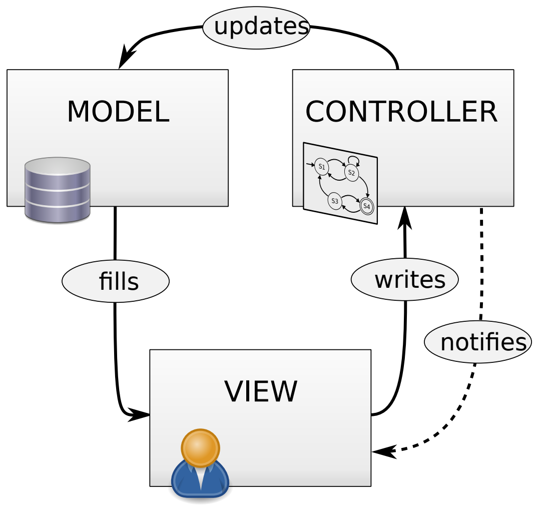


Figure 3.1: MVC Process
(Image "MVC Diagram (Model-View-Controller)"
by Grégoire Surrel, available on Wikimedia Commons, CC-BY-SA-3.0)

input/output ports lists are specified in the *Component* class. The *Component* class also manages the attributes e, σ, t_n and t_l which are used during the simulation phase. The time structure and the scheduling will be detailed in the section 3.3. *AtomicComponent* class contains the states which are linked by transitions, while the *CoupledComponent* class records the hierarchy and connections lists.

★ Controller: The controller has many tasks. A first is to establish the link between the view and the model in the design stage. After the model is completed, during the simulation phase, the *Controller* manipulates the model. The figure 3.3 shows the initialization process of the Controller. The controller runs the simulation time from 0 and advances simulation time until the end time defined by the user. It activates each Component one by one if the simulation time matches its own internal time specification (t_n). There are two different controllers inside the tool: a Classic DEVS and a Parallel DEVS.

The study of this DEVS simulation tool structure helped us understand the functioning of DEVS and PDEVS simulation. The MVC structure is convenient for a medium size development. It cuts the major parts of the software into several sub-systems for partitioned development. We chose not to use MVC structure for our PSM development due to several reasons.

First, the MVC structure of ProDEVS requires to modify the GUI (Graphical User Interface), which is time-consuming work that is not part of our research focus. Moreover, the structure of the existing tools needs lots of mod-

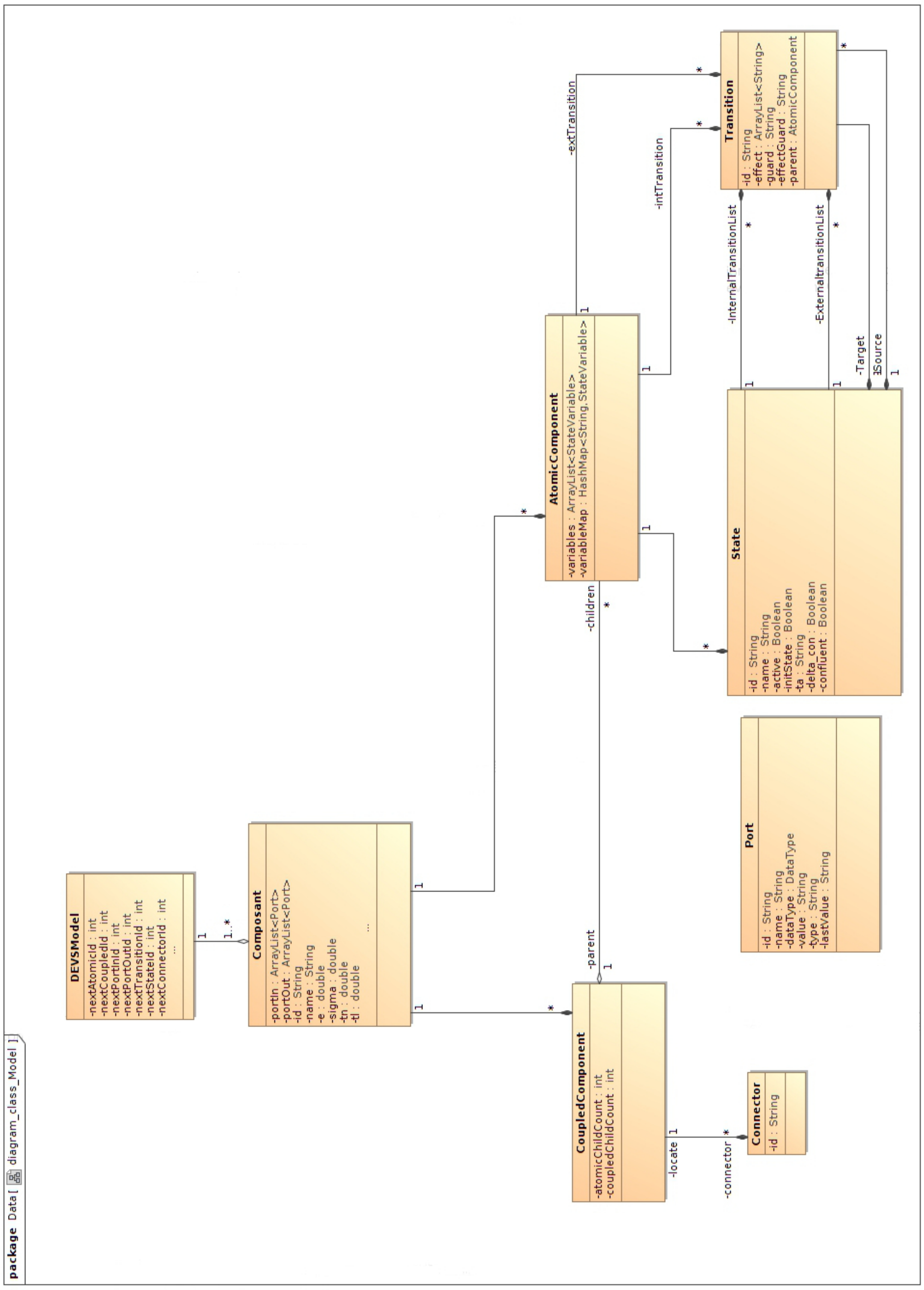


Figure 3.2: Class diagram of Model package

ification. E.g. the top level of the model should contain a library of available components, according to our PRDEVS meta-model definition. This change would lead to a different DEVS model verification, code generation, etc. Finally, a PRDEVS model needs a different controller able to treat the dynamic messages and to modify the model structure on-the-fly, which is difficult to achieve considering the static structure of ProDEVS.

In order to implement a PIM into a software PSM and test it rapidly, we are going to write the model source code manually. Building it from scratch seems here faster than modifying an existing tool.

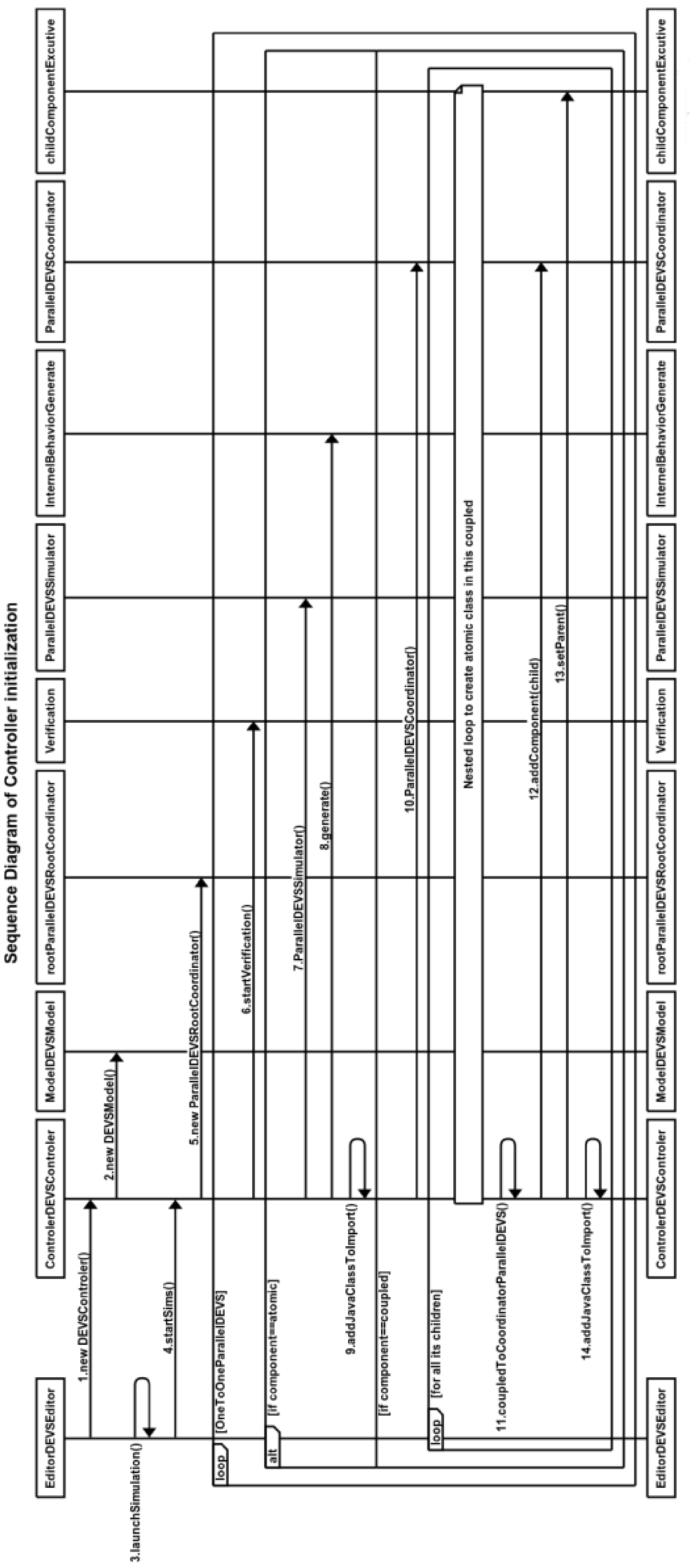


Figure 3.3: Sequence diagram of controller initialization

3.2 Hierarchy of the software PDM

In our PDM, to match the predefined PIM meta-model there will be a unique top-level coupled model (C^{TOP}). A library (L) contains the available components.

The UML representation of the PRDEVS is shown on figure 3.4.

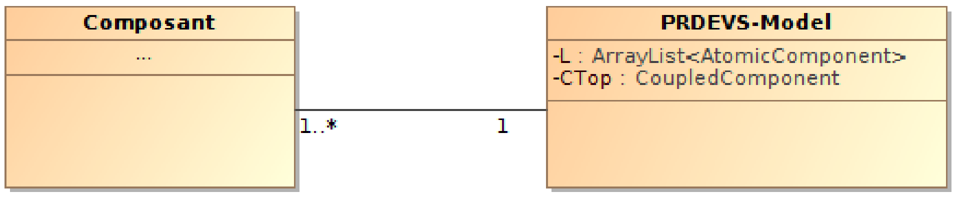


Figure 3.4: PRDEVS UML representation

3.2.1 Atomic/coupled component

Atomic and coupled components inherit from the *Component* class. The *Atomic* and *Coupled* classes share some common characteristics: there are both with their ID, inputs set and output set. Moreover, atomic components have two types of transitions and two types of output emissions. Atomic components have one parent. The coupled component C^{Top} has a list of its children.

The UML representation of the Component classes is shown in figure 3.5.

After one-to-one correspondence, the simulator which represents atomic component inherits different variables:

- current_state, current DEVS state in simulation
- next_state, next DEVS state if an internal transition is active
- time_elapsed_on_current_state (e), simulation time has already passed on current DEVS state
- time_of_next_event (t_n), the time condition for the internal transition to be actived on current state. If no internal transition, t_n can be infinity
- time_of_last_event (t_l), time for which we have been in the current state

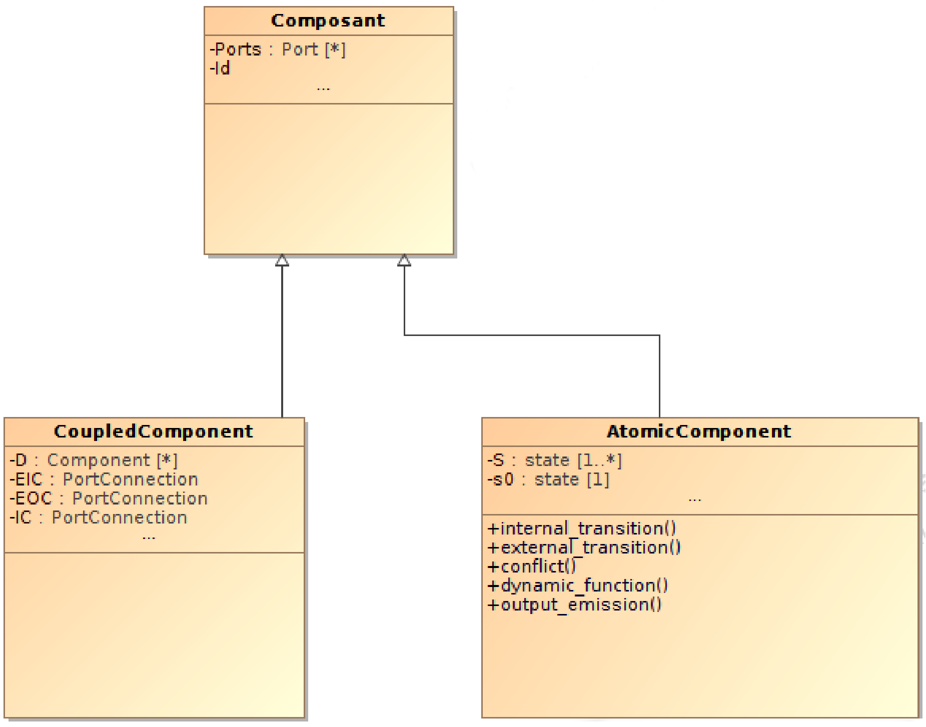


Figure 3.5: Components UML representation

The coordinator which represent coupled component anlso inherits variables:

- time_minimal_for_next_event(tn_min), the minimal tn among its children
- IMM, the list of its imminent children.

3.2.2 Port/connector

Each component has a list of input ports and a list of output ports. The ports are in charge of receiving and sending the messages bags. Each port can either be an input or an output port. It records the type of messages bags that it accepts (integer, real or boolean). Under our OOP development, *Port* is a class which can be instantiated several times.

The UML representation of the *Port* and *PortConnection* types are shown on figure 3.6.

The output ports record which input port it is linked to. The input ports have a list of input values that have not been treated yet. The exchange of the

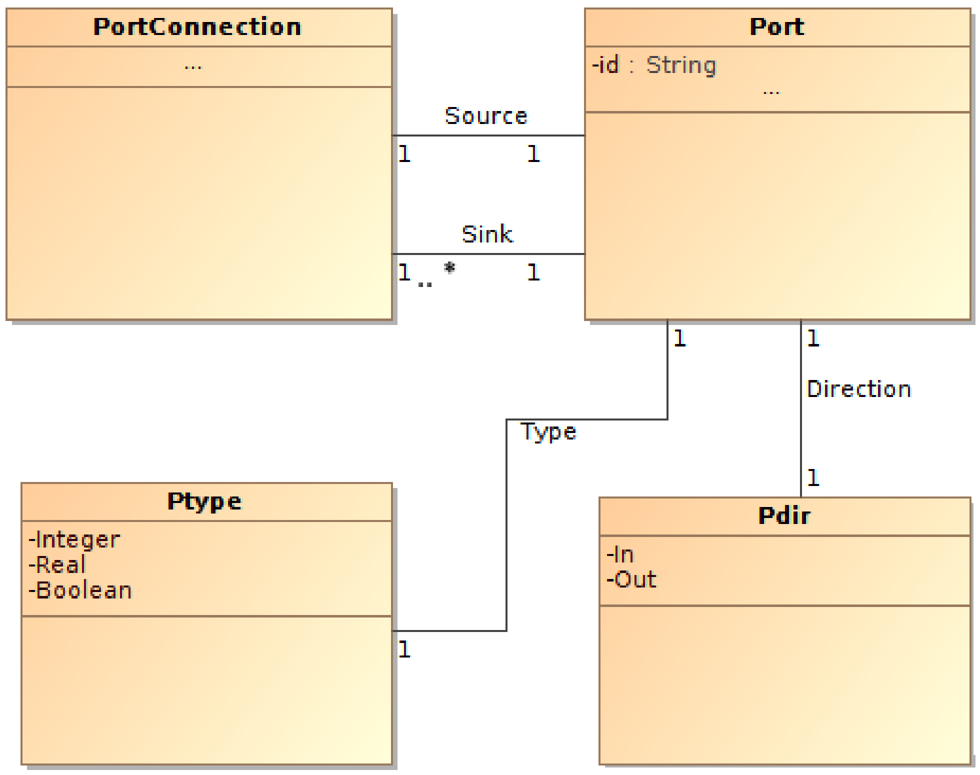


Figure 3.6: Port and PortConnection types UML representation

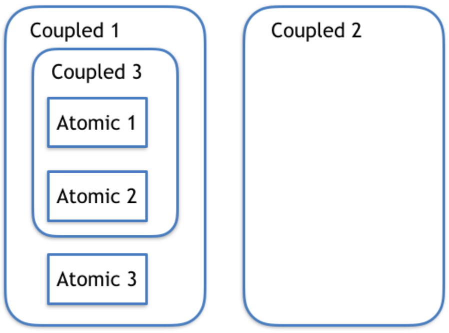
information happens at the end of one simulation cycle. According to Ziegler's definition, the connection is one way. One output port can connect to several input ports. However an input port can receive from only one output port.

3.3 Mapping and realization of the PDM

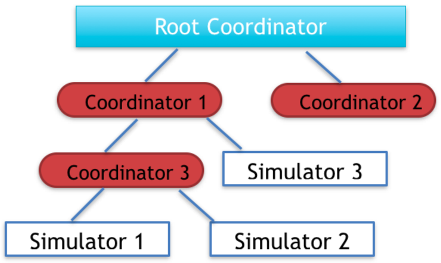
The simulator described by Ziegler uses a hierarchical tree of models, which has a coordinator object for each coupled model and a simulator object for each atomic model.

To each of the simulator objects, a model object describing the structure of the represented atomic is associated. There is a single root coordinator which lead the hierarchical tree. The root coordinator contains a list of imminent models and their next event time. This list is updated at the beginning of each event step.

Under root coordinator are coordinators which can be the parents of coordinators or simulators which are the leaves of this hierarchical tree. A correspondence from model to simulation can be seen in figure 3.7. Each branch of the hierarchical tree is independent during the simulation. Simulators manage the next time event. Coordinators control the next simulation time of all its children. We name this tree building a one-to-one correspondence process. This is part of the initialization phase.



(a) Model



(b) Controller

Figure 3.7: One-to-one correspondence

In a first approach, a flat representation is chosen for our implementation of PRDEVS. That is to say, the levels of hierarchy are ignored as if all the atomics were directly instantiated in C^{TOP}. In this organization, only atomic components will be in C^{TOP}, so there is only one coordinator in the model (C^{TOP}). A list of available components L is held in the root coordinator. A simulator which corresponds to a component type in L can be created using the SC-functions.

To apply the object-oriented programming (OOP), several python based static files which describe the components in L are created. The controller starts the simulation with an initialization phase: put the initial python classes into C^{TOP}. During the simulation, dynamic functions correspond to the methods of a class. An addComponent() function call can be applied to create a new object from a class in L.

Example:
when addComponent(T,ID) function is called, a new objet is created.
Latest id is updated: id_{last}++
The new objet is linked with the lastest id in the model: objet.id=id_{last}
The object is referenced into its parent: parent.addchildren=object
And the object parent is set: objet.addparent= parent
Then we return the new id: $return_{id} = id_{last}$

The core simulation contains a cycle which starts from time 0 utill the end of simulation time set by the user. There is a global simulation time t in the controller. Each coupled component has its next event time (tn_{min}). Each atomic component has its own separate local time (t_l,t_n,e). Each state in the atomic component has its own time advance (t_a), as shown in figure 3.8

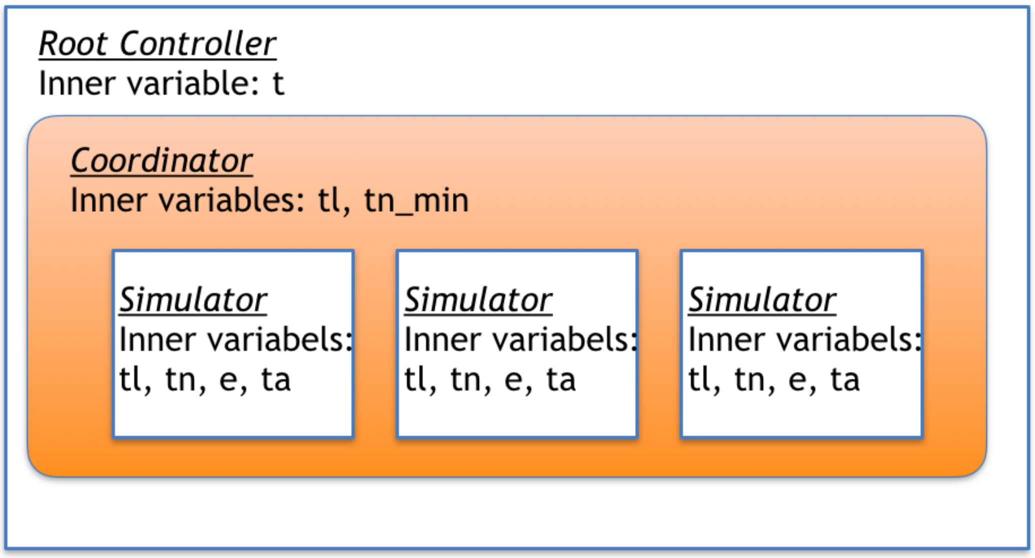

Root Controller
Inner variable: t
Coordinator
Inner variables: tl, tn_min
Simulator
Inner variabels: tl, tn, e, ta
Simulator
Inner variabels: tl, tn, e, ta
Simulator
Inner variabels: tl, tn, e, ta

Figure 3.8: Time variable in the different objects

In order to get the information of the lower level, there are getters and setters for each variable.

3.3.1 Meta model transformation for time

The PSM gets not only the structure from PIM, but also the time specifications from the PIM.

In our software PDM, the memory allocated to the simulation is virtually unlimited. It means we can add as much components as we want to into the model. In the model, each atomic component registers its time in the state class, we name it time advance, or τ in the formalism, Ta in programming. Transition connecting the states are triggered if the time and guard are met for internal transition, or input and guard are met for external transition.

The simulation time t is coordinated with Imminent Time ($T_{n_{min}}$). $T_{n_{min}}$ is calculated on the basis of T_n. T_n is based on Last event Time (T_l) and all the Time advance (T_a) of current state in components.

3.3.2 Input/output message bags

Our PIM is based on PDEVS, the parallel DEVS meta-model, where several components can send the messages at the same simulation cycle. So a message bag is used for the output event. The message bag contains an unsorted set of events from one or more different sources.

Each component is linked to one or several ports classes. Ports can hold a list of messages which is not treated yet. The messages delivered within the ports are one way. Under the software implementation, a port has a type. e.g. if a port is defined to handle numeric messages, it can not treat text messages. We define 4 types of messages: X-message, Y-message, *-message and SC-message which correspond respectively to input messages, output messages, state change messages and dynamic messages.

The first three messages are defined as Zeigler's: imminent models are chosen based on their minimal next event step and the imminent models trigger *-message. If the conditions to trigger λ on the current state are met, a Y-message is then integrated into the Y-message bag. The message bag is sent to the target model and is received as a X-message at the end of each cycle event. The model which received the X-message will move to upcoming state and wait for next cycle. SC-message, in some aspect similar to Y-message, will build a SC-message bag and the dynamic SC functions are executed at the end of each event cycle. The SC-message is then treated in zero-time compared to the simulation time.

3.3.3 Dynamic functions calls

Dynamic functions are collected and treated in priority than other messages. If a dynamic message is applied during the cycle, the other messages are suspended.

If the Dynamic functions exectued only on itself, such as addPort() or removePort() for itself, it does not influent other component's state. The addConnector() and removeConnector() can influence two components. However no messages are sending within the new connector in the same cycle since the other actions are suspended.

3.3.4 Time advance (Ta)

Ta matches T_a by Ziegler. Time advance is different from component to component, from state to state. Ta accepts positive integer numbers. Tn_min is calculated on the basis of all ta in its children.

3.3.5 Imminent time (Tn_min)

Imminent time is calculated based on the next event time of the imminent simulator. The minimal next event time of the imminent simulator becomes the imminent time of coordinator at the beginning of the next simulation cycle.

3.3.6 Last event time (Tl)

Each simulator keeps its last event time. Based on the last event time and current time advance, the next event time is calculated.

3.3.7 Elapsed time (e)

Elapsed time saves the time passed in the current state. If it equals to the time advance, the current state is finished and the simulator can move to next state.

3.4 Case study: a PSM test

We apply PRDEVS syntax by creating a PSM implementing on a basic Multi-Agent System (MAS). A multi-agent system consists of independent agents and their environment.

MAS is an upraising research domain of artificial intelligence (AI). AI is the study of agents that receive percepts from the environment and perform actions [Russell 2016]. What it treats is a system with multiple agents, act independently but possible has an interface between them. Agents are autonomous and have skills to achieve their goals and tendencies [Ferber 1999]. An agent can be a physical or virtual entity: network servers under the same root server or simulated players in video games. Since each agent has its own configuration, it is possible modeling it and simulate its actions.

MAS offers a conceptual approach to include multi-actor decision making into modeling and simulation [Ligtenberg 2004]. Where several studies [Uhrmacher 2009] have mentioned simulation for MAS and MAS for simulation, such as combining simulation and formal tools for developing self-organizing MAS [Gardelli 2009] or agent for traffic simulation [Kesting 2008].

There are also semi-formal modeling approaches: Agent Modeling Language (AML) [Trencansky 2005] for specifying, modeling and documenting systems that incorporate features drawn from multi-agent systems theory.

Very often [Narendra 1990, Qu 2009], when we talk about dynamic models, we mention dynamic behaviors. Then it leads to systems modeled by differential equations or equilibrium points, stability, limit cycles and other key concepts of dynamical systems.

However, our study of dynamic systems, based on reconfigurable computing, pointed to the fact that a system can keep running with a part of it being changed. For example, one program in the school syllabus can change to another (due to shorthanded or other reasons) without interrupting the fulfillment of the system goals.

Within a $size \times size$ grid co-exists three types of players: chicken, fox, and egg. Each cell can hold only one player and players move under certain rules, as shown in figure 3.9: chickens can move randomly around in four directions – south, north, east and west; foxes can move randomly around in all eight directions including south-east, north-east, south-west, and north-west; eggs cannot move.

The general position of each player is registered in component *rules*. When a player needs to know if its neighbor position is free, it contacts *rules*.

Chickens and foxes components all have one input port: *validation* and one output port: *askAvailability* for checking the position.
$P_{validation} =< validation, in, \{isChicken, isFree, other\} >$

.	$\uparrow$	.
$\leftarrow$	c	$\rightarrow$
.	$\downarrow$	.

(a) Chicken

$\nwarrow$	$\uparrow$	$\nearrow$
$\leftarrow$	f	$\rightarrow$
$\swarrow$	$\downarrow$	$\searrow$

(b) Fox

.	.	.
.	e	.
.	.	.

(c) Egg

Figure 3.9: Moving Rules for players (c, f, e represents Chicken, Fox and Egg)

$P_{askAvailability} =< askAvailability, out, (positionX \in size, positionY \in size) >$
Players calculate their destination themselves and communicate with a *Rules* component to know if the cell is free for moving. An example is shown for Chicken in figure 3.10.

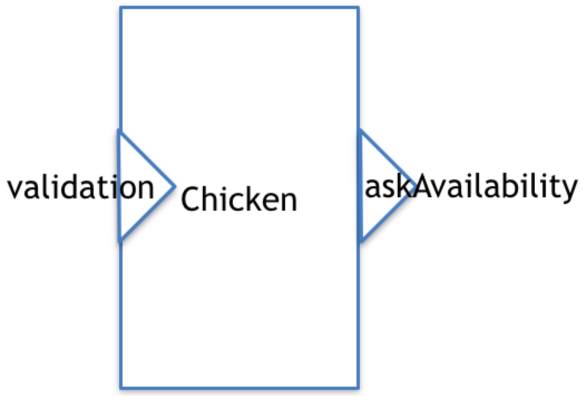


Figure 3.10: External view of chicken

The DEVS state machine for the Chicken model is as shown in diagram 3.11. The initial state of a chicken is $S1$. When the chicken component is imminent, it receives a *-message to execute the internal transition and randomly defines the desired destination. It moves to state $S2$. Then an output is sent to verify the availability of this position. It moves to state $S3$ and waits for an input. The Y-message arrives to the rules model which responds according to the availability. Depending on this response, the chicken model moves to state $S4$, $S5$ or $S6$ and then the SC-function and the internal transition is called.

The game starts with an initial numbers of players, an example is presented on figure 3.12a. Each round, all players are imminent and move simultaneously. If a chicken reaches another chicken, an egg will be laid randomly around and the chicken stays at the same position. If a fox reaches a chicken,

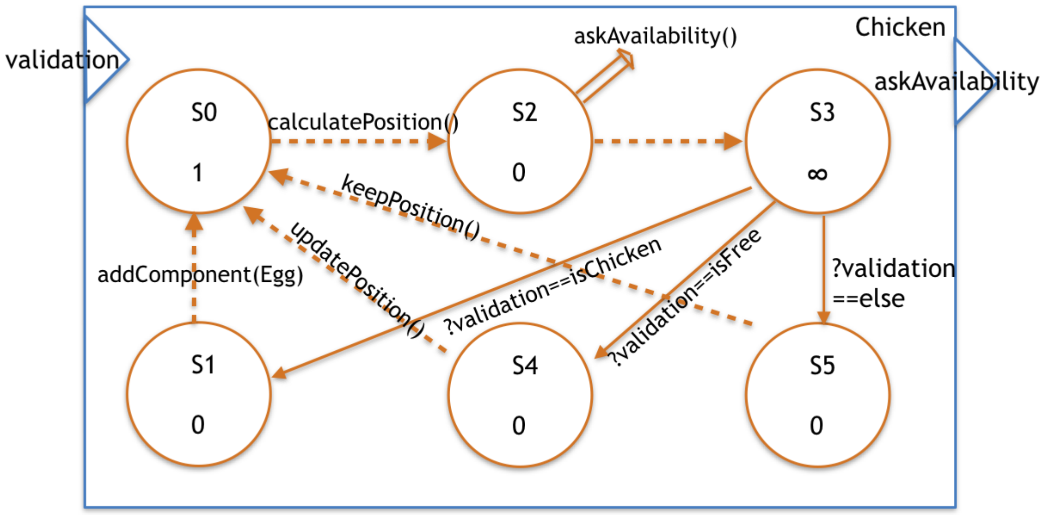


Figure 3.11: Internal view state machine of chicken

the chicken is eaten and the fox replaces the position of chicken. The game ends when there are no chicken any more, as in figure 3.12d or if all foxes are blocked by eggs.

c	c	.	.
.	.	f	.
.	.	.	.
.	e	.	.

(a) Game Starts

.	c	.	.
c	e	.	.
.	f	.	.
.	e	.	.

(b) Round 1

c	.	.	.
.	e	.	.
f	.	.	.
.	e	.	.

(c) Round 2

.	.	.	.
f	e	.	.
.	.	.	.
.	e	.	.

(d) game end

Figure 3.12: Example of a game turn

3.4.1 Initialization of the model

Before the simulation starts, an initial model is created, as shown in the example figure 3.12 for the example game turn. Under the PRDEVS model, there are a C^{Top} and L. The component types C, F, E which represent Chicken, Fox and Egg in L. Rules is the component which is in charge of updating the global geographic position.

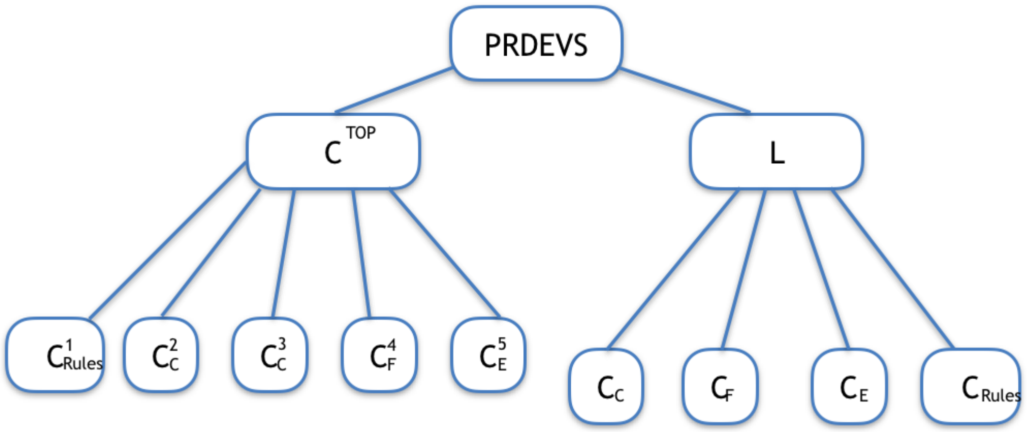


Figure 3.13: Tree view of model

There are two simulators corresponding to atomic components with type chicken, a simulator associated with an atomic component with type egg and a simulator which corresponds to an atomic component with type fox. They are all in the coordinator corresponding to C^{Top}.

3.4.2 Simulation

At the beginning of round 1, there are four players. Since one chicken meets another chicken, a SC-message is sent and an egg is created at the end of this round. At the beginning of round 2, there are five players. During round 2 fox finds a chicken and a SC-message addressing removeComponent() is sent. The SC-message is not treated immediately but added to a SC-message bag which will be executed at the end of this round. For this reason, even if the chicken has already been eaten, it still effects its internal transition within this round (it is still possible to meet another chicken for laying egg). The simulator which corresponds to the chicken is removed at the end of this round. At the end of the game, all chicken are eaten and a new egg is created. In the root coordinator, only three simulators are left in C^{Top}.

3.5 Conclusion

The PRDEVS meta-model with a system engineering approach smoothes the application to the different platform. Before describing a detailed hardware platform, a software simulation platform has been described first. This has allowed us to test the limitations of the PIM and the description needed to define a complete platform. A clear definition helps us ease the implementation on hardware. The management of the messages bags, the priorities of the messages and the scheduling have been tested under the software platform. The implementation on a software platform is different from the hardware platform. For example, the messages are sending virtually, the structure change is done without delays.

A software PSM has been presented with a case study of multi-agent system. A software implementation PDM has been defined, the realization specification made. PRDEVS model structure and dynamic functions have been verified.

CHAPTER 4

An hardware FPGA PDM for the PRDEVS PIM

Contents

The goal of our study is to define a hardware PDM compatible with PRDEVS PIMs which covers dynamic reconfiguration capabilities. Advantages of the partial reconfiguration have already presented in section 1.3. The difference between a software PDM and a hardware PDM is that we have more constraints, as we must totally define the architecture while the software already relies on an operating system and language libraries. However, the fundamental difference is that the hardware can treat parallel tasks natively.

In this chapter, we are going to build a PDM based on a hardware design, more specifically, a Field-Programmable Gate Array (FPGA) based PRDEVS PDM, with partial reconfiguration capabilities.

First, we must consider the constraints of current FPGA technology, notably the partial reconfiguration constraints. A detailed definition of the PDM architecture is in section 4.1 with a structural, spatial definition and its link to our PIM meta-model. A temporal scheduling of communications between different actors is defined in section 4.2. Then the communication interface between different parts of the model are presented in section 4.3. A transformation of PRDEVS atomic models to finite state machines is presented in section 4.4. The structure change functions are explicit in section 4.5. Finally, a PSM with the details of implementation on FPGA is presented in section 4.6.

4.1 PDM architecture: a high-level view

According to the definition of our PIM meta-model, there are two types of components: atomic component which is the smallest unit and coupled component which can hold other components. A PIM has a hierarchical architecture where two components can be connected by ports.

Current partial reconfiguration technics are based on independent, isolated reconfigurable areas. On partially reconfigurable materials, a hierarchical architecture is complicated to carry out [DeHon 1999]. Thus, as a matter of simplification, we will not consider model hierarchy in the hardware PDM.

The lack of hierarchy on the hardware level can be seen in two different ways. The first one is to consider that the PIM hierarchy exists in the simulator high level as a data structure. The components then have the virtual notion of coupled that is matched with the actual, flat-structure, hardware by the simulator. Another way to see it is to totally remove the notion of coupled components in the PDM. Considering the second approach is a restriction of the meta-model, and a PIM must then be adapted to the PDM.

As a first approach, we chose to use the simplified version where no coupled exist in the PRDEVS (except C^{TOP} which match the whole FPGA). This is not an issue as the hierarchy can be introduced later using the data structure approach without modifying the hardware structure.

In PRDEVS there are three levels of reconfiguration: components, connections and context. Components reconfiguration match the concept of reconfigurable area. However, connections reconfiguration is difficult to build with this concept. We could consider connections between components as several reconfigurable areas and update these areas on demand. But since we do not know the numbers of total connections in advance, the numbers of this areas would be unknown. For this reason, we propose a bus structure which can transfer all types of data and is connected to all components whether a direct connection is established or not. The context dynamic capacity is at the data level and requires no structure reconfiguration.

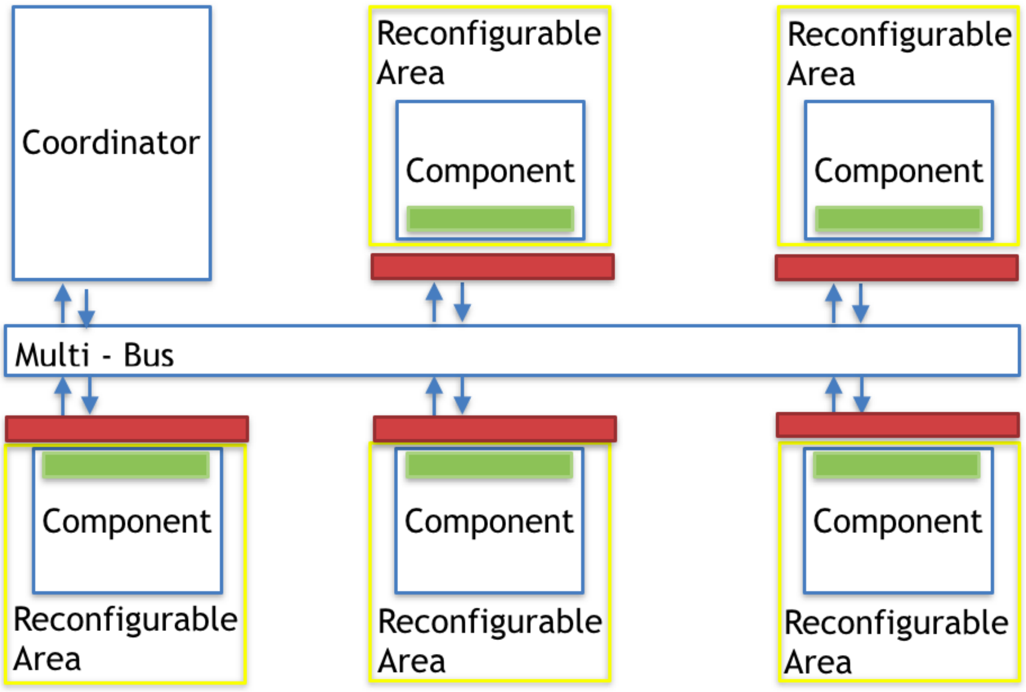


Figure 4.1: Global view of PSM

Reconfigurable areas We present the proposed FPGA structure in figure 4.1, with several reconfigurable areas. Each reconfigurable area can hold zero or one atomic component at a time. The component does not have to make use of all the resources of the reconfigurable area. It is not possible to add a component which needs more resources than the definition of a reconfigurable area.

For example, as presented in figure 4.2 a new component C is added.

In reality, the new component C is loaded in an empty reconfigurable area,

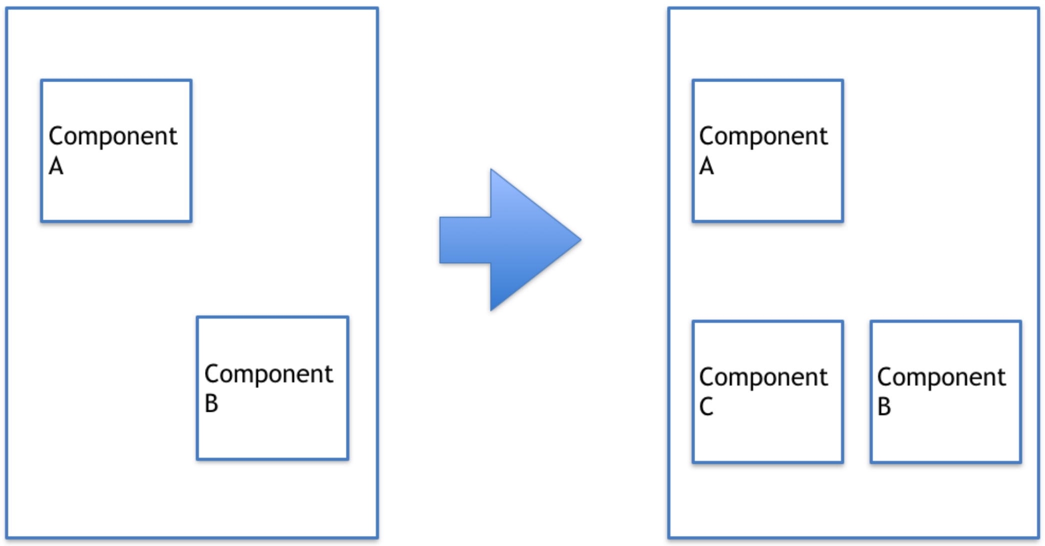


Figure 4.2: Example of addComponent into the circuit

and a connection to the multi-bus is etablished automaticaly, as presented in figure 4.3.

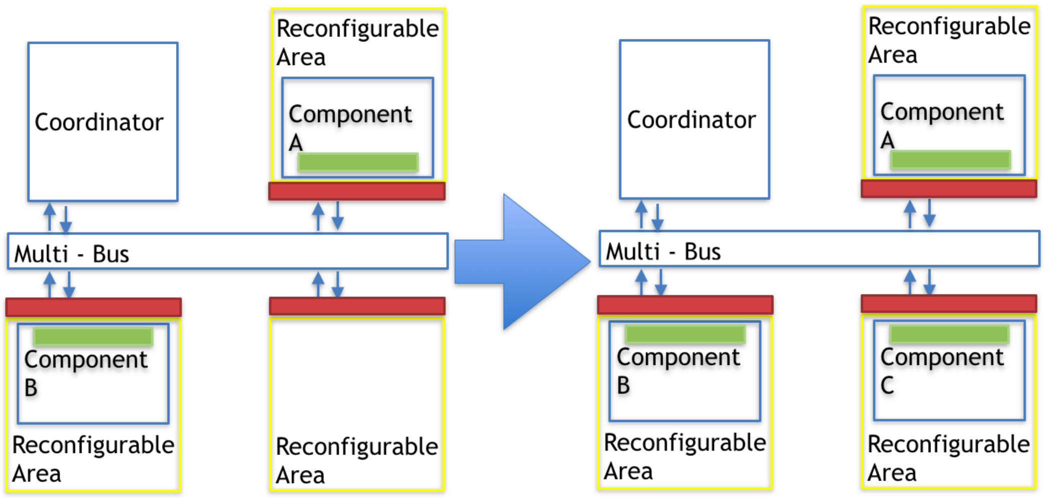


Figure 4.3: Architecture of a new component added

All component areas include a decoupler, in red on the figure, that isolates the static part from the dynamic part, preventing the reconfiguration from emiting random signals to the bus.

Communication For the purpose of linking the components, we define a communication bus which is in charge of the message exchanges. On the figure, it is part of the multi-bus. On the communication bus, the messages between components are broadcasted, and inside a component, the green area represents a communication block which filters the input messages. The communication block contains the identifier of the remote port which it is linked to. As the connections change the remote port identifier must be updated, which means it is contained in a writable memory.

A connection change, for example, an A-C connection to an A-B connection, as shown in figure 4.4 can be decomposed in two steps: remove the A-C connection then add an A-B connection.

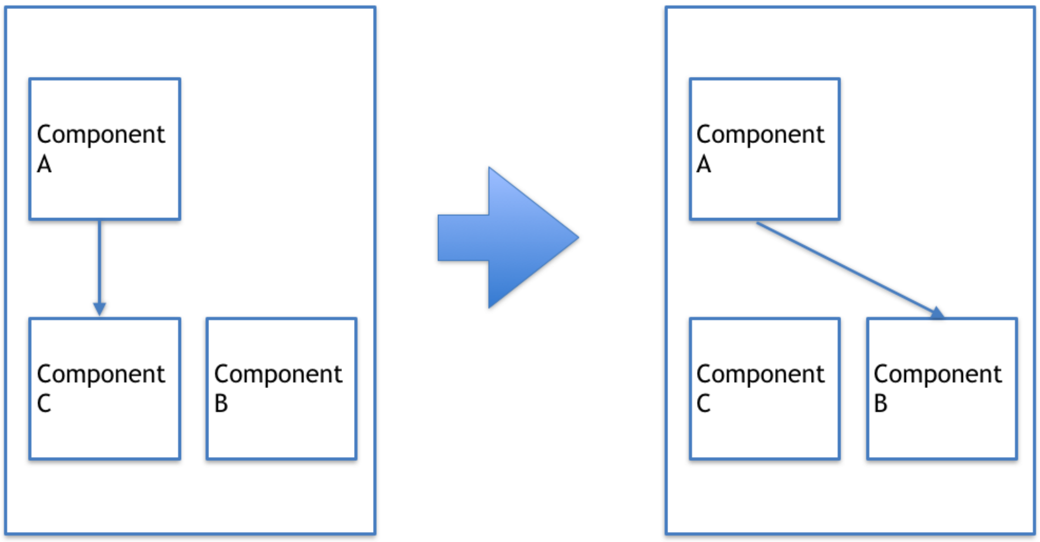


Figure 4.4: Example of a connection change

In reality, the connections change are done in the communication block and only this part is changed without change of the communication bus, as presented in figure 4.3.

Control Apart from the components, there is only one coordinator connected to the multi-bus, leading the simulation and synchronizing each cycle of simulation. The coordinator is considered a static component. Details of the control interface are defined in section 4.3.1. The coordinator acts on the components using a control bus, which is also part of the multi-bus.

Structure change capabilities The components are also able to call dynamic functions, which will be handled by the coordinator. To do so, we introduce a third bus in the multi-bus: the SC bus. The SC bus will contain the structure required to call the functions, transmit the parameters and

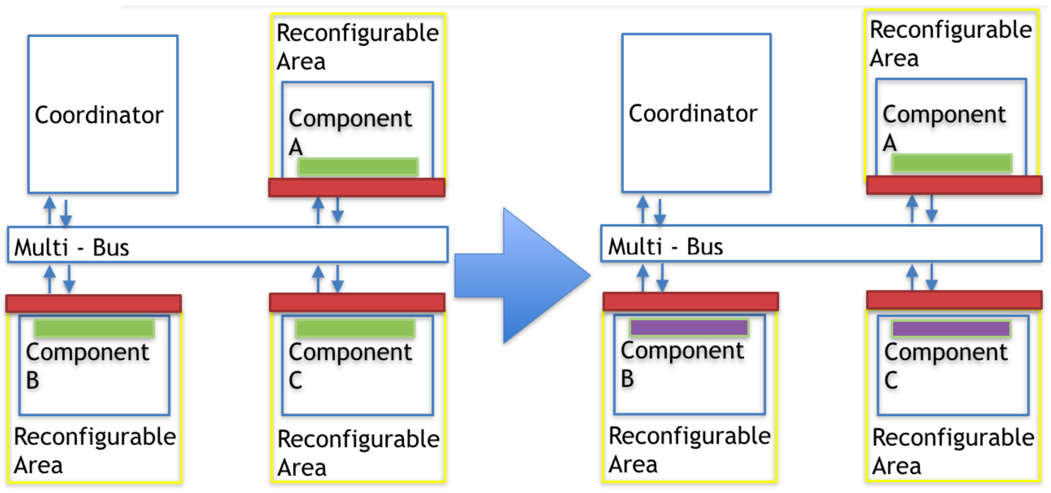


Figure 4.5: Architecture of a connection change

obtain the return values. It will then be deeply linked to the SC-functions signatures.

The SC bus will also contain the interface required to update the identifiers within the communication block for connection updates. Finally, the context will also be handled by the structure change bus.

4.2 Scheduling and temporal aspects

Time advance of a DEVS simulation is based on an absolute next event time (tn) calculated from relative time (ta) of the current state of each component.

For our PDM, a cycle of simulation starts when the coordinator broadcasts the minimal next event time (tn_min) along with the `step` order. tn_min is transmitted using a signal `tn_min` on the control bus. All components receive the order and check if the `tn_min` value corresponds to its local tn. If so, the component is imminent and performs an internal transition and/or an output emission.

After a component finishes its local cycle, it returns its new t_n to the coordinator using the `tn` signal on the control bus, along with the `stepped` event. A cycle ends when all components have returned their own next event time. This sequence as shown in figure 4.6

A communication between two components is initiated by a component during its step cycle. However, the target component can have already sent its `stepped` event when it receives the communication.

As shown in figure 4.7, even when component A announces the end of its cycle, component B can still send messages to component A. To allow this,

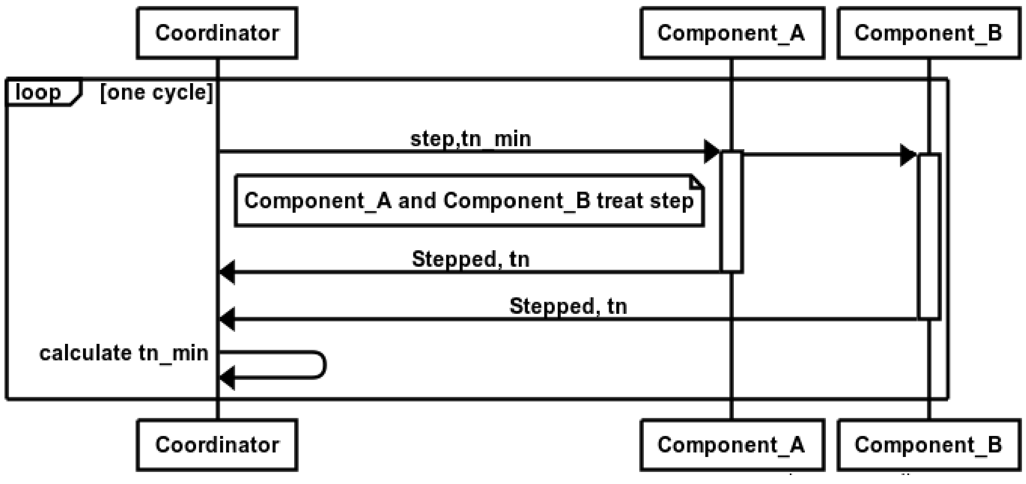


Figure 4.6: General simulation cycle

we separate the message handling mechanism from the simulation cycle. The message received is retained by the communication block and will be treated in the next simulation cycle.

4.3 Components interface

There are several buses in the set of multi-bus in order to handle control orders, dynamic structure changes orders and transmit messages between components. In this section, we will detail the low-level interfaces of these buses.

4.3.1 Control interface

Conforming to the figure 4.7, the coordinator needs to send a `step` event along with `tn_min`. `step` is a one bit signal and `tn_min` is an integer. The diagram also tells that we need a `stepped` event from component, together with a `tn` value. `stepped` is a one bit signal and `tn` is an integer. We define a particular type `tn_type` to represent all time-related integers. The `tn_type` will be constrained by a maximum value, because the implementation has to know the number of bits required to represent a value.

For `tn_min`, we are sure that it is not an infinite number, the other way it would be a modeling error as no component is imminent. However, for `tn`, it is possible that the component has an infinite value on certain states. As

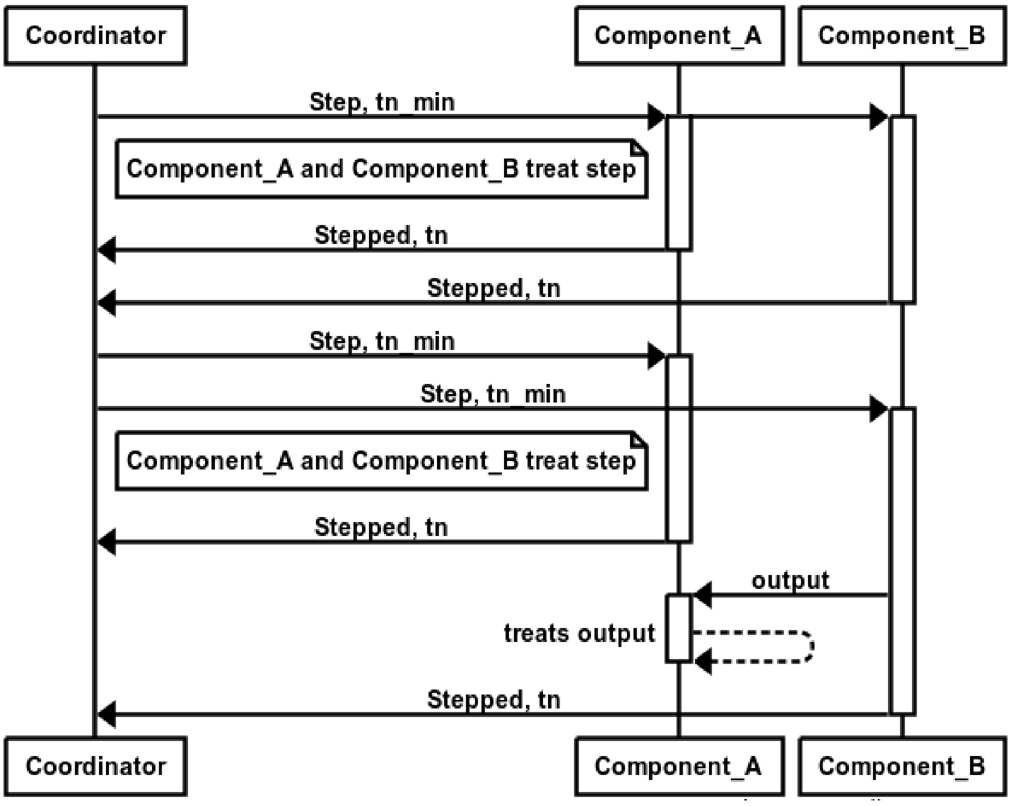


Figure 4.7: Example of communication between two components

we can not represent infinite as a binary number, we add an additional bit `tn_valid` to indicate that `tn` is not infinite. If `tn_valid` is equal to 1, then `tn` is not infinite. When `tn_valid` is 0, the `tn` value is ignored.

Several components can send the `stepped` event at the same time, but only one will be granted access to the bus at a time. We then add a one bit signal `ack_stepped` returned to the component, indicated that `tn` has been correctly captured by coordinator.

In summary, messages exchange between coordinator and components includes 6 signals: `step`, `tn_min`, `stepped`, `tn`, `tn_valid`, `ack_stepped`, as shown in figure 4.8.

Figure 4.9 is an example of the control bus communication with two components. The `step` broadcast announces the beginning of a cycle. `tn_min` is held until all component send the `ack_stepped` event indicating their end of cycle. The second cycle presents a bus conflict between the components.

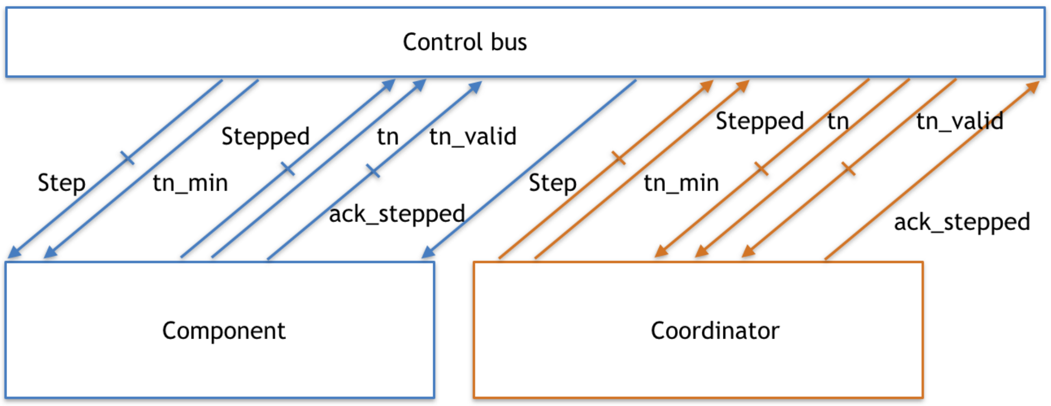


Figure 4.8: Messages exchanges between component and coordinator in control bus

Only one is granted the access, while the second one waits for its turn.

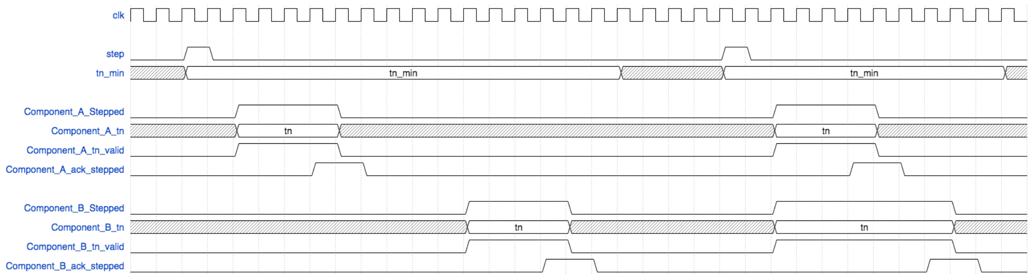


Figure 4.9: Wave of a cycle of `Step` under control bus

4.3.2 Communication interface

We will here use as an example a component with a X input and a Y output.

For input, we need a one bit signal `input_available` announcing the presence of a data transmission, and a vector `input_value` for receiving the input value. Moreover, as messages are broadcasted on the bus, an input value `input_sender` must also be present to indicate the identifier of the port emitting the data.

For output, the interface is a mirror of the input interface. We also need a one bit input to indicate that the port has been granted access to the bus.

According to the DEVS definition, one output port can be connected to one or several input ports, while one input port can only be connected to a single output port.

Considering that each input port can treats only one message at a time, a communication block is placed in the input side of each atomic component,

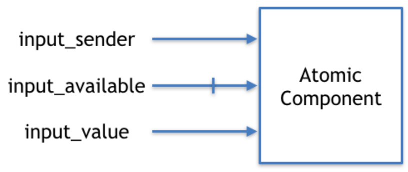


Figure 4.10: Input X

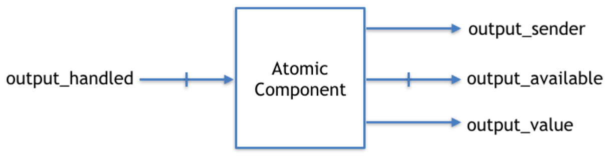
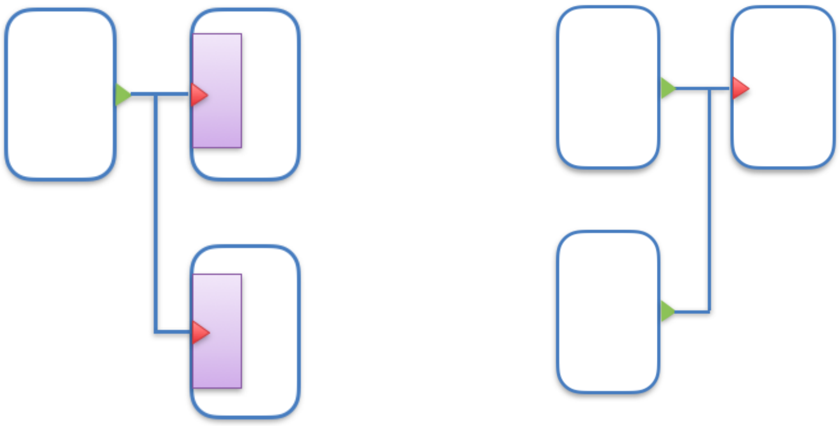


Figure 4.11: Ouput Y



(a) Authorized Connections (b) Non-Authorized Connections

Figure 4.12: Connections between components

acting as a buffer. It is in charge of checking the remote port id when a message is incoming. If it matches the port connection, the communication block will accept the message and indicate to the component that a message is available on the input port. Moreover, the communication block is also in charge of requesting access to the bus when the component wants to emit a message.

4.3.3 Dynamic functions calls

To handle λ_{SC} calls, we use a Structure-Change (SC) bus. Dynamic functions calls are initiated by the atomic component itself. They start by the assertion of a `sc` signal, linked to the SC bus.

There are several dynamic functions, which have different parameters and return values. In figure 4.13, we grouped the parameters as `Sc_information` and the return values as `Sc_return`.

`Sc_informaton` depends on the `Sc_type` which could be the Ids of the target component, $Type$ of the component, P_type of the target ports, P_dir of the target ports. `Sc_done` is an input message from the coordinator together with `Sc_return` which is the return of the structure change functions.

`Sc_return` is a vector and it arrives with one bit signal `Sc_done` which indicates the SC function has finished its treatment.

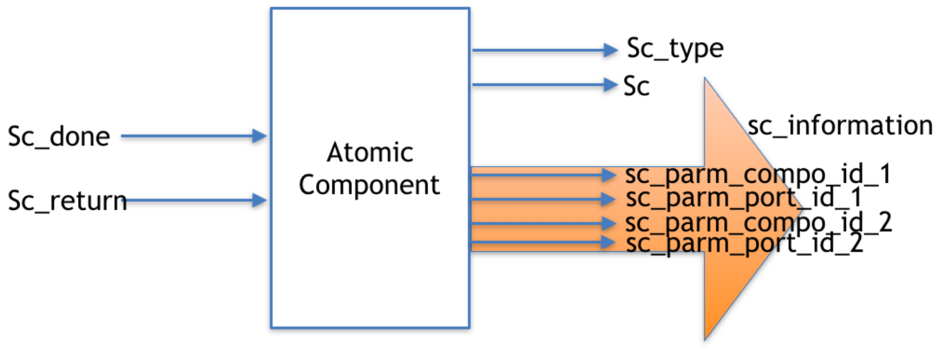


Figure 4.13: Dynamic calls extra ports

As specified before, we slightly restrict our meta-model for easing the implementation. As there are no coupled, the functions addPort and removePort have not been integrated. Moreover, the addComponent function does not require the id_{host} parameter, all added components being placed into C^{TOP}. The return value is always an identifier.

The SC bus is also in charge of updating the communication block of the component when there is a connection change. To do so, we define three signals: an `update_connection` bit which indicates that a connection change is occurring, and two signals carrying the new connection. The new connection is identified by the output port and input port full identifiers. A full identifier contain both the component and port identifier. All three signals are broadcasted.

When a communication block detects a connection update, it checks if the input port identifier is its own. If so, it updates its remote port identifier storage. When a connection is removed, the output port identifier is set to 0.

Out of lack of time, we did not have time to implement the context mechanism. However, it could be done by adding an additional interface to the SC bus. As for the connection update interface, the context interface should contain a signal indicating that there is a context change order, along with the target component identifier. The difficulty is that the context size can differ from component type to component type. A component is aware of its own context size, so there is no need to transmit it on the bus. To be able to handle any context size, we could define a packet length (e.g. 32 bits). The packet are sent one per clock cycle on a signal whose size matches the packet. The context switch mechanism is then internal to the component. Reading a context from a component behaves the same, the component sending packets one after the other.

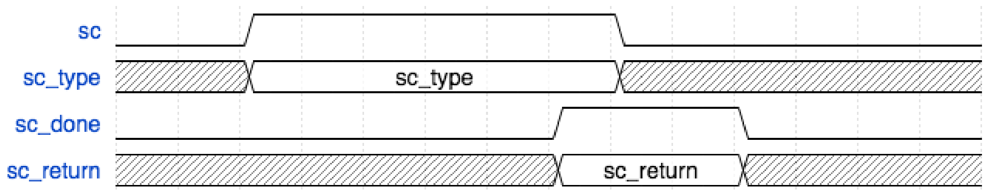


Figure 4.14: Wave of a cycle of Step

4.3.4 Multi-bus

As all we have seen before, there are three different buses. The control bus synchronizes the simulation cycle, the structure change bus handles the SC functions calls and acts on the components to change their and a communication bus which passe the message between components. The complete interface is defined in figure 4.15.

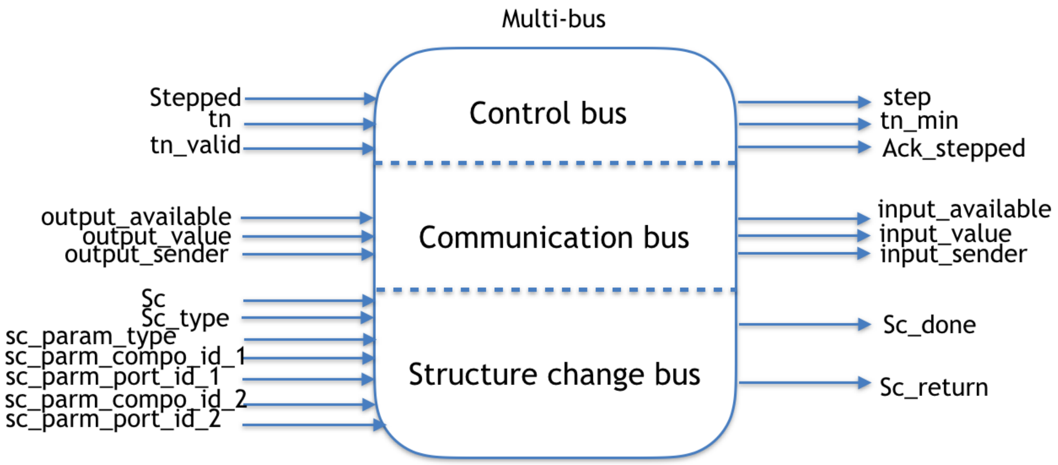


Figure 4.15: Multi-bus structure

4.4 Atomic component behavior specification

An atomic component begins its computations when it receives the `step` event, and provides a `stepped` event when over. During one cycle of simulation, the behavior of the component depends on its current DEVS state and elements associated to the current state. On top of states, the other elements of a

DEVS machine represent actions: internal transitions, external transitions, output emission and SC calls.

In our PDM, we chose to implement the behavior of a component doing a simulation step using Finite State Machines (FSM). We then have two levels of state machines: the DEVS state machines which represent an atomic component model phase, and the low-level Finite State Machines which apply this model to the simulation level.

4.4.1 FSM general structure

The FSM must store internally at least two variables: the current DEVS phase *Ds* and the next event time *tn_i*.

Regarding the FSM structure, an initial state *wait_step* represents the beginning of the FSM, as well as the idle state when waiting for a simulation cycle start. It passes to *Begin_step* when a *step* event is received. The state machine then must act according to the current *Ds*, elements associated with *Ds*, the current *tn_i* and the inputs received. To do so, we define one FSM branch for each possible *Ds*, which we build depending on the *Ds* elements. When the FSM finishes its treatment, all branches are gathered on an *End_step* FSM state, which emits the stepped event and waits for the ack_stepped reply from the coordinator to return to *wait_step*.

The figure 4.16 details this general FSM structure.

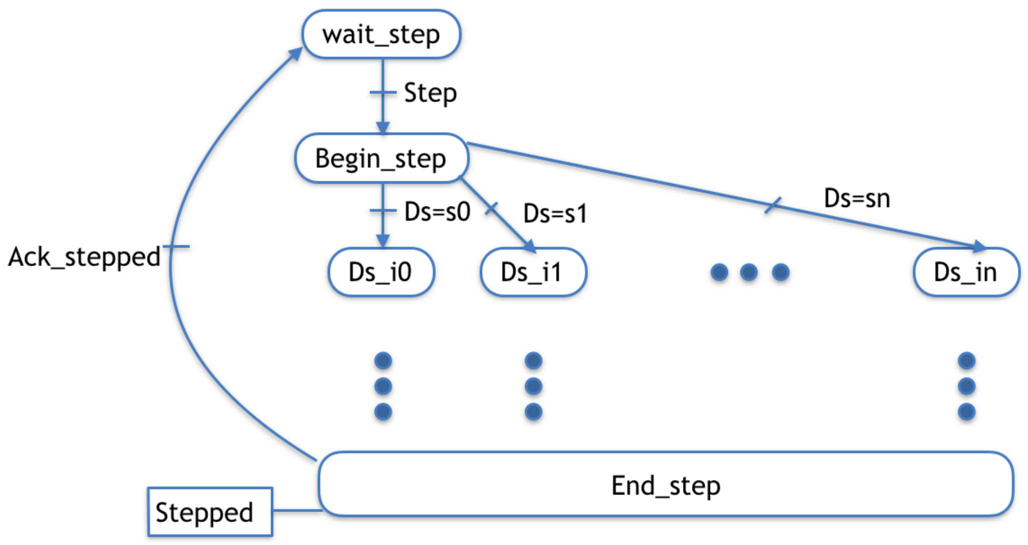


Figure 4.16: General state machine

4.4.2 FSM representation of internal transitions

An internal transition does not depend on input messages, it only depends on the component being imminent, i.e. simulation time equals to current state next event time.

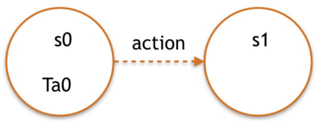


Figure 4.17: DEVS model of an internal transition

To represent a DEVS state containing an outgoing internal transition such as S_0 on figure 4.17, a comparison of local next event time tn_i and next event time sent by coordinator tn must be made. We represent this option by two FSM transitions with guards depending on the equality between these two times.

If $tn\_i = tn$ the internal transition must be crossed, which changes the current DEVS state. Moreover, as a result of DEVS state changing, the time of next event must also be updated to match the one of the new DEVS state. The *State_change()* function on the figures stands for the update of both Ds and tn_i accordingly. After what, the simulation step ends by moving the FSM to End_step. If $tn\_i \neq tn$, nothing has to be made and we directly move to End_step.

The figure 4.18 represents the FSM branch representing the above figure DEVS machine.

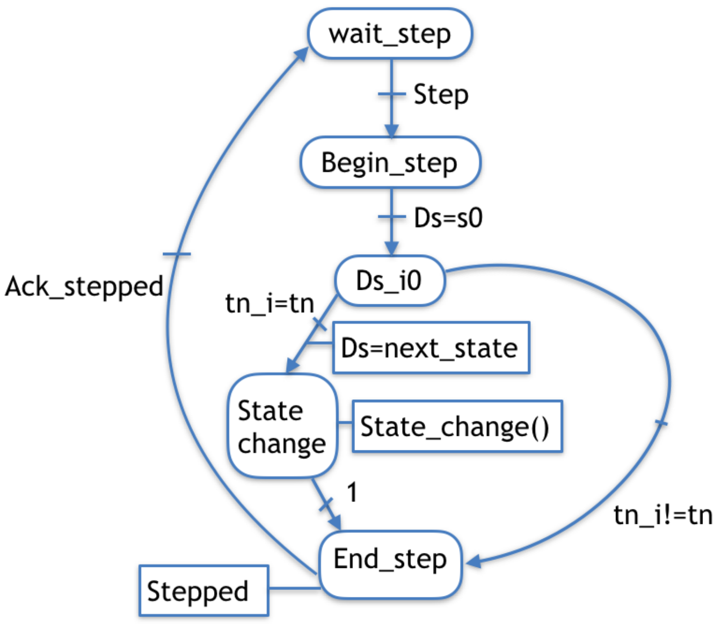


Figure 4.18: Internal transition

4.4.3 FSM representation of external transitions

An external transition depends on input messages, actions take place when an input is detected. As an example, in figure 4.19, the state will change from S_1 to S_2 when X is received.

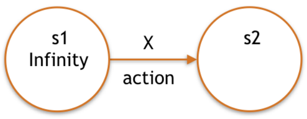


Figure 4.19: DEVS model of an external transition

To represent a DEVS phase containing an outgoing external transition, a verification of input must be made. We represent this option by two FSM transitions with guards checking if an input is available in the communication block. The input_available signal from the communication block indicates that a message is available.

If an input is available, the external transition must be crossed, which changes the current DEVS state. The *input_read* event is emitted, which is used by the input manager to clear its *input_available* flag. After what, the simulation step ends by moving the FSM to End_step. If no input is detected, nothing has to be done and we directly move to End_step.

The figure 4.20 represents the FSM branch representing the above figure DEVS machine.

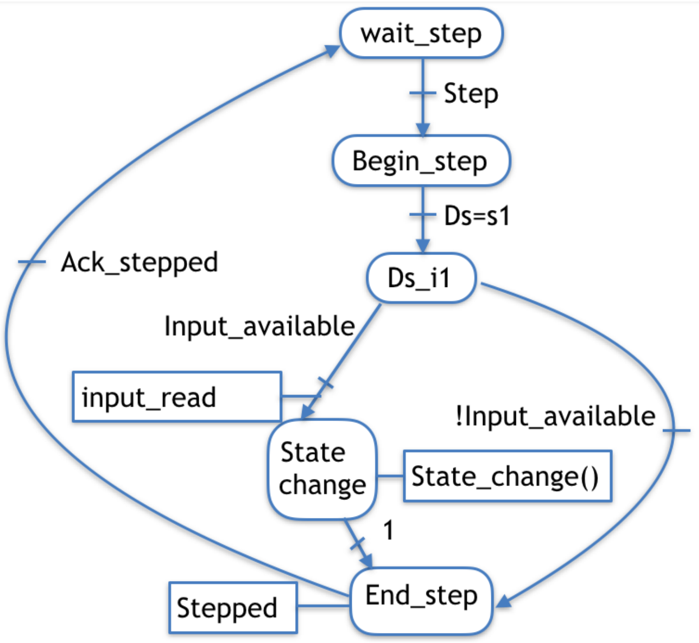


Figure 4.20: External transition

If multiple inputs exist in the component, there are as many input manager signals as inputs.

4.4.4 FSM representation of output emission

An output emission (λ) generates an output message if the DEVS component is imminent. The output emission does not change the DEVS phase, so an internal transition is needed if it needs to be updated.

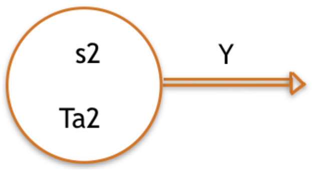


Figure 4.21: DEVS model of an output emission

To represent a DEVS phase containing an output, such as in figure 4.21, a verification of imminence must be made, as for internal transitions. The guards on the FSM transitions are then equivalent to the ones of internal DEVS transitions.

An output event is sent, together with output_available signal. If multiple outputs exist, the multiplexing of the outputs and output request to the bus are handled by the communication block.

When output_written is received, the simulation step ends by moving the FSM to End_step. If the component is not imminent, nothing has to be made and we directly move to End_step.

The figure 4.22 represents the FSM branch representing the above figure DEVS machine.

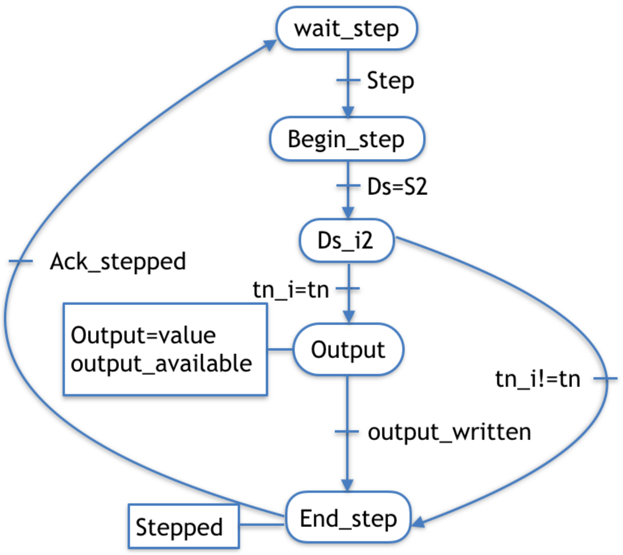


Figure 4.22: Output emission

4.4.5 Conflict

In a single DEVS phase, it is possible to have several simultaneous actions with no conflict. e.g. an internal transition together with an output emission

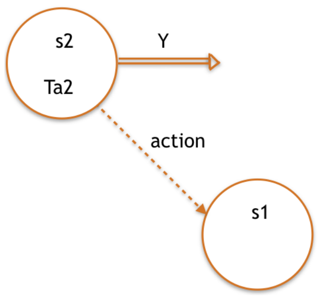


Figure 4.23: Non-conflict transition

where one is in charge of changing the state and the other to manipulate the output, as shown in figure 4.23. In this case, we simply combine the two branches so that the output is emitted first, and then the state is changed, as required by Zeigler's algorithms.

However, in certain cases, it is called conflict where we must choose one action and ignore the others. e.g. when an internal transition is together

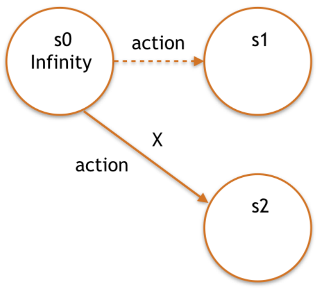


Figure 4.24: Conflict transition

with external transition and both are active, we must ignore one of them, as presented in figure 4.24. In such a case, DEVS provides the δ_{con} resolution function.

In our case, we set either $\delta_{con} = \delta_{ext}$ or $\delta_{con} = \delta_{int}$, as chosen by the modeler. In future work, the δ_{con} mechanism can be enhanced to propose various other choices.

4.5 SC functions implementation

All SC functions will have the same FSM structure. Here is a state machine of addComponent structure change function, as shown in 4.25.

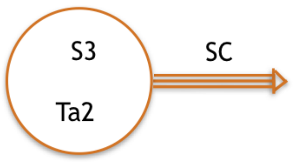


Figure 4.25: DEVS model of a structure change emission

Dynamic transitions start from component. a component with dynamic transitions adds several input/output ports concerned to transition: $Sc \in \{0, 1\}$ indicates if there is a SC call to the coordinator. Sc_type is the signal specifying which dynamic function is called. The parameters depend on which SC function is called. For example, for the addComponent function, in this simplified PRDEVS implementation, only the component type is required as a parameter.

Figure 4.26 presents the call of an addComponent SC function.

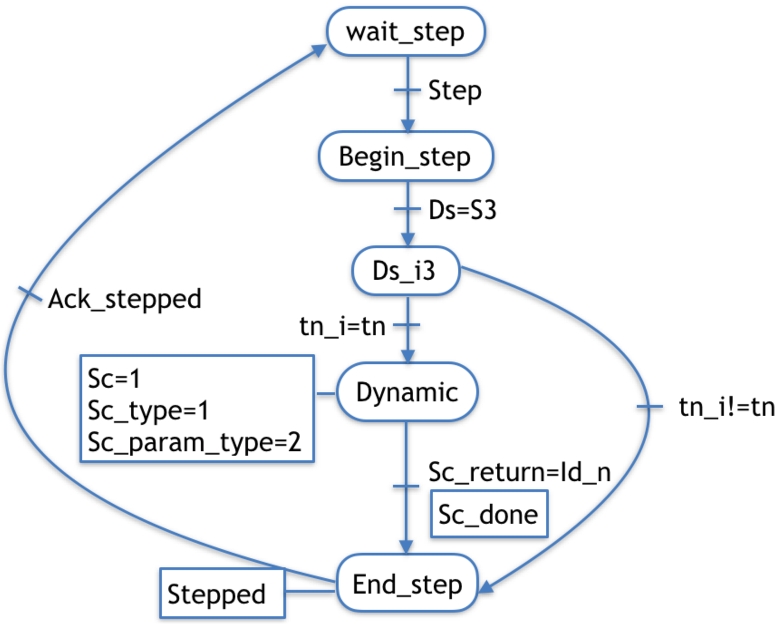


Figure 4.26: Dynamic transition: addComponent

Inside the coordinator, there is a static part and a dynamic part, as depicted on figure 4.27.

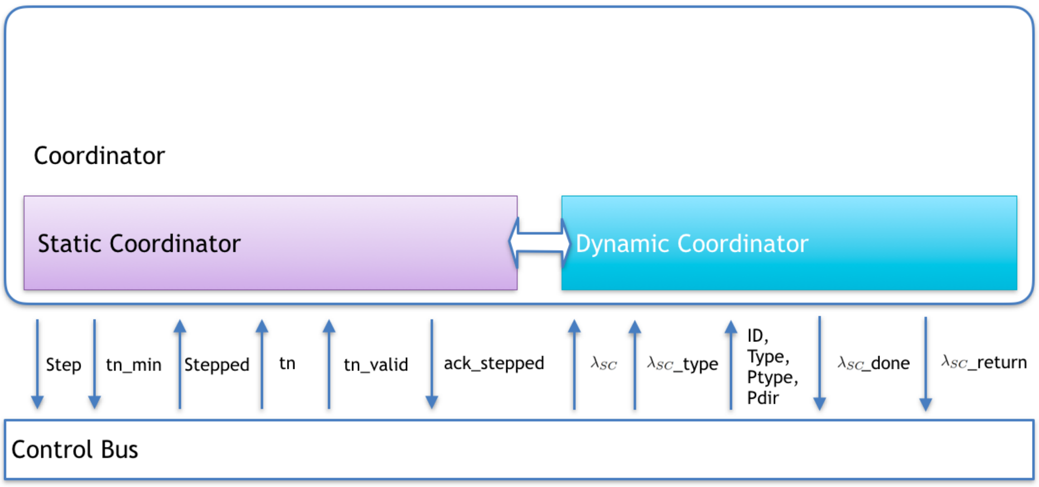


Figure 4.27: Coordinator

4.5.1 Static coordinator

The static part is in charge of the scheduling of the simulation. It first emits a `step` event to the control bus. Then, it waits for a component to send its `stepped` event. When the coordinator receives this event, it stores the `tn` and the `tn_valid` values in a table. After all the components have sent their cycle ending event, the coordinator iterates on the table to find the `tn_min`. Then, a new cycle begins.

4.5.2 Dynamic coordinator

The dynamic coordinator has to handle the SC function calls, and emits the signals in charge of changing the configuration of the components. In our implementation, we simplified the available SC calls to restrict them to these four functions: addComponent, removeComponent, addConnection and removeConnection. We can separate the connection management from the component management.

Connection management In our PDM, we do not handle modeling errors. Thus, the connection manager does not require any table in the coordinator to memorize existing connections. When a SC message concerning a connection change is received, it is broadcasted "as is" to the other components. The values indicated on the parameters of the SC function change are simply copied to the SC connection interface.

Component management To handle component management, the dynamic coordinator must be able to add and remove components. In hardware, adding a component corresponds to the action of reconfiguring a reconfigurable area using a partial bitstream file. Removing a component is also a reconfiguration, but using a "blank" bitstream. Each type of component requires a specific bitstream file. But, as each reconfigurable area has a different position on the FPGA, the same component placed in two different places requires two different bitstreams. Thus, we need as many bitstreams as there are components times reconfigurable areas, plus a blank bitstream for each reconfigurable area, as presented in table 4.1

From the bitstream identifier, the dynamic coordinator is able to trigger a reconfiguration in the Partial Reconfiguration Controller, as depicted in Section 1.3.2.

Bitstream relocation technics exist, consisting in modifying a bitstream in order to change the resources coordinates. It can be applied when two different reconfigurable areas are exactly the same except for the position. We will not use these techniques as they constitute an optimization that is out of the scope of this study.

Another table is required to memorize the current status of the reconfigurable areas. The table links each area with the type of component currently

Reconfigurable area	Component type	Bitstream ID
1	generator1	12
2	generator1	13
1	generator2	21
2	generator2	22
1	counter	34
2	counter	31
1	blank	01
2	blank	02

Table 4.1: Example of bitstream memory table

inside, and the identifier of the component. An example of such a table is given in table 4.2.

From this table, the dynamic coordinator is able to retrieve the number of components currently existing in the model. This value is provided to the static coordinator which requires it to determine that all components have ended their cycle.

Reconfigurable area	Component type	Component ID
1	generator1	gen1
2	counter	count1

Table 4.2: Example of occupation memory table

The removeComponent call uses the second table to determine the area in which resides the component to be deleted. Applying a removeComponent consists in applying a reconfiguration with a blank bitstream in the given area, and updating the table.

4.6 Example

In order to detail the structure of the PRDEVS meta-model, we give an example of a dynamic structure model, tested under Xinlix Vivado and FPGA.

It is a generator-counter model, similar to the example in Chapter 1. A generator sends a coin to the counter following its own state machine. the counter counts all income coins and change the generator every ten units of time. In this example, the structure change functions as addComponent, addConnection and removeComponent, removeConnenctions are tested. However, the dynamic context is not present in the model due to the lack of time.

With two different generators and one counter in the library L, the C^{Top} model the numbers and the connections of generator to counter are changable.

The difference between two senders is: $generator_1$ sends a coin to counter every 2 units time and $generator_2$ sends a task to receiver every 3 unit time. $counter_1$ receives coins and counting the total numbers of coins received.

Root model Under the root model there are four types of components under L and currently only two components under C^{Top}.

$ModelExample =< L, C^{Top} >$ where

$$L = \{C^{generator_1}, C^{generator_2}, C^{counter_1}\}$$

$$C^{Top} = \{X^{Top}, Y^{Top}, D^{Top}, EIC^{Top}, EOC^{Top}, IC^{Top}\}$$

with

$$X^{Top} = \{\}$$

$$Y^{Top} = \{\}$$

$$D^{Top} = \{C_{generator_1}, C_{counter_1}\}$$

$$EIC^{Top} = \{\}$$

$$EOC^{Top} = \{\}$$

$$IC^{Top} = \{((C^{generator_1}, event), (C^{counter_1}, event))\}$$

Atomic component - generator The generator is static which can be discribed as:

$M^1_{generator} =< X, Y, S, s_0, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda_{SC}, \lambda, \tau >$ with

$$X = \{\}$$

$$Y = \{P^{EVENT}\}, \text{ with } P^{EVENT} = (out, \mathbb{B})$$

$$S = \{s_0\}$$

$$\delta_{int} = \{s_0\}$$

$$\lambda(s_0) = (P_{EVENT} \leftarrow \mathsf{TRUE})$$

$$\tau = 2$$

$$\delta_{ext} = \delta_{con} = \lambda_{SC} = \{\}$$

Atomic component - counter The counter is dynamic which can be discribed as:

$M^2_{counter} =< X, Y, S, s_0, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda_{SC}, \lambda, \tau >$ with

$X = \{P^{EVENT}\}$, with $P^{EVENT} = (in, \mathbb{B})$

$Y = \{\}$

$S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9\} \times ID \times ID \times ID \times ID \times \mathbb{N}$
repectively: s, senderID, portID, recevierID, portID, count

$s_0 = (s_0, \varnothing, \varnothing, \varnothing, \varnothing, 0)$

$$\delta_{ext}((s, count), in) = (s_1, count + 1) \text{ if } s = s_0 \wedge in \rightarrow \mathsf{TRUE}$$

$$\delta_{int}(s, count) = \begin{cases} s_0 & \text{if} \quad s = s_1 \;\wedge [count \neq 10 \& count \neq 20]) \; or \; (s = s_5) \; or \; (s = s_9) \\ s_2 & \text{if} \quad s = s_1 \;\wedge [count = 10] \\ s_3 & \text{if} \quad s = s_2 \\ s_4 & \text{if} \quad s = s_3 \\ s_5 & \text{if} \quad s = s_4 \\ s_6 & \text{if} \quad s = s_1 \;\wedge [count = 20] \\ s_7 & \text{if} \quad s = s_6 \\ s_8 & \text{if} \quad s = s_7 \\ s_9 & \text{if} \quad s = s_8 \end{cases}$$

$$\delta_{con}(s) = \{\}, \forall s$$

$$\lambda_{SC}(s) = \begin{cases} removeConnection(gen1, P^{EVENT}, cpt, P^{EVENT}) & \text{if} \quad s = s_2 \\ removeComponent(gen1) & \text{if} \quad s = s_3 \\ addComponent(gen2) & \text{if} \quad s = s_4 \\ addConnection(gen2, P^{EVENT}, cpt, P^{EVENT}) & \text{if} \quad s = s_5 \\ removeConnection(gen2, P^{EVENT}, cpt, P^{EVENT}) & \text{if} \quad s = s_6 \\ removeComponent(gen2) & \text{if} \quad s = s_7 \\ addComponent(gen1) & \text{if} \quad s = s_8 \\ addConnection(gen1, P^{EVENT}, cpt, P^{EVENT}) & \text{if} \quad s = s_9 \\ \{\} & else \end{cases}$$

$$\tau(s) = \begin{cases} \infty & \text{for} \quad s == s_0 \\ 0 & \text{for} \quad else \end{cases}$$

Graphic representation - Top level Here is a graphic representation of this example. At C^{TOP} there are two components in the beginning, as shown in 4.28.

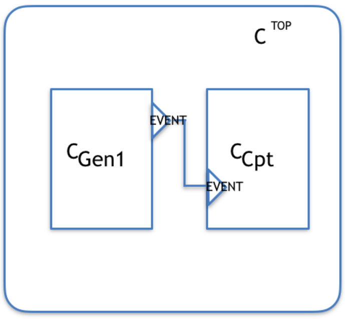


Figure 4.28: Graphic presentation of the example

Graphic representation - generator atomic In generator there is only one state. It is static, as shown in 4.29.

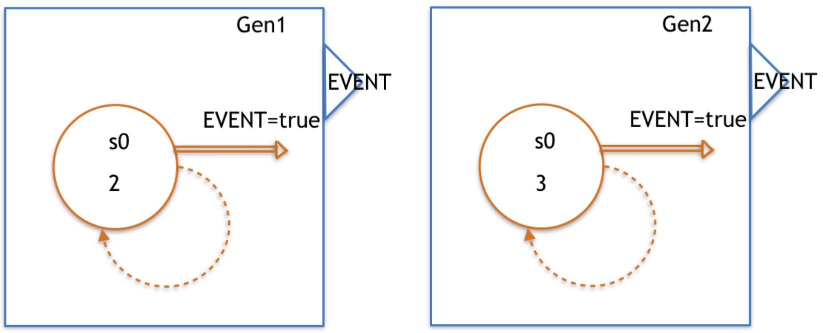


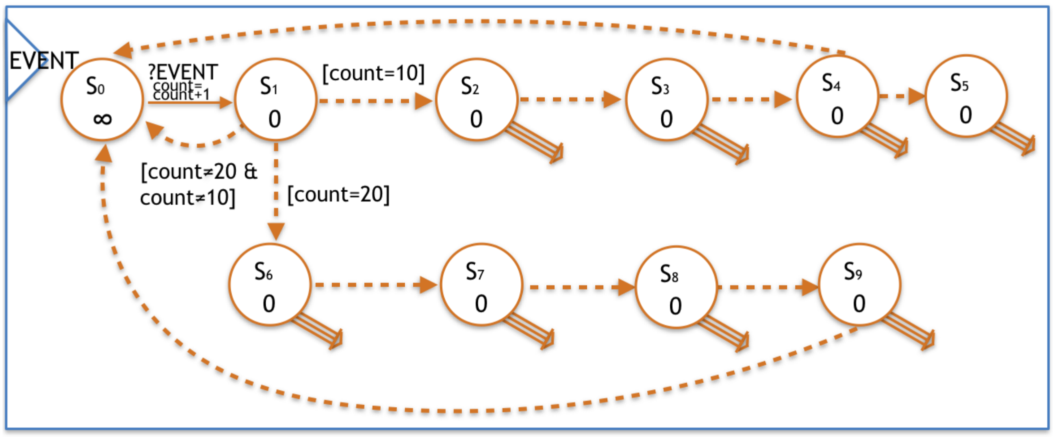
Figure 4.29: Graphic presentation of the generator

Graphic representation - counter atomic In counter there are ten states, as shown in 4.30

4.7 Conclusion

In this chapter, we have built a hardware based PDM, cooperate with PIM, forming a PSM. The PDM hardware is different from the PDM software definition. The multi-bus are detailed for the communication, control and structure change. The component behaviors are specified by the state machines. The SC functions are implemented by two coordinators.

We applied restrictions over the PRDEVS formalism, in order to ease the deployment. The main one is about the model hierarchy, which is not supported by the implementation. This leads to slightly change the syntax of the SC functions, which do not require the coupled component identifier



s2: removeConnection(id,Event,cpt,Event)
s3: removeComponent(id)
s4: id = addComponent(gen2)
s5: addConnection(id,Event,cpt,Event)

s6: removeConnection(id ,Event,cpt,Event)
s7: removeComponent(id)
s8: id = addComponent(gen1)
s9: addConnection(id,Event,cpt,Event)

Figure 4.30: Graphic presentation of the counter

in which the new component have to be added. This was done to reflect the flat nature of FPGA partially reconfigurable areas, which doesn't support reconfiguration hierarchy, at least within the standard tools.

It could be however possible to work around this restriction in future works by respecting the hierarchy within the coordinator representation of the model. The mapping to the flat design could then be transparent for the model itself, the translation being handled seamlessly by the coordinator.

Conclusion

In this thesis, we were willing to deal with the model description for dynamic structure systems. In order to formally describe a dynamic structure model and be able to deploy it to several simulation platforms, the system engineering approach has been chosen. We thus defined a meta-model suitable for dynamic model representation and an execution semantics for its simulation. We chose to base our meta-model on DEVS, which is a strong mathematical foundation. We called this new formalism Partially Reconfigurable Discrete Event System Specification (PRDEVS). PRDEVS is thus a discrete-event based formalism designed to model dynamic structure systems following the model-driven architecture approach.

Before PRDEVS was defined, several preliminary steps have been performed.

First, the system engineering approach was investigated. The system engineering approach adds traceability, especially with the Model-Based System Engineering (MBSE) process. MBSE reduces the text-based documents and improves the understanding between engineers. It also provides a life cycle description from operational models to component models. In this approach, models with different abstraction levels can coexist within the same virtual system. The different levels of the model make it possible to edit separately each layer.

In this thesis, we looked into the Model-Driven Architecture (MDA) process, which is a system engineering approach focused on model transformation targetting different execution platforms. The Model-Driven Architecture proposes an integration architecture based on three levels of models. A platform-independent model focuses on the model functionality, while the platform description model focuses on the application platform. These two models are then integrated into a platform-specific model. Compared to other approaches, MDA is suitable for our objective where the dynamic model must be compatible with different execution platforms.

MBSE is then a design methodology and can be combined with discrete-event systems. Discrete Event System Specification (DEVS) is a suitable specification for use with our aim of hierarchical, modular, discrete-event modeling. However, DEVS has restrictions when it comes to supporting dynamic structure modeling. There are several extensions trying to expand the capability of the models to support dynamic structure. DSDEVS is one of these extensions. But the formalism relies on a mathematical description which assumes a predefined set of structures. A proposed simulator then provides dynamic functions to ease the model manipulations. This way of doing requires the implementation of the meta-model to fill the lacks of the formalism by proposing helper functions. Moreover, the context management is not handled by the formalism. Notably, DSDEVS does not allow the original state of a newly added component to be different from the initial component state.

Another DEVS extension targetting dynamic structure systems is DynDEVS. It proposes a network transition function extended to atomic components. Where DSDEVS can only replace the coupled structure, DynDEVS allows for atomics to change their inner behavior. However, the remarks we made for DSDEVS are still valid for DynDEVS in terms of atomic state management, even if the atomic component reconfiguration allows for more adaptability.

Finally, RecDEVS considers dynamic hardware early in the model architecture without the approach of system engineering. In RecDEVS, a model must consider low-level notions such as address even at the highest abstraction level. This formalism specifically targets hardware, and it may be difficult to adapt a RecDEVS model to a software execution environment.

On top of all these formalism considerations, we also want to address the simulation platform itself. Especially, we want to allow targetting reconfigurable hardware platforms. Indeed, these platforms, notably FPGAs, allow for dedicated circuitry for specific applications, which develops more efficient computations that software processors. They still remain more flexible than ASICs, which are designed for a single purpose and do not provide dynamic capabilities. The programmable logic device makes it possible to change the configuration on the hardware to a different logic, which can lead to different functions and can even repurpose specific areas of the chip on-the-fly.

With these considerations, we think that a formalism for supporting dynamic structure models and integrating FPGAs as a target platform does not exist yet. Thus, we chose to build our own architecture having in mind the MDA approach, the DEVS background, the support for dynamic hardware and discrete-event simulation.

Contributions

The first contribution that we made was the PRDEVS meta-model. PRDEVS is a DEVS-based formalism which can manage dynamic structure changes. Unlike DSDEVS and DynDEVS, PRDEVS does not presuppose reachable structural states and the dynamic functions are defined as the same level of other actions, directly within the formalism.

PRDEVS keeps PDEVS atomic and coupled components structure as a base. However, it adds a new type of outputs, λ_{SC}, dedicated to structural change. A component can issue dynamic structure functions calls which are handled by a coordinator. The λ_{SC} functions include the component management, connection management and context management.

A PRDEVS model has an initial structure C^{Top} which contains, as for PDEVS, coupled components and atomic components in an initial state. But along with this structure comes two libraries: a component library (L) and a context library (L_Φ). At any time in the simulation, components can request insertion or removal of a component. Adding a component requires referring to a component $C_t \in L$, which will be added into a coupled belonging to C^{Top}. All components in a PRDEVS have a type, which identifies a unique behavior. A type can be shared by multiple components, but all components of a type do not require to be in the same state. Each component in C^{Top} is uniquely identified by an identifier, which is used as a label to apply dynamic functions on it.

The second level of dynamic capability targets the connections between components ports. PRDEVS allows for removing or adding connections between the components, targetting the DEVS connections IC, IOC and EOC sets. Moreover, in order to account for hierarchy evolution, coupled components can see their set of ports changed. Unlike ρ-DEVS, which is an extension of DynDEVS, atomic components have a static set of ports, as we do not address atomic component behavior reconfiguration. We chose to not support this behavior and to focus on structure and context changes.

Finally, PRDEVS takes into consideration the context of atomic components. A context consists of all the state variables within a component. It can be saved from a running component, and be added to L_Φ in order to be reused later. A context can also be read from L_Φ and applied to a component with a matching type. This capability allows for scheduling management, such as removing a component at a point and creating it anew later in the exact same state. Moreover, the context library can contain elements at the beginning of a simulation. This allows for using previously defined initial contexts for different components.

In all our definitions, to account for the MDA approach, we do not consider the execution platform. This allows for building platform-independent models, and then match them with a simulation platform. A single model can then be applied to different computing architecture, such as software and reconfigurable hardware.

The second contribution that we proposed in this document in a software-based execution platform targetting the PRDEVS meta-model. We applied an object-oriented view on the software programming, defining classes to represent the different elements of the formalism. We defined atomic components and coupled components using an UML representation. As they have several common characteristics, such as ports and identifications, we use the notion of inheritance to model it. The atomic component class contains states and functions, such as internal transitions, external transitions, output emissions and dynamic functions call. The coupled component class keeps the information on the connections, such as the *IC*, *IOC* and *EOC* sets. They are respectively managed by a simulator and a coordinator.

We do not change the semantics of time in a simulator from Zeigler's definition. The difference that we made from other simulators is that the dynamic functions calls take priority over other messages. A case study of this software platform specific model has been presented, proving the feasibility to the PRDEVS meta-model integration into a software platform. However, we did not extend our tests to a hierarchy beyond a single level. Nevertheless, as we were based on Zeigler's semantics, and the nesting of software objects is a very common process, using this implementation with hierarchical models would be possible. The only requirement for doing so is a function which is able to locate a precise coordinator in the hierarchic tree, the *getExistingComponent()* function defined in the PRDEVS semantic.

The third contribution that we made is a hardware platform description model compatible with the PRDEVS formalism. Based on the PRDEVS meta-model, an integration from a platform-independant model to an FPGA specific development was presented.

The hardware platform is divided in reconfigurable areas, following the state of the art methodology for partially reconfigurable systems. These reconfigurable areas are identified by a unique reference. They do not have the same configurable resource layout, and thus a different potential for deploying a component. A look-up table establishing the correspondence between the component type and reconfigurable area on one hand, and the matching bitstream on the other hand is then required. Three interfaces have been built to connect the components. The control interface synchronizes the time in the simulator. The communication interface makes possible the inter-component communications. The structure change interface is in charge of the dynamic functions calls and acts on the components configuration.

State machines which represent the generic atomic behavior have been proposed to implement the DEVS component evolution in a hardware-friendly way. Along with the DEVS component integration, we proposed a coordinator, divided into a static part and a dynamic part, which manages the simulation from within the FPGA. Even if we did not provide a hardware-description language match, we proposed a systematic transformation to a state-machine representation, from which many existing tools are able to infer an implementation. In comparison, SynDEVS, which is oriented at hardware/software co-design, does not provide a precise way of generating a hardware representation. In our work, the platform specification for the PRDEVS meta-model is defined in detail, down to the multi-bus structure providing the inter-component communications and synchronization. As for RecDEVS, which proposes a precise architecture for the reconfigurable platform, they do not define the inner atomic components behavior.

This development based on PRDEVS proves the possibility to apply the model driven architecture approach to a DEVS-based formalism. However, as a lack of time, we still miss the context management in this implementation. This part does not represent a complicated work to carry out, but the multi-bus structure and component interfaces must be adapted to support the operation.

Future work and perspectives

For a full integration of the complete PRDEVS meta-model, there are several important points which could be dug into. There are functions of PRDEVS, such as context management, which have not been applied to the FPGA implementation yet. With the theory that we proposed in this thesis, it could be done with several extra tables. However, where to store these tables and by which ways the context is loaded should be defined. It is possible to add an interface to the components to collect the context information and send it to the coordinator. As for the internal behavior of an atomic component intended at managing the context operations, we can imagine that the context of all components is stored as an array. The variables can then be references on specific areas of this array for internal use, while the array can be read or written at once for context operations.

Also, handling the model hierarchy could be applied using a virtual coupled component. The actual, flat, implementation then would require a translation for structure operations applied to this virtual component. This could be done by the dynamic coordinator, which would contain this virtual hierarchy and apply the mapping each time a dynamic function call is triggered.

Since the PRDEVS meta-model can adapt to platforms with different description models, development to apply PRDEVS to another reconfigurable platform could also be considered. GPU, HPC or a co-simulation between hardware and software are good candidates for such an application.

The PRDEVS meta-model supports the parallelism as it is based on PDEVS. However, some parallel optimizations were not considered in this work. When we specify the model into the hardware platform, the bus structure limits the parallelism. Moreover, the dynamic structure calls suspend the other operations. In the future researches, the parallelism could be considered into the platform-specific model, or at the meta-model level.

Also, the model verification and transformation could be considered in the simulator. For the moment, the model verification must be done by the model constructor. An automatical model to simulation verification could be considered, either as part of the simulation platform or directly in the structure change functions semantic.

Bibliography

[Adamski 2005] Marian Andrzej Adamski, Andrei Karatkevich et Marek Wegrzyn. Design of embedded control systems, volume 267. Springer, 2005. (Cité en page 29.)

[Ambler 2004] Scott W Ambler. The object primer: Agile model-driven development with uml 2.0. Cambridge University Press, 2004. (Cité en page 10.)

[Aspray 1997] William Aspray. *The Intel 4004 microprocessor: What constituted invention?* IEEE Annals of the History of Computing, vol. 19, no. 3, pages 4–15, 1997. (Cité en page 29.)

[Balaji 2012] S Balaji et M Sundararajan Murugaiyan. *Waterfall vs. V-Model vs. Agile: A comparative study on SDLC.* International Journal of Information Technology and Business Management, vol. 2, no. 1, pages 26–30, 2012. (Cité en page 10.)

[Barros 1997a] Fernando J Barros. *Modeling Formalisms for Dynamic Structure Systems.* ACM Transactions on Modeling and Computer Simulation (TOMACS), vol. 7, no. 4, pages 501–515, Octobre 1997. (Cité en page 18.)

[Barros 1997b] F.J. Barros et B.P. Zeigler. *Adaptive Queueing: A Study Using Dynamic Structure DEVS.* International Transactions in Operational Research, vol. 4, no. 2, pages 87–98, 1997. (Cité en page 19.)

[Barros 1998] Fernando J Barros. *Abstract simulators for the DSDE formalism.* In 1998 Winter Simulation Conference. Proceedings (Cat. No.98CH36274), volume 1, pages 407–412. IEEE, Dec 1998. (Cité en pages 18, 20, 21 et 44.)

[Bergero 2011] Federico Bergero et Ernesto Kofman. *PowerDEVS: a tool for hybrid system modeling and real-time simulation.* Simulation, vol. 87, no. 1-2, pages 113–132, 2011. (Cité en page 54.)

[Boehm 1988] Barry W. Boehm. *A spiral model of software development and enhancement.* Computer, vol. 21, no. 5, pages 61–72, 1988. (Cité en page 10.)

[Dalrymple 1999] Mary Dalrymple. Semantics and syntax in lexical functional grammar: The resource logic approach. MIT Press, 1999. (Cité en page 13.)

[DeHon 1999] André DeHon. *Balancing interconnect and computation in a reconfigurable computing array (or, why you don't really want 100% LUT utilization).* In Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays, pages 69–78. ACM, 1999. (Cité en page 74.)

[Deleganes 2002] Daniel Deleganes, Jonathan Douglas, Badari Kommandur et Marek Patyra. *Designing a 3 GHz, 130 nm, Intel®Pentium®4 processor.* In VLSI Circuits Digest of Technical Papers, 2002. Symposium on, pages 130–133. IEEE, 2002. (Cité en page 29.)

[Eickhoff 2009] Jens Eickhoff. Simulating spacecraft systems. Springer Science & Business Media, 2009. (Cité en page 11.)

[Estefan 2007] Jeff A Estefan*et al. Survey of model-based systems engineering (MBSE) methodologies.* Incose MBSE Focus Group, vol. 25, no. 8, pages 1–12, 2007. (Cité en page 10.)

[Ferber 1999] Jacques Ferber. Multi-agent systems: an introduction to distributed artificial intelligence, volume 1. Addison-Wesley Reading, 1999. (Cité en page 67.)

[Franceschini 2014a] Romain Franceschini, Paul-Antoine Bisgambiglia, Paul Bisgambiglia et David Hill. *DEVS-ruby: a domain specific language for DEVS modeling and simulation (WIP).* In Proceedings of the Symposium on Theory of Modeling & Simulation-DEVS Integrative, page 15. Society for Computer Simulation International, 2014. (Cité en page 54.)

[Franceschini 2014b] Romain Franceschini, Paul-Antoine Bisgambiglia, Luc Touraille, Paul Bisgambiglia et David Hill. *A survey of modelling and simulation software frameworks using Discrete Event System Specification.* In OASIcs-OpenAccess Series in Informatics, volume 43. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014. (Cité en page 54.)

[Friedenthal 2007] Sanford Friedenthal, Regina Griego et Mark Sampson. *INCOSE model based systems engineering (MBSE) initiative.* In INCOSE 2007 Symposium, 2007. (Cité en page 10.)

[Gardelli 2009] Luca Gardelli, Mirko Viroli et Andrea Omicini. *Combining simulation and formal tools for developing self-organizing MAS.* Multi-Agent Systems: Simulation and Applications, Computational Analysis, Synthesis, and Design of Dynamic Systems, pages 133–165, 2009. (Cité en page 67.)

[Harel 2000] David Harel et Bernhard Rumpe. *Modeling languages: Syntax, semantics and all that stu*. Rapport technique, Technical report, 2000. (Cité en page 13.)

[Hauck 2010] Scott Hauck et Andre DeHon. Reconfigurable computing: the theory and practice of fpga-based computation, volume 1. Morgan Kaufmann, 2010. (Cité en page 31.)

[Himmelspach 2009] Jan Himmelspach et Adelinde M Uhrmacher. *The JAMES II framework for modeling and simulation*. In High Performance Computational Systems Biology, 2009. HIBI'09. International Workshop on, pages 101–102. IEEE, 2009. (Cité en page 54.)

[Hu 2008] X Xiaolin Hu et Bernard P Zeigler. *The Architecture of GenDevs: Distributed Simulation in DEVSJAVA*, 2008. (Cité en page 54.)

[Iannucci 1988] Robert A Iannucci. Toward a dataflow/von neumann hybrid architecture, volume 16. IEEE Computer Society Press, 1988. (Cité en page 29.)

[Iyengar 2005] Sridhar Srinivasa Iyengar. *Metadata driven system for effecting extensible data interchange based on universal modeling language (UML), meta object facility (MOF) and extensible markup language (XML) standards*, Mars 29 2005. US Patent 6,874,146. (Cité en page 13.)

[Kesting 2008] Arne Kesting, Martin Treiber et Dirk Helbing. *Agents for traffic simulation*. arXiv preprint arXiv:0805.0300, 2008. (Cité en page 67.)

[Kim 2009] Sungung Kim, Hessam S Sarjoughian et Vignesh Elamvazhuthi. *DEVS-suite: a simulator supporting visual experimentation design and behavior monitoring*. In Proceedings of the 2009 Spring Simulation Multiconference, page 161. Society for Computer Simulation International, 2009. (Cité en page 54.)

[Kleppe 2003] Anneke G Kleppe, Jos B Warmer et Wim Bast. Mda explained: the model driven architecture: practice and promise. Addison-Wesley Professional, 2003. (Cité en page 13.)

[Latella 1999] Diego Latella, Istvan Majzik et 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. (Cité en page 46.)

[Ligtenberg 2004] Arend Ligtenberg, Monica Wachowicz, Arnold K Bregt, Adrie Beulens et Dirk L Kettenis. *A design and application of a multi-agent system for simulation of multi-actor spatial planning.* Journal of environmental management, vol. 72, no. 1-2, pages 43–55, 2004. (Cité en page 67.)

[Lysaght 2006] Patrick Lysaght, Brandon Blodget, Jeff Mason, Jay Young et Brendan Bridgford. *Enhanced architectures, design methodologies and CAD tools for dynamic reconfiguration of Xilinx FPGAs.* In Field Programmable Logic and Applications, 2006. FPL'06. International Conference on, pages 1–6. IEEE, 2006. (Cité en page 30.)

[Madlener 2013] Felix Madlener. *A Model of Computation for Reconfigurable Systems.* PhD thesis, Technische Universität, Darmstadt, 2013. (Cité en page 27.)

[Maxfield 2004] Clive Maxfield. The design warrior's guide to fpgas: devices, tools and flows. Elsevier, 2004. (Cité en pages 30 et 31.)

[Miller 1985] James Edward Miller. *Semantics and syntax: Parallels and connections.* 1985. (Cité en page 13.)

[Mina 2016] J Mina, Z Flores, E López, A Pérez et J-H Calleja. *Processor-in-the-loop and hardware-in-the-loop simulation of electric systems based in FPGA.* In Power Electronics (CIEP), 2016 13th International Conference on, pages 172–177. IEEE, 2016. (Cité en page 12.)

[Molter 2012] H Gregor Molter. Syndevs co-design flow: A hardware/software co-design flow based on the discrete event system specification model of computation. Springer Science & Business Media, 2012. (Cité en page 26.)

[Narendra 1990] Kumpati S Narendra et Kannan Parthasarathy. *Identification and control of dynamical systems using neural networks.* IEEE Transactions on neural networks, vol. 1, no. 1, pages 4–27, 1990. (Cité en page 67.)

[Object Management Group 2016] Object Management Group. *MDA - The Architecture of Choice for a Changing World*, 2016. [Online; accessed 19-January-2017]. (Cité en page 12.)

[Platzner 2010] Marco Platzner et Norbert Wehn. Dynamically reconfigurable systems: Architectures, design methods and applications. Springer Science & Business Media, 2010. (Cité en page 31.)

[Poole 2003] John Poole, Dan Chang, Douglas Tolbert et David Mellor. Common warehouse metamodel developer's guide, volume 24. John Wiley & Sons, 2003. (Cité en page 13.)

[Ptolemaeus 2014] Claudius Ptolemaeus. System design, modeling, and simulation: using ptolemy ii, volume 1. Ptolemy. org Berkeley, 2014. (Cité en page 46.)

[Qu 2009] Zhihua Qu. Cooperative control of dynamical systems: applications to autonomous vehicles. Springer Science & Business Media, 2009. (Cité en page 67.)

[Russell 2016] Stuart J Russell et Peter Norvig. Artificial intelligence: a modern approach. Malaysia; Pearson Education Limited,, 2016. (Cité en page 67.)

[Trencansky 2005] Ivan Trencansky et Radovan Cervenka. *Agent Modeling Language (AML): A comprehensive approach to modeling MAS*. Informatica, vol. 29, no. 4, 2005. (Cité en page 67.)

[Uhrmacher 2001] Adelinde M Uhrmacher. *Dynamic Structures in Modeling and Simulation: A Reflective Approach*. ACM Transactions on Modeling and Computer Simulation (TOMACS), vol. 11, no. 2, pages 206–232, Avril 2001. (Cité en page 23.)

[Uhrmacher 2006] Adelinde M Uhrmacher, Jan Himmelspach, Mathias Rohl et Roland Ewald. *Introducing variable ports and multi-couplings for cell biological modeling in DEVS*. In Proceedings of the 2006 Winter Simulation Conference, pages 832–840. IEEE, 2006. (Cité en page 24.)

[Uhrmacher 2009] Adelinde M Uhrmacher et Danny Weyns. Multi-agent systems: Simulation and applications. CRC press, 2009. (Cité en page 67.)

[Van Tendeloo 2016] Yentl Van Tendeloo et Hans Vangheluwe. *An overview of PythonPDEVS*. In Collectif Workshop RED, editor, JDF, pages 59–66, 2016. (Cité en page 54.)

[Von Bertalanffy 1956] Ludwig Von Bertalanffy. *General System Theory*. General Systems, vol. 1, pages 1–10, 1956. (Cité en page 9.)

[Vu 2015] Le Hung Vu, Damien Foures et Vincent Albert. *ProDEVS: an event-driven modeling and simulation tool for hybrid systems using state diagrams*. In SimuTools, pages 29–37, 2015. (Cité en pages 13 et 54.)

[Wainer 2002] Gabriel Wainer. *CD++: a toolkit to develop DEVS models.* Software: Practice and Experience, vol. 32, no. 13, pages 1261–1306, 2002. (Cité en page 54.)

[Xilinx 2018] Xilinx. *Partial Reconfiguration in the Vivado Design Suite*, 2018. (Cité en page 32.)

[Xu 2005] Qiang Xu et Nicola Nicolici. *Resource-constrained system-on-a-chip test: a survey.* IEE Proceedings-Computers and Digital Techniques, vol. 152, no. 1, pages 67–81, 2005. (Cité en page 29.)

[Zeigler 2000] Bernard P Zeigler, Herbert Praehofer et Tag Gon Kim. Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems. Academic press, Orlando, FL, USA, 2nd édition, 2000. (Cité en pages 11, 14, 15, 17 et 19.)