Complex Cyber Physical Systems Co-Simulation With The CoSim20 Framework For Efficient And Accurate Distributed Co-Simulations

Giovanni Liboni

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE DE DOCTORAT

Co-simulation de systemes complexes avec le framework CoSim20

Pour des co-simulations distribuées performantes et fidèles

Giovanni LIBONI

Inria Sophia Antipolis – Méditerranée, Équipe Kairos
Safran Tech, Modélisation & Simulation

Présentée en vue de l'obtention du grade de docteur en Informatique **d'**Université Côte d'Azur
Dirigée par : Frédéric Mallet
Co-encadrée par : Julien Deantoni
Guy De Spiegeleer
Date de soutenance:
21 Avril 2021

Devant le jury, composé de :

Jean-Philippe Babau, Professeur, Université de Bretagne Occidentale – Rapporteur
Frédéric Boulanger, Professeur, Centrale Supélec - Examinateur
Bernard Coulette, Professeur, Université Toulouse Jean Jaurès - Rapporteur
Julien Deantoni, Maître de conférence – Université Côte d'Azur - Co-Encadrant
Virginie Galtier, Enseignante-chercheur , Centrale Supélec - Examinatrice
Frederic Mallet, Professeur, Université Côte d'Azur – Directeur de thèse
Maurice Theobald, Ingénieur chercheur, Safran Tech – Invité

Résumé

Les systèmes cyber-physiques sont des systèmes d'ingénierie complexe dans lesquels les parties computationnelles communiquent entre elles et avec les parties physiques décrivant l'environnement. Pour apprivoiser la complexité croissante de ces systèmes, ils sont généralement décomposés en différentes parties qui sont modélisées par différents experts, éventuellement issus de différentes organisations. Ces experts utilisent un langage adapté à leur problème, à la fois syntaxiquement et sémantiquement. À ce stade, d'une part on obtient des modèles exécutables des parties computationnelles et des modèles exécutables des parties physiques et, d'autre part, les modèles exécutables ne doivent pas violer les propriétés intellectuelles lorsqu'ils sont partagés et sont donc généralement partagés en tant qu'unités de simulation en boîte noire. Cependant, pour comprendre le comportement émergeant de l'ensemble du système, il est nécessaire de faire une simulation collaborative ; où les unités de simulation de différentes disciplines sont coordonnées pour échanger des données aux moments opportuns. Les problèmes avec les approches existantes sont multiples. Premièrement, les unités de simulation exposées sous forme de boîtes noires ne permettent pas de prendre en compte les spécificités sémantiques de chaque modèle. Deuxièmement, les co-simulations sont aujourd'hui principalement basées sur une interface de programmation dirigée par le temps, et il a été montré que de telles interfaces introduisaient, lorsqu'elles sont appliquées à des systèmes cyber-physiques, des «retards artificiels» dus à la co-simulation elle-même. De tels retards impliquent une mauvaise précision qui peut invalider les résultats de la co-simulation. De plus, réduire ces délais de co-simulation implique de mauvaises performances globales de co-simulation. Troisièmement, la définition d'un cadre de co-simulation précis et performant peut être complexe d'un point de vue algorithmique, de sorte qu'un support est nécessaire pour augmenter le niveau d'abstraction lors de la définition de la coordination. La thèse développée dans ce document est qu'il est possible 1) de fournir une interface de coordination de niveau modèle qui englobe à la fois les modèles cyber et les modèles physiques d'un système cyber-physique et 2) de définir un langage dédié à la coordination de tels modèles. Sur la base de ces artefacts et d'une interface de co-simulation originale, il est possible de générer automatiquement une infrastructure distribuée précise et performante pour la co-simulation. La thèse est soutenue par la mise en œuvre de deux langages dédiés. Un pour la définition de l'interface de coordination du modèle et un pour la définition de la spécification de la coordination. De plus, la thèse est également soutenue par un prototype d'API respectueux de la sémantique comportementale des modèles, qui est introduite comme une généralisation des API standard existantes. Enfin, un compilateur a été défini pour que les coordinations définies avec les langages proposés puissent être utilisées pour générer automatiquement un environnement de co-simulation distribué. Les différentes propositions sont appliquées sur une étude de cas où les avantages de l'approche sont clairement illustrés.

Mots-clés: co-simulation; sémantique comportementale; coordination; algorithme maître; modèles hétérogènes; Langages de coordination;

Abstract

Cyber-Physical Systems are complex engineered systems where computational parts communicate together and with physical parts describing the environment. To tame the growing complexity of such systems, they are usually decomposed into different parts that are modeled by different experts, possibly from different organizations. These experts are using a language tailored to their problem, both syntactically and semantically. At this point, on the one hand, it results in executable models of the cyber parts and executable models of the physical parts and, on the second hand, executable models should preserve intellectual properties and are usually shared as black-box simulation units. However, to understand the behavior emerging from the whole system, it is required to do a collaborative simulation; where the simulation units from different disciplines are coordinated to exchange data when appropriate.

The problems with the existing approaches are manifolds. First, simulation units exposed as black boxes do not allow taking into account the semantic specificities of each model. Second, co-simulations are nowadays mainly based on a time-driven application programming interface, which has been shown to introduce 'artificial delays' when applied to cyber-physical systems. Such delays imply a bad accuracy that may invalidate the co-simulation results. Moreover, reducing such co-simulation delays implies bad overall co-simulation performances. Third, the definition of an accurate and performant co-simulation framework may be algorithmically complex so that support is required to increase the abstraction level when defining the coordination.

The thesis developed in this document is that it is possible 1) to provide a model level coordination interface that encompasses both the cyber and the physical parts of a cyber-physical system and 2) to define a language dedicated to the coordination of such parts. Based on these artifacts and an original co-simulation programming interface it is possible to automatically generate an accurate and performant distributed infrastructure for co-simulation.

The thesis is supported by the implementation of two domain-specific languages. One for the definition of the Model Coordination Interface and one for the definition of the Model Coordination Specification. Additionally, the thesis is also supported by the prototype of a semantic-aware API introduced as a generalization of existing standard APIs. Finally, a compiler has been implemented so that coordinations defined with the proposed languages can be used to automatically generate a distributed co-simulation framework. The different proposals are applied to a case study where the advantages of the approach are clearly illustrated.

Keywords: co-simulation; behavioral semantics; coordination; master algorithm; heterogeneous models; coordination languages;

Contents

List of Figures

Introduction 1

1.1 Motivation

The increasing complexity of systems makes their design a highly challenging task that requires advanced engineering techniques to reason, develop, build, and maintain those systems. The fast 21-th century technological advancement and demand for more integrated and interconnected systems lead engineers to develop systems such as smart cities, networked medical devices, advanced automated aircraft systems, critical electrical power control, integrated defense systems. Those systems typically combine cyber capabilities (*e.g.* computation, control, and communication) with physical processes (*e.g.* electrical, mechanical, or chemical). For instance, the automatic pilot is a system that controls the trajectory of an aircraft, car, or vessel without constant control by a human operator. The vehicle moves physically into space following a trajectory that is determined by a computerized control algorithm that adjusts it using actuators(*e.g.* flaps) based on sensor readings of the physical space. We refer to these systems as Cyber-Physical Systems (CPS).

Since CPS integrates cyber and physical parts, we need to have knowledge of both in order to understand the emerging behavior of those systems. Their development requires advanced and deep knowledge in several domains, such as Software Engineering, Electrical Engineering, Mechanical Engineering, and Control engineering to cite a few of them. During the development, each domain requires more than one point-of-view and involves skills from different scientific and technical fields [1]. Nevertheless, it is not enough to have deep knowledge into a domain in isolation: we need to understand *how* the different domains work together, and *what* behaviors emerged *when* they interact and interface.

We need to represent those systems to understand their behavior, to predict their effects on the environment under their control, and to safely determine their behaviors. Given the critical nature of those systems, the increasing demanding complexity, stricter safety, and environmental regulations, and faster project development cycles, the development may be distributed across different societies, experts, stakeholders, and engineers using unambiguous representations. Formal specifications and representations permit to perform validation and verification tasks on each model and, more importantly, on the overall system.

The Model Driven Engineering (MDE) promotes the use of Domain-Specific Modeling Languages to develop complex CPS, using languages, tools, and methodologies tailored both syntactically and semantically to the domain of expertise of the user [2]. The use of those languages and tools ease the work of modeling, simulation, verification, and validation of parts of the system under a specific point-of-view. Therefore, using different languages and tools to develop different parts of the system leads to an *heterogeneous* development environment *i.e.* the system is made using different models that conforms to different languages. The CPS development can be considered an example of *heterogeneous* development environment due to the intrinsical heterogeneity of languages and tools due

to the CPS requirement to represent the different formalism and domains that composed those systems (*i.e.* physical and cyber domains).

In particular, the heterogeneous nature of CPS leads to the use of different formalisms representing the different parts of those systems. For instance, physical processes are defined by using a *Continuous Time* (CT) formalism to better represent the continuous nature of the physical mechanisms in nature where the time is represented as a continuous flow where between two points in time there is an infinite number of other points. On the contrary, the cyber processes must express the logic and algorithms used in a CPS and are better described using, for instance, a *Discrete Event* (DE) formalism where their execution is given by the sequence of triggered events.

The validation and verification of those systems are challenging tasks due to the heterogeneity of the formalisms, languages, and tools used in the development. One of the possible solutions to this problem is to use co-simulation. In a collaborative simulation or *co-simulation*, different loosely coupled and stand-alone simulation units conjointly simulate. A *simulation unit* (SU) is an executable entity that produces output data and consumes input over its execution. For instance, it can be a stand-alone model with its solver or simulator, a software with its interpreter, or a proxy to an existing part such as hardware- or software-in-the-loop. A simulation unit may conform to a black-box, which encapsulates algorithms, equations, and other workings into a model where only its inputs and subsequent outputs are known, ensuring that internal behaviors are not visible to external.

The black-box approach assumes an important role in those enterprises that require to collaborate with other companies. In an extended enterprise, the extensive use of electronic communications to exchange information with suppliers and vendors allows reducing the life cycle of material and information processing, product and infrastructure development, the required time to market given by the increasing competition and to create a more effective and efficient organization and systems [3]. An extended enterprise is then responsible for the whole life cycle of their products: from material procurement and supply chain management to production, manufacturing, product distribution, and customer service. Further, it is responsible for the disposal and recycling processes of end-of-life products. However, the development of a complex system based on the model-based approach requires sharing and exchange models to perform the integration among them, for validation and verification tasks. The black-box approach ensures Intellectual Property protection by hiding the sensitive knowledge in a non-readable format (*e.g.* a binary file).

To support a black-box co-simulation, the FMI standard [4], nowadays implemented by more than a hundred industrial tools, proposed to bundle, in a black-box manner, the simulation units using a homogeneous time-driven interface. Alternatively, the HLA standard [5] proposed the same idea but uses an event-driven interface. Based on one of these interfaces, the coordinator (also called Master Algorithm) is in charge of keeping time consistency and exchange data between the different models under execution. The development of such a coordinator usually relies on (1) the graph representing the sharing of data between the different models [6–9], and (2) the nature of the interaction between different models [10].

Several works have shown that neither a pure time-driven nor a pure event-driven approach can handle all model executions correctly [11–13]. Other studies have shown that the correctness of the co-simulation does not only rely on the

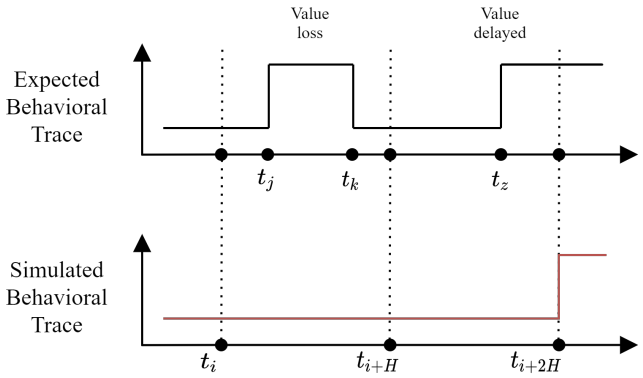


Figure 1.1: Temporal inaccuracy and delays introduced by using a Time-Triggered API on a piece-wise constant data.

correctness of each executable models but also depends on the coordinator [11, 14–16]. It is then the system engineer, who is in charge of writing the coordinator so that the co-simulation provides the actual system emerging behavior.

In this work, *we consider a coordination algorithm as correct if it does not introduce any delays or lose information during the communication with the simulation unit*. Consequently, delays and information loss that appear when using a time-triggered API on a piece-wise constant data are considered incorrect (see Figure 3.27). Three important things must be noticed at this point. First, sampling a piece-wise constant value can make sense and does not necessarily introduce a major problem; however, this should be done on purpose and not be the result of an inappropriate API. Second, there exists in many API (e.g., the FMI standard [4]) the possibility to avoid such delay, typically by roll-backing the simulation to a previous state and trying to locate the actual value change. This can be done only if the simulation can actually be rolled-backed; also this is costly in terms of simulation time. Finally, third, it is worth noticing that the problem is broader than the simple illustrative case. As illustrated in [17], the coordination algorithm can have an impact on the correctness of the simulation of the system.

The core of the problem was identified in several papers: it is not appropriate for any simulation unit to communicate only through a time-triggered or event-triggered API. In the literature, some approaches proposed to extend some existing API to fix a particular problem. This was for instance the case in [18] where they proposed to add a new parameter to the FMI time-triggered *doStep(*Δt*)* function. The new parameter is $nextEventTime$, a placeholder to store the time at which unpredictable events occurred. [11] went further by proposing to extend the FMI API with new functions that simulate until input and output ports are respectively ready to be read or just written. Finally, the new features of FMI 3.0 for hybrid co-simulation tries to aggregate such propositions (see chapter 5 of FMI 3.0 development version*).

However, in all these related works, the problem is not handled in its generality and they make specific cases of something that should be straightforward. In order to speak *correctly* with a simulation unit, you should be aware of its behavioral semantics and adapt the way to realize the *doStep* accordingly. As an abstraction of a simulation unit behavioral semantics, previous works proposed to focus on the nature of the inputs and outputs of the simulation units [8, 19] like for instance *continuous*, *piecewise-continuous*, *piecewise-constant* or *spurious*.

We then identify three main aspects that affect co-simulation:

* https://fmi-standard.org/docs/3.0-dev/#fmi-for-hybrid-co-simulation

1. the writing of the coordinator is more and more complex because it must take into account the characteristics of the data it conveys;
2. the characteristics of the models, the behavioral semantics of the used language, and their implementation by a solver;
3. the topology between the different interconnected executable models.

Considering all of these aspects are often neglected but it has been shown to condition the correctness of the co-simulation.

In this thesis, we take into consideration the three aspects previously identified proposing a framework to ease the writing of correct coordinators based on a dedicated set of information on the simulation unit, such as a partial view of the syntax and semantics used internally. The framework is then composed of three main elements (see Figure 1.2): a Model Coordination Interface, a Model Coordination Language, and a distributed co-simulation algorithm as a Runtime Framework. The Model Coordination Interface is inspired by works on the Architecture Description Languages [20, 21], Coordination Languages [22] and heterogeneous frameworks [23–26], and exhibits the minimal amount of information needed to define meaningful interaction between different simulation units, which are developed using different languages or tools.

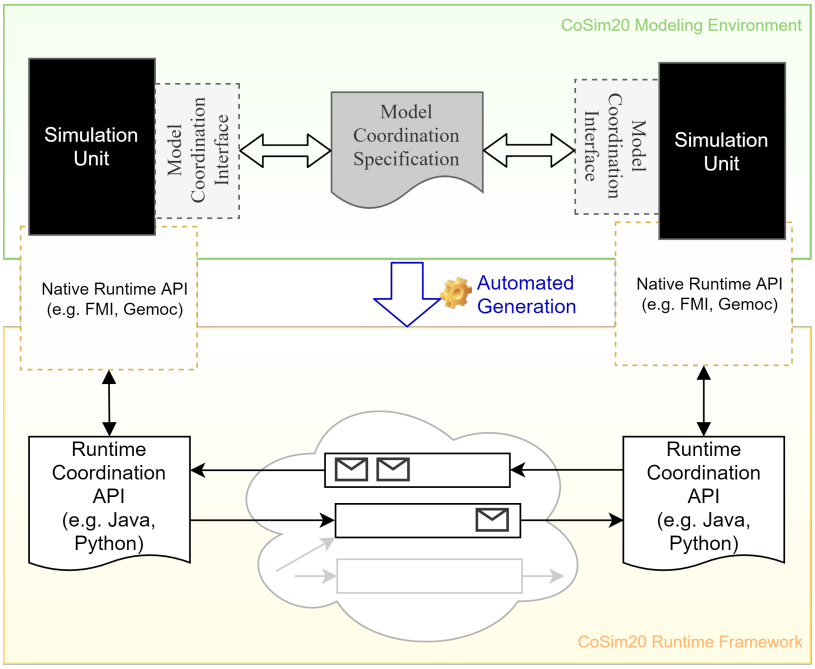


Figure 1.2: Overview of the proposition for the CoSim20 Modeling Environment and Runtime Framework.

The characteristics of the model and its solver are key elements to define the interactions with the other SUs. A possible solution is to exhibit them through an interface tailored to the semantics of the underlying simulation unit. In contrast with the white box *Model Coordination Interface*, this proposition exposes only a partial view of the syntax and semantics of the SU. The interface is tailored specifically to share only the elements necessary to coordinate the execution and communication among the simulation units. The goal is to help to protect the Intellectual Property contained in the SU. We proposed to provide a higher abstraction level about the semantics to ease the reasoning. We divided the interface into three main elements:

- *Set of Ports*, their properties such as name, direction, and type, and their *data nature* that represents the element which exhibits a partial view of the semantics of the models by defining it through its behavior;

- *Temporal reference* used by the model representing which temporal referential the model is using. In the context of heterogeneous systems, each model can be simulated using a different temporal referential *e.g.* time, angle, or distance. For example, in a fuel engine, the camshaft angle is used to define when to open/close valves or to actuate the fuel pumps. The execution semantics for this model is not based on time but on the angle. The actions depend on the movement of the camshaft and not on the physical time;
- *Set of model properties* denoting model-wide behaviors (*i.e.* rollback capabilities or saved states which allow restoring a previous state if necessary). However, this set of properties are not yet exploited by the current proposition.

The resulting interface provides enough information to ensure that a system integrator has the knowledge that is required to correctly coordinate the different executable models, as studied in [27].

The Coordination Model Specification (MCS) explicitly defines the interactions and rules between different Model Coordination Interfaces. It is composed of a set of *Connectors*. Each connector is in charge of defining *when* and *how* one or more data are conveyed from a model to another. In order to define interactions, we structured a connector through three different elements:

- *Interaction* specifies how data are actually exchanged between ports and which transformation should take place here *i.e.* unit alignment between Meter and Feet;
- *Triggering Condition* defines the instant at which the interaction must be realized. The instant can be expressed as a periodic condition (*e.g.* every 5ms) or as an aperiodic condition (*e.g.* event, internal variable usage [11]);
- *Timing Constraint* specifies a relation between the temporal references of the models.

We validate our proposal by developing an Integrated Development Environment (IDE) which supports the specification of Model Coordination Interfaces (MCI) and Model Coordination Specifications (MCS). Two Domain-Specific Languages were then proposed to write these specifications using a language capable to express coordination concepts and their integration. Both languages can be used in a textual environment, provided by Xtext, and a graphical visualizer provided using Sirius. Some facilities, such as FMI importing features and auto-completion ease the designer to write correct specifications. However, the final coordination model must be developed by the designer. Once specified in the coordination model, an automatic process generates the corresponding run-time implementation of the system by providing a Java implementation that supports the co-simulation of the FMI compatible model (*i.e.* FMU) and Gemoc exported models. We propose a runtime framework capable of coordinating several distributed entities without the need for a centralized coordinator. At runtime, this framework encapsulates the SU, according to its MCI, with a dedicated wrapper tailored to its semantics. The wrapper contains a semantics parametrized API to handle the execution of the model and a communication module to interact with the rest of the system.

In the next section, we present the challenges we address in this thesis.

1.2 Main Challenges

In this section, we illustrate our main challenges addressed in this thesis. Our main goal is to develop a co-simulation tool that can be used seamlessly at different stages of the development to support the engineering process among different stakeholders. Some of the main challenges are then presented.

The extended enterprise approach gives the opportunity to share executable models across multiple enterprises. The black-box approach for IP protection is one of the most important aspects of this process and its preservation is mandatory.

The current standards and tools offer little support to enable a correct co-simulation based on a black-box approach. It introduces little or no control over the internal configuration and execution of the model and its simulator.

The different languages and tools used to develop a system lead to heterogeneity of the system in terms of how time is represented and handled within the model and in the global system. Timed models may conform to a different formalism that represents time using different representations (*e.g.* Newtonian, discrete, or superdense [28]). The relationships between those representations are given by their relationship with the physical or wall-clock time. However, due to the heterogeneity of the systems, other representations of time should be able to integrate into the system. For instance, a *multiform* time is a time representation where time process in a non-uniform way and independently from any reference to physical time. In this thesis, we address this problem as temporal synchronization among heterogeneous models.

System Architects and System Designer may want to use the co-simulation as a method to bring out the overall behavior of the system under co-simulation. Its access should be transparent to them and easy to put in place. However, the actual approaches required to configure and tune the algorithm used to coordinate the execution of the co-simulation. Moreover, the specification of the algorithm is written using General-Purpose Language (*e.g.* Java, C++, Python), limiting the choice of co-simulation frameworks according to the language known by the System Architects/Designer. It allows using co-simulation during all the phases of the development, from the early phase of the development where several solutions are evaluated to the verification and validation of the final solution.

The presented challenges will be illustrated and discussed broadly and addressed in the proposition (in Chapter 4). However, this work has assumed some limitations on the systems we are dealing with and their composition to focus our research on the interesting challenges.

1.3 Limitations

In this section, we present the assumptions and limitations for this work.

In an extended enterprise context, we assume that the suppliers want to protect their models using state-of-the-art approaches. Sharing a model with its simulator avoids exposing internal computations and behaviors that can be captured by an external simulator. Note that if the suppliers do not mind sharing a white-box model then our approach can benefit from additional information on its internal semantics and on the simulator control.

The model integration can be driven by the approach choose to exchange models: a white-box approach shares the internal mechanisms and source code, giving clear access to the Intellectual Properties; a black-box approach shares an entity with only exposed inputs and outputs. We think that sharing a black-box model may result in better IP protection given the technical obstacle to reverse engineer the inner algorithms. However, this approach leads to model integration issues whose presence is absent or limited in the white-box approach.

While a single language can be used to develop different models, it is not always the more appropriate choice. The heterogeneity of reality leads to having a different point of view and behaviors to represent. For instance, languages tailored to represent physical processes such as Modelica, cannot be used to represent logical processes such as the behavior of algorithms used in software development.

From the industrial point of view, a black-box approach ensures the Intellectual Property Protection of the internal workings of the model and the knowledge used to develop them.

As assumptions, we choose to focus our work on the coordination aspect of the co-simulation, we assume that the models we are working with are correct and valid.

The recent growing interest in co-simulation techniques changes continuously the state of the art. The importance of this technology has lead several companies and research organizations to invest a significant effort into its research and development: consequently, the state of the art and the maturity of some presented technologies may vary during the last period needed to publish this thesis.

1.4 Structure

In this section, we give the outline of the manuscript by briefly presenting its organization. The thesis is organized into six chapters.

Chapter 1 introduces the context and motivations for this thesis, reviewing the current techniques used for the development of Cyber-Physical Systems. We then illustrate our work, the proposed approaches, and our use case used to validate our proposition. We discussed the main challenges addressed in this thesis, the assumptions we took, and the limitations encountered and identified in the current state of the art.

Chapter 2 introduces the background of Complex Cyber-Physical Co-simulation, giving the reader the context and the essential vocabulary to understand the issues of co-simulation and the problems that this thesis addressed.

Chapter 3 presents an overview of the semantics used to coordinate a co-simulation, the Domain-Specific Languages proposed to address coordination issues and to express the structure and interactions between components, and the interfaces those languages and standards are based on. Then, we illustrate the distributed co-simulation approaches, giving the main concepts of distributed topologies. Finally, we expose the addressed research questions of this thesis.

Chapter 4 describes our proposition by presenting the three main elements of our framework: the *Model Coordination Interface*, the *Model Coordination Specification*, and the *Distributed Coordination Algorithm*.

Chapter 5 illustrates our approach by presenting a representative use case of a CPU cooling system. The heterogeneity of the systems is achieved by decomposing the systems into three different sub-systems: a hard overheat controller, written using a DSL to specify Timed Finite State Machine, represents a software controller that handles the state of the fan; a fan controller, written using Modelica and then exported as an FMU, defines PID controller that computes the speed of the fan according to the current temperature; the fan and the CPU, both written using Modelica and exported as a single FMU, represents the close system to control.

Chapter 6 summarizes our work and provides the conclusion of this thesis along with some future perspectives.

Background 2

Cyber-Physical Systems combine physical, computing, and communication processes to execute tasks to allow interactions with the surrounding environment and humans. The development and operation of those systems are extremely complex tasks. Many different fields such as aerospace, automotive, railway, and maritime have to comply with an increasingly demanding complexity, stricter safety and environmental regulations, and faster project development cycles.

Model-Driven Engineering (MDE) proposes methods and standards to promote and ease communication among different stakeholders. MDE enables the collaboration among different experts but introduces issues during the integration phase. Model changes can affect other models in the system and it becomes difficult to understand the emerging behavior of the system.

Simulation has become a key method to tackle these challenges. It allows a more efficient and quick exploration of the design space, helping in the early phase to retains certain development direction. In particular, virtual prototyping enables an early verification of concepts and their characteristics, dynamics, and performances.

In the next section, we introduce the context of this thesis detailing the complexity of the development for complex systems. In section 2.2, we detail the main concepts of Model-Driven Engineering such as model transformation and model integration.

2.1 Complexity of Cyber-Physical Systems

In Computer Science and Engineering, a system is a group of organized and interacting entities that forms a unified whole to accomplish an overall objective. A system can then be physical, conceptual, or both [29]: matter and energy are the main parts of a *physical system*, information and knowledge are encoded using matter-energy relationships that exhibit the overall observable behavior. A *conceptual system* abstracts the physical parts and relationships of a physical system to exhibit its meaning. It defines abstracted relationships on its element but not the actual interactions. The combination of both systems must be studied, organized, and managed in a well-defined method. An engineered system includes both conceptual and physical elements. A definition is then given by the International Council on Systems Engineering (INCOSE) *: "An engineered system is a system designed or adapted to interact with an anticipated operational environment to achieve one or more intended purposes while complying with applicable constraints" [29].

We then consider a system to be *complex* when it is composed of interconnected heterogeneous components which interact in intricate ways [30]. It is worth mentioning that a complicated system is not necessarily a complex one. A complicated system can be decomposed into its parts and each part can be studied in isolation. Its behavior and properties, which are valid isolated from

* https://www.incose.org/

the original system, are still valid once reassembled with the other pieces of the system. The resulting behavior and properties of the system are the composition of all the behavior and properties of each model. Their interactions do not change their original behavior [31]. In contrast, in a complex system, interactions among the components lead to changes in the behaviors of each model in a not-unambiguous way or unexpected behaviors. For instance, in [32], they present the example of a water tank whose overflows are due to neglected handling of the accumulation of water during the shutdown procedure. That behavior was not taken into account during the development of the system but emerged as a result of the interaction among components.

A complex system then requires to be executed as a whole to observe its global behavior. The execution is a manner to observe the emerging properties and behaviors in such a way that they can be validated and verified against the expected ones. Moreover, in a complex system, the composition of the properties for each component does not hold and they cannot be studied as a disjoint subset of the complex system properties [33].

The model-driven approach is a software design method that proposes to tackle the complexity of the development by relying on abstractions of the different components. The abstracted entities are called *models*. A *model* is an abstraction of an original component, device, or system that reflects only the relevant subset of properties for a given purpose where it can then be used instead of the original, avoiding handling all the complexity of the reality. Within a defined context, it eases the understanding of a specific behavior by excluding the non-essential aspects. A model should be standalone, reusable, and inter-operable [34, 35]. The independence and reusability allow an expert to use existing models as building bricks to develop new systems, along with the degree of abstraction required [36]. The interoperability allows using the model in different systems without significant effort by the final user [37].

In the context of Modeling & Simulation, models are used as a representation of systems, entities, or processes that show their behavior over time. We can then distinguish two different types of time: wall-clock time and simulated time. The wall-clock time is the physical time we normally experience in reality. The simulated time is a virtual mathematical representation of time that emulates the wall-clock time in a simulated environment. There is not a strong correspondence between the two types of time: the simulated time can represent a wider timeline in respect to the wall-clock time spent to simulate it. For example, a geological simulation, that studies the Earth's crust tectonic movement for thousands of years, executes in hours [38]. Vice versa, continuous-time simulations describe physical processes that happen within seconds or minutes but require days or even weeks to execute.

We can use models to test the system without actually test it in the reality. Simulation allows us to understand the impact of a change in the model without an expensive test on the field like in the case of tectonic studies from [38], without the possibility to test a proposition in the real world. Simulation becomes a key technology to develop complex systems that have a direct impact on humans and their life. For instance, the development of a collision-avoidance system for cars can be simulated before the actual implementation and test on a real car, reducing to zero the risk of injuries compared to a real test. An interesting aspect of this example is that it requires the integration of different components, abstracted as models, used to describe the different physics and logical controls that allow the overall system to be simulated.

The increasing integration of technology in our life has developed a category of complex systems called *Cyber-Physical System* (CPS). It uses advanced and smart computational techniques to sense and control some aspects of the surrounding physical world. The interaction of computational (*i.e.* cyber) units and physical unit, when connected through a global network (*i.e.* Internet), brings to advanced interconnected implementations called *Internet of Things* (IoT). Both IoT&CPS aims to support real-time applications that combine digital controls and the physical environment.

In a real-time system (RT), its correctness depends not only on the logical result of the computation but also on the time at which the results are produced [39]. In particular, the software behavior must be predictable in its execution time: a given set of inputs must produce the same output, respecting some temporal constraints such as deadlines. For instance, a user interface in an aircraft cockpit must meet constraints on response times to visualize data within a certain delay. Depending on the importance of meeting deadlines, RT systems are divided into two types: *soft-RT* systems cannot meet their temporal constraints since their failure does result in a dramatic impact on human beings (*e.g.* personal computers or audio and video encoders/decoders); on contrary, *hard-RT* systems must meet temporal constraints otherwise their failure have a dramatic impact on human beings (*e.g.* automotive engines, aerospace and avionics command-and-control, or nuclear power plants control).

Therefore, the complexity of those systems makes their development a challenging multi-expert effort. Each expert focuses on a specific aspect of the system and uses dedicated tools and languages tailored to her or his domain of expertise [27]. Cooperation among these different stakeholders is necessary due to the strong inter-dependency of every part of the system. Cyber-Physical Systems (CPSs) require modeling techniques that take into account the complexity of the design of this system and embrace the cyber and physical parts of the system [40].

For such a system, the optimization of the overall performance as a whole cannot be achieved by optimizing the performance of each subsystem and component. Instead, it must be achieved taking into consideration the interactions between components, environment, control systems, software, and humans behavior.Only if we take into account the system as a whole, we can then optimize it regarding the desired optimization constraints *e.g.* efficiency, safety, consumption, or cost [41].

The model integration is required to understand the impact of a particular optimization on the system. In particular, model developers can have feedback on their changes to study and understand their impact on the system. It is an important aspect since multidisciplinary experts have to work together, usually on the same system but using different models that conform to different tools and languages [42]. The intrinsic connection among different physics, hardware, and software makes the model integration an important aspect in the development of complex systems.

Moreover, different competencies can be distributed geographically. In the globalization era, a modern manufacturing enterprise can combine its effort with several other companies to develop and create a new product.For instance, an extended enterprise [3] is typically an international company with distributed sites across several cities and continents. Engineers, experts, technicians, designers, and system architects need to exchange information and knowledge to reach a final result (*e.g.* the development of a new product).

In the Modeling & Simulation context, the exchange of information among stakeholders is represented by sharing models of components or systems. The challenge is to protect the knowledge and to ensure Intellectual Property while allowing the sharing of parts of the system, as models. A possible solution is to adopt the *black-box* concept: "*A device, system or object which can be viewed solely in terms of its input, output and transfer characteristics without any knowledge of its internal workings, that is, its implementation is opaque*" [43]. On one hand, the black-box concept allows to abstract away the details on how the inputs are transformed to outputs easing the discussion and integration of different models focusing on their inputs and outputs. On the other hand, the internal implementation of the model is hidden and so the language, tool, and formalism used preventing to expose them to the external environment.

Model integration requires integrating models which have been developed using different formalisms and tools. The component-based approach adopted by MDE enables the design of complex systems using reusable components which are standalone executable entities accessible only through an interface [44].

2.2 Model-Driven Engineering

The Model-Driven Engineering (MDE) [45–47] is a software development approach that allows interoperability and reusability of components. It raises the level of abstraction of traditional languages by using the concept of *model* tailored to the concepts it is representing [48]. In the context of MDE, a *system* is a "*generic concept from designating a software application, software platform, or any other software artifact*" [49]. Also, it can be a hierarchical system, with sub-systems, and may communicate with other systems. In this context, *models* are then abstractions of the system under development that already exists or may exist in the future. The developer uses a language at the right level of abstraction and expressiveness to build domain-specific models.

The Model-Driven Engineering approach combines *Domain-Specific Modeling Languages (DSMLs)*, whose syntax and semantics are tailored for a particular domain, and *Model Transformation engines and generators*, which analyze models looking for certain aspects and then generate corresponding entities, called *artifacts*, such as General Purpose Language source code, or transform them into a different model representation.

Initiatives such as the Object Management Group (OMG) [50] and Eclipse [†] help to popularize the Model-Driven Engineering approach by promoting several standards (*e.g.* Unified Modeling Language and its subset *fUML* [‡]), or providing a set of programming and modeling tools like the Eclipse Modeling Framework. This framework permits the implementation of modeling-related languages and tools by providing a set of features and facilities based on the Model Driven Architecture standards of the OMG.

Domain-Specific Modeling Languages *Domain-Specific Modeling Languages (DSMLs)* formalize the elements of the language, such as syntax, type system, and semantics within specific domains, such as avionics embedded controls, hardware description, infrastructure configuration, relational database queries, or even French Tax [51]. The direct representation allows a strong coupling of problems to

[†] https://www.eclipse.org/
[‡] https://modeldriven.github.io/fUML-Reference-Implementation/

solutions. It helps to understand the problem and to represent a possible solution. Furthermore, the reduced expressiveness then General-Purpose Languages and their characterization for a specific domain allow imposing constraints on the syntax to enforce correct patterns. Moreover, the domain-specific constraints allow performing model checking to detect errors from the early phases of the development cycle.

As shown in Figure 2.1, on the bottom of the pyramid, there are the objects of reality, for instance, a car or a house. An element of the real-world is then represented by a model. Concepts, as the model specification, are then defined using a DSML. In the context of MDE, a *model* is an abstraction of some aspect of a system, that exists or may exist in the future. It is created for particular purposes such as to present a human-understandable description of some relevant aspects of the system or to formalize its characteristics to analyze some properties. The resulted abstraction can then be used as a replacement for the reality since it provides equivalent results.

The *model* is then described using a language with a proper expressiveness, such as DSMLs. A DSML, as general-purpose languages, is composed by a *syntax* and *semantics*.

The *syntax* is the set of symbols that can be combined to build well-formed programs. It can be represented as an abstract syntax, that defines the concepts used in the DSML and the relationship between these concepts, and as a *concrete* syntax, that represents the grammars using a human-readable form such as images or words. The DSML syntax is described using a *metamodel*, which defines the concepts in a precise domain along with relationships and constraints among these concepts [52]. It defines the valid elements in a specific model such as *class*, *attributes*, and *associations*.

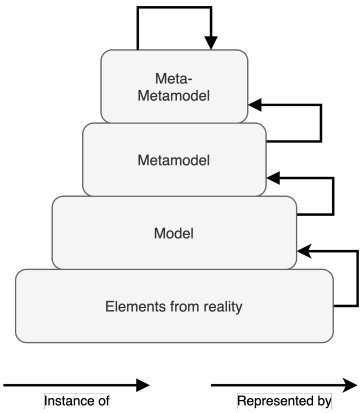


Figure 2.1: Four layer architecture of MDE. Taken from [48].

The *Semantics* gives the meaning to a syntactically correct sentence in the language. There are several forms of semantics, for instance *operational*, *axiomatic*, *translational*, and *denotational*. The *operational* semantics describes how a program is executed on a virtual machine as a sequence of operational steps [53]. The sequence of steps represents the meaning of the program. For this reason, the operational semantics is useful to understand the behavioral execution of a program. In a *axiomatic* semantics, the meaning of a program is given by describing assertions of truth using axioms and proof rules [54]. It is particularly useful to provide the correctness proof of a program with respect to given specifications. The *translational* semantics defines an exogenous transformation from a syntax of a source language to another target language. It exploits the meaning associated with an element in the source language to transform it and create a meaning compatible set of elements in the target language. Moreover, the *translational* semantics may transform a source language into a mathematical model by defining a *denotational* semantics [55]. These semantics are useful to understand and study the internal logic of a program. Software development, integration, and management benefit from the usage of models, metadata, and model-code transformations at different abstractions levels: it is possible to define platform-independent models for business purposes and then refine them to platform-specific models while maintaining predictable transformation between different levels [56].

Model Transformation Engines and Generators Once a model is defined, it can be statically analyzed and checked for inconsistency and errors. Furthermore, the development of a system requires modifying or generate new

models in an automated way. It is possible thanks to *Models Transformations Engines and generators*. Model transformations play a central role in the MDE, by allowing to modify and refactor models or generate a new model specification from an existing one. A model transformation is a set of *transformation rules*. A rule defines which changes to perform on a specific element of the model, based on its metamodel.

There are two types of transformation: if the transformation is done to convert a source model to a target model and both models conform to the same metamodel, then it is called an *endogenous transformation*. On the second type, if the transformation converts a source model to a target model and both models conform to different meta-models, it is then called *exogenous transformation*. In this thesis, we use the second type to transform a set of DSL specifications that represent the Model Coordination Interface and the Model Coordination Interface into a set of Java source code specifications that can be later compiled and executed.

This automated model transformation helps to achieve a "correct-by-construction" implementation process, as opposed to the conventional "construct-by-correction" software development implementation process that is tedious and error-prone. Several frameworks and languages were proposed to specify and perform model transformation tasks. We then distinct two main categories: model-to-model and model-to-code transformations. In fact, a model-to-code approach can be considered as a special case of model-to-model transformation: the target generated code is usually a textual model, which is compiled or interpreted later by the corresponding compiler or interpreter. We can then categorize the approaches by the techniques used.

The model-to-code approaches can use a visitor-based or a template-based mechanism to transform a model into code.

The model-to-model approaches can be based on direct-manipulation, graph-transformation, structure-driven, or relational.

Some of these languages are then supported and integrated into the EMF. For instance:

- *ATL* [57] is declarative transformation language that defines a model transformation by declaring a set of rules. A rule is composed of a source pattern and a target pattern. The source pattern is used to identify a subset of the source model, while the target pattern defines the transformation to apply. The application of those model transformations consist to execute a pattern-matching task in a loop;
- *VIsual Automated model TRAnsformation* (*VIATRA*) is a declarative query language based on graph transformation patterns. It is based on the same concepts of ATL The model transformation is then applied on the same source model to ensure termination [58];

Modeling Tools Developing a language requires developing some programs such as a lexer, a parser, and an evaluator. Moreover, nowadays modern languages are expected to be shipped with extra features as syntax highlighter, syntax checker, completion, and quick fixes. In this context, the Eclipse Modeling Framework (EMF) [59] offers an environment composed of a set of Eclipse plug-in which forms a modeling framework that enables to define Domain-Specific Languages and related tools. It provides an environment with tools and facilities such as model and metamodel transformations, model checking, and model-to-model and model-to-code transformations. The metamodel specification can

be described using XMI, UML, or annotated Java. Usually, the metamodel in specified using a dialect of the UML class diagram called Ecore. It implements a widely used standard to define a metamodel called Essential Meta-Object Facility (EMOF). Using the Ecore representations, we can then defined the syntax of a DSL defying its grammar and the relationships between its elements.

The metamodel can then be used to automatically generate tooling for that language. For instance, Xtext is a framework that supports metamodel defined as Ecore and enables the automatic generation of the associated tooling such as the parser, the lexer, the type-checker, and modern facilities as the syntax highlighter, completion, and quick-fix.

A meta-model conforming to the *Meta Object Facility (MOF)* [§] standard is defined in terms of packages, classes, properties, attributes, and operation signatures. These elements can be used to represent the syntax of a language but it does not include the support to specify its operational semantics and constraints. For this reason, EMF integrates into its framework a set of DSL dedicated to defining the semantics of a language. For instance:

- *Kermeta* [60] is action language based on the aspect-oriented paradigm that allows to extend meta-models with dynamic and static semantics.

2.3 Model Integration

Model-based design of Cyber-Physical System (CPS) requires approaches that can handle both the cyber and the physical part of the system [61], accepting the heterogeneity of the system along with the formalisms, tools, and techniques with whom it is developed [62]. The component-based approach is used as the base mechanism to enable integration and simulation of models. In particular, co-simulation is a model integration approach in which models are integrated using their exposed inputs and outputs.

In this context, *co-simulation* enables to combine heterogeneous components (*i.e.* models) following the *hierarchical heterogeneity* principle [9, 14]. It consists of techniques to enable the simulation as a whole of a coupled-system by composing different heterogeneous models and simulators. For instance, a co-simulation can integrate continuous dynamics into real-time hardware-in-the-loop simulators [63], physical test stands [64], or software and hardware [65].

The basic element of co-simulation is the *executable model*: it can be run, tested, measured, and debugged as executable code. As an executable code, it is possible to read its implementation only if the program code is available: in this case we refer to the model as *white box*. Otherwise, if the program code is not available and only the executable is provided, then we refer to it as *black box*. As seen in Section 2.1, the black-box approach is particularly useful in the context of an *extended enterprise* that adopts a model-based design. It allows to share models and thus reusing existing ones created with dedicated knowledge and softwares [33], while protecting the Intellectual Property of vendors and suppliers, to reuse a well-defined and built model, and to ease the integration and exchange of models.

In this thesis, we refer to components and models using a wide-open definition that embraces different by nature entities. We use the term Simulation Unit (SU) to denote an entity that produces output data and consumes input over its execution.

[§] https://www.omg.org/mof/

It can be a stand-alone model with its simulator, a real-world component, a software with its interpreter, a proxy to an existing part (hardware or software-in-the-loop), or a composition of these entities. A SU is then encapsulated into a black-box hiding its internal mechanisms and semantics. However, conforming to the co-simulation model integration concepts, it is possible to execute and interact with a SU through its inputs and outputs.

Traditionally, the simulation of complex systems results in a closely-coupled and monolithic simulation customized for a specific purpose. The entire process is costly and lacks the reusability of components. Co-simulation enables interconnecting loosely-coupled independent models and sub-systems and simulating the created system as a whole. The execution of the different SUs needs to be orchestrated to obtain a coherent and correct simulation. The algorithm that controls the execution of each SU is called *Master Algorithm*. The MA controls how the simulated time advances in each SU and exchanges data among the SUs according to their topology, given by their inputs and outputs connection.

Functional Mockup Interface (FMI) A widely used standard that carries out supporting these requirements is the Functional Mockup Interface Standard (FMI) [4], introduced by the MODELISAR project [66] in 2008. It defines a standardized interface of a simulation unit to support collaborative simulation, composing model components designed using different modeling tools [67, 68]. The simulation unit, called Functional Mockup Unit (FMU) implements the FMI Standard interface, providing a set of functions that the MA can exploit to control the advancement of the internal simulated time, to set input data, and retrieve output data. The internal time advances step-by-step by defining an integration step size to pass to the model.

High Level Architecture (HLA) The High Level Architecture (HLA) [69] standard was originally developed and standardized by the Defense Modeling & Simulation Office (DMSO) in the 90s. The specification has two main standardization: the HLA 1.3 [70], the last final release from the defense community, and the IEEE 1516 [71] released to facilitate the adoption of the standard for applications other than military simulation. The standard defines a general-purpose architecture specification based on an event-driven Application Programming Interface (API). However, the standard does not provide any implementation. A distributed simulation is called *Federation* and it is composed of several entities called *Federate*, which interact with each other by using the Run-Time Infrastructure (RTI). The RTI provides several management services such as Federation, Object, Time, Declaration, Ownership, and Data Distribution Management. The *federates* exchange data by relying on a publish/subscribe pattern. Time synchronization and event management are then handled by the RTI. HLA provides two possible styles of time synchronization for federates: a time regulating federate prevents other federates to advance their internal time by sending their internal time as a timestamped event, and time-constrained federate which internal time advancement is constrained by other federates and executes itself under the constraints to process events in their timestamped order (using a conservative or optimistic approach).

Combination of FMI and HLA and their limitations Both FMI and HLA present a standardized interface to enable co-simulation among different simulators and executable models. While the HLA standard is more focused on

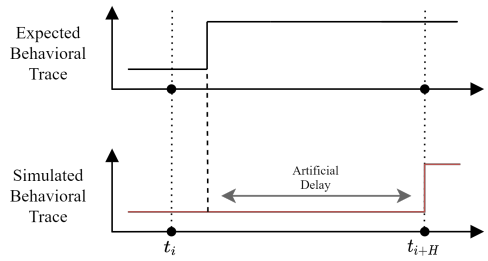


Figure 2.2: Artificial delay introduces by the sampling in time-triggered approaches.

the distribution of the co-simulation, by defining the RTI that embeds critical coordination services such as data exchange, time advancement, and event handling. On the contrary, in FMI these services are not defined by the standard and must be implemented into the Master Algorithm. The former proposes a time-triggered API while the latter an event-driven API. In the CPS context, neither FMI nor HLA could support natively the co-simulation of a heterogeneous system. Several works [11, 72, 73] show side effects of using pure time-trigger communication in a co-simulation of a hybrid system. Two main concerns must be taken into consideration: the artificial delay introduced by the sampling and the delay in the propagation of data between SUs.

The first concern results in a bias in the internal simulation time. As shown in Figure 2.2, the time-triggered approaches retrieve data from a SU at specific points in time called *communication points*. A finer accuracy reduces the performance, augmenting the communication points required. Fast simulation implies reducing the number of communication points, which may produce non-correct simulation results [11]. An ideal solution could be to go beyond time-trigger approaches in order to use the right number of communication points to achieve a correct simulation. It requires coordinating a component in such a way it produces and consumes data only when its internal semantics require it to. This idea is one of the principles of the approaches presented in this thesis.

The time-triggered approach imposes to propagate data from a SU to another at specific points in time (*i.e. communication points*). At these points, for each SU, the MA is in charge to copy the data from the outputs port of a SU to the connected inputs on other SUs. In a monolithic or a local co-simulation, the overhead due to this phase may be a minor issue. Data can be transferred by copy or reference, avoiding moving them across different memory areas. However, in a networked co-simulation, each communication point curtails the overall performance: data must be moved from a machine to another over a network. In this case, the data transfer time introduces an important overhead for simulation time. Approaches like [73] proposed to minimize data transfers that require network by grouping in the same machine the SUs that have a strong connection (*i.e.* strongly coupled).

To reduce the simulation time, it is important to reduce the communication between the models. Two options are available here: reduce the number of communication points reducing the sample rate (non the right choice due to the instability added to the system that results in wrong simulation and results) or to reduce the number of communication points by adapting them to the minimum required to achieve a correct simulation of every model.

2.4 Conclusion

In this chapter, we provided an overview of the main concepts in Model-Driven Engineering and the context of this thesis. Many concepts that were introduced briefly are then discussed in later chapters.

We focused on the complexity of the development of software and hardware systems that interact with the real world. The increasing complexity of those systems and their capabilities required advanced techniques to understand and develop them while meeting fast development, safety requirements, and user's needs. Model-Driven Engineering tackles the complexity relying on abstractions of the reality, allowing to work and focus only on relevant parts of the system, called models.

Models are described using dedicated languages, with their syntax and semantics tailored to a particular domain to help experts to express their solution using elements and concepts close to their domain of expertise. The intrinsic connections among models in a system can cause integration problems due to the difference between different semantics and tools used. In this thesis, we focus on these integration problems and, in particular, on the semantics gap caused by generic master algorithms. These algorithms do not consider the semantics specificities of the models: the internal semantics of the model is hidden behind the interface due to the black-box hypothesis meanwhile the master algorithm enforces its semantics to the model through the use of specific API (*e.g.* time-triggered API in FMI Standard). On the contrary, a *dedicated* master algorithm defines a semantics-aware coordination model tailored to the semantics specificities of each model. However, those specificities require to be exposed and exploited by the master algorithm.

To ease the specification of a *dedicated* master algorithm, we propose a Domain-Specific Modeling Language dedicated to the coordination domain: it uses coordination concepts and elements to build optimized connections between heterogeneous simulation units.

In order to define a *dedicated* master algorithm, we propose an interface tailored to coordination purpose: it exposes a partial view of the internal syntax and semantics of the SU allowing their exploitation to construct a *dedicated* master algorithm.

State of the Art 3

Complex system engineering requires integrating several different formalisms, tools, and standards. The intrinsic complexity of the real world leads to the use of abstractions and languages dedicated to different domains that explicitly show interesting behaviors. Tools and languages are used to express these behaviors and reason on possible solutions. The heterogeneity of those languages and tools leads to a transposition of the complexity that we find in the real world, also in the engineering world. In the Software Architecture community, Architecture Description Languages (ADLs) were proposed to tackle this problem: they provided Domain Specific Languages (DSLs) to organize and integrate different models. Typically, an Architecture Description Language (ADL) manages a SU as a component, encapsulated into a homogeneous interface that exposes some relevant part of the model as a set of input/outputs that allow exchanging data with the model. Then, based on that, they specify the connections between models. It is worth noticing that ADLs and Coordination Languages (CLs) mainly focus on the integration of software components instead of Cyber-Physical System (CPS) components; however, they proposed interesting concepts that were used in this thesis.

In the simulation community, the collaborative simulation focuses on the orchestration among different simulation units that represent different parts of the same system, in order to better understand the emerging behavior of the system. In this context, the simulation unit is a, usually black box, executable entity, which may for instance encapsulate a model and its solver, a binary executable process, or a proxy to a hardware device. The orchestration is then of prime importance because it defines the instants when a simulation unit exchanges data with other simulation units. There exist several models of coordination that follow different semantics. For instance, the most popular coordination models are based on time-trigger and event-trigger semantics. In addition, a coordination model may be distributed across different cores, CPUs, devices, or network infrastructure. All these properties and semantics heterogeneity impact the accuracy and the overall performance of the co-simulation, as we will discuss later in this chapter.

In the next section, we present the main coordination semantics used to coordinate a co-simulation such as Continuous-Time and Discrete-Event semantics. In section 3.2, we analyze the Domain-Specific Languages for Co-simulation proposed to describe a collaborative architecture and the interactions that must occur among its components. In section 3.3, we analyze the coordination interfaces exploited by the presented languages. In section 3.4, we present the co-simulation approaches highlighting their capabilities to distribute a co-simulation execution. Centralized and distributed approaches are presented based on the supported topology. Finally, in section 3.5, we discuss the correctness of the co-simulation approaches and we present the three identified research problems that we address in this thesis.

3.1 Coordination for Co-simulation

In this section, we illustrate the techniques used to coordinate the execution of a co-simulation. The terms *simulation* and *co-simulation* can be confused during the reading: for the sake of clarity, we use the term *simulation* to indicate an independent process that executes a model and its solver, a software and its interpreter, or an executable. These entities are considered to run within a process or a thread and they were not meant to run in a cooperative and collaborative context. Consequently, we use the term *co-simulation* to indicate a collaborative simulation between two or more simulations. In particular, we perform co-simulation using a black-box simulation unit: the internal mechanisms and algorithms are hidden behind an interface that exposes an API to be controlled from the external without disclosing its internal IP.

In this section, we analyze the underlying algorithms used to actually run the co-simulation and which implications they have on the execution. In the literature, the algorithm that coordinates the execution of a set of models is called in several ways: Master Algorithm and orchestrator [12] in Functional Mockup Interface (FMI) [4], coordinator in CLs [74], or director in Ptolemy II [14]. In this thesis, we refer to the implementation (*i.e.* source code) of the executable semantics used to develop the coordination as *coordination algorithm*. It is worth noticing that the coordination algorithm can be distributed or centralized: the FMI community proposed mainly centralized Master Algorithms, meanwhile, Coordination Languages focused on distributed coordination [75, 76].

The next subsection introduces the coordination algorithm for Continuous Time (CT)-based co-simulation, subsection 3.1.2 focuses on Discrete Event (DE)-based co-simulation, and finally subsection 3.1.3 analyses approaches which propose a hybrid coordination algorithms to integrate CT and DE-based simulations.

3.1.1 Continuous-Time Based Co-simulation

In Cyber-Physical Systems, the *physical* components are usually designed and modeled using differential equations or a language on top of them. For instance, Micro Electro-Mechanical Systems (MEMS), mechanical parts, and analog circuits can be described using differential equations; these equations can be then defined using the Modelica language [77] that supports the definition of differential equations systems. In these components, the input and output signals are mostly continuous-time signals. Differential equations used to model the continuous-time dynamics can be divided into several types: the distinction is based on the fact that equations are ordinary or partial, homogeneous or heterogeneous, and linear or non-linear. For instance, a non-exhaustive list comprehends Ordinary Differential Equations (ODE), Differential Algebraic Equations (DAE), and Partial Differential Equations (PDE).

Simulation In particular, we focus on Ordinary Differential Equations (ODE), which, in this thesis, are differential equations over the time variable. A common approach to expose numerical solutions of differential equations is by using numerical integration. The base problem is to compute an approximate solution to a definite integral to a given degree of accuracy. In this section, we give a brief overview of numerical integration and solvers based on [78] definitions.

We introduce a simple model with two continuous-time variables, ω and x, as a descriptive example. We assume that τ is an integrable function of the form

$\tau : \mathbb{R} \rightarrow \mathbb{R}$. Further, we assume to provide a function to evaluate $\omega(T)$ for any $t \in \mathbb{R}$. Then, the model provides x according to

$$x(t) = x_0 + \int_0^t \omega(\tau) d\tau$$

where x_0 is a constant. Then, $x(t)$ represents the area under the curve formed by $\omega(\tau)$ in the interval from $\tau = 0$ to $\tau = t$, translated by the initial value x_0. Therefore, we can compute x by providing τ as the input to an numerical integrator with the initial state x_0.

The basic problem of numerical integration is to compute an approximate solution to a definite integral to a given degree of accuracy. The degree of accuracy depends on the application: one of the criteria to determine the required accuracy is that the computed value for x should be sufficiently accurate at a sufficient number of points that those values can be used to calculate future values of x in time $t \in \mathbb{R}$

The realization of a numerical integration algorithm is called *solver*. We introduce a numerical approach based fixed-step size solver called *Euler* method. It is the simplest of explicit numerical methods for solving differential equations. A constant step size h is fixed, the approximation x_{n+1} to $x(t_{n+1} = (n+1)h)$ is explicitly computed from x_n as

$$x_{n+1} = x_n + h\tau(x_n)$$

Starting from the initial value $x(0) = x_0$, the method computes the sequence of approximations $x_1, x_2, ..., x_n$ to the solution using one evaluation of f per step [79]. This method is less accurate and errors tend to accumulate faster but the solver does not require knowing the input at time $n + 1$ [78]. If a high degree of accuracy is required, a smaller step size h can be used but, as a drawback, it increases the number of computations.

The *Euler method* can be generalized to adjust the step size according on how rapidly the signal is varying. Such solvers use specific algorithm to evaluate the more appropriate step size to evaluate the model at each time instant. First, a tolerance must be chosen based on the degree of accuracy required. The idea is that the algorithm fixes a step size h, then computes the first approximation x_{n+1} and the estimated error ϵ. If the error is above the tolerance ϵ, then the algorithm must re-computes the approximation x_{n+1} using a smaller step size h. The variable-step-size *Euler solver* will then define a time increment h_n to compute $t_n = t_{n-1} + h_n$ and then compute

$$x(t_n) = x(t_{n-1}) + h_n\tau(t_{n-1})$$

The variable-step-size *Euler solver* is then a special case of the wide used Runge-Kutta (RK) methods [78, 80] in co-simulation.

Co-simulation of Continuous-Time Simulation Units The objective of a co-simulation is to approximate the behavior of the coupled SUs with a certain degree of accuracy. In this context, where several models have their internal step size and are independent of another, a macro-step size has to be defined. We then defined H as the communication step size (*i.e.* sample rate value) at which dynamic models are required to exchange data. Usually, it is greater or equal to all step sizes defined in each model. To orchestrate the overall co-simulation, a Master Algorithm (MA) needs to be defined. It is in charge to determine in

which order the models are given inputs and outputs and instructed to compute the next time step interval. Two widely used methods to implement the MA are then here presented: *Gauss-Seidel* and *Jacobi* methods [81].

An *algebraic loop* can occur when an input port with direct feedthrough is driven by the output of the same simulation unit, either directly (see Figure 3.1), or by a feedback path through other simulation units which have direct feedthrough (see Figure 3.2).

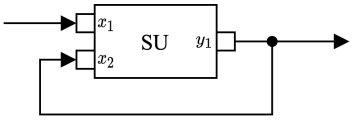


Figure 3.1: Simple example representing an algebraic loop on the same simulation unit.

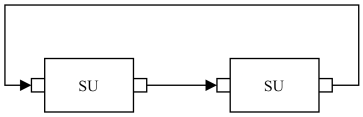


Figure 3.2: Simple example representing an algebraic loop between two simulation units.

If an algebraic loop is detected, there are two possible solutions [15]: perform a fix-point algorithm [82, 83], or add a third component that delays one of the two inputs [15].

However, in many cases, solving an algebraic loop is not always possible due to the lack of rollback support, and adding a third component, with input/output dependencies, may not be a feasible solution. The solution of algebraic loops is a well-known problem [9, 15, 83] and it is outside of the scope of this thesis. However, amenable algebraic loop solvers, based on the Newton-Raphson approach, have as requirement smooth functions and are not compatible with discrete changes [84]. For this reason, algebraic loops should be rejected by our framework and a warning will be raised to the system designer. Moreover, the following algorithms will not take into consideration algebraic loops.

In *Gauss-Seidel* method, the MA asks sequentially each model to compute its next time step interval and retrieves outputs. Then, the MA provides the most recent outputs to the next model (according to their topology). The sequential nature of the algorithm requires establishing order among the models. The total order can be found using approaches proposed in [85]. A possible implementation of the *Gauss-Seidel* method is given by [85], which is reproduced in Algorithm 1. We illustrate the pseudo-code for the implementation of the *Gauss-Seidel* method to coordinate a co-simulation system. We define the list of models as a vector of Simulation Units SU. The vector SU is already ordered using a topological sort. The algorithm takes as input the simulation stop time T, the communication step size H, and the ordered vector SU. The method $C_{[su]}(\{y_{[v]} | v \in S_{[su]}\})$ computes the input for the simulation unit su from the output samples of its sources. The method $getOutput(su, uc_{[su]})$ retrieves the output values for the simulation unit su, using the value in the variable $uc_{[su]}$, if needed. The method $doStep(su, H, uc_{[su]}, up_{[su]})$ ask to perform a computational step of size H to the simulation unit su.

In *Jacobi* method, the MA asks simultaneously every model to compute their interval. Then, at the end of their step, the MA sets their inputs (according to their topology). The list of models SU does not require to be ordered due to the fact that the master algorithm does not impose any execution order. A possible implementation of the *Jacobi* method is given by [85], which is reproduced in Algorithm 2. Usually, the *Jacobi*-based algorithm is less accurate than the *Gauss-Seidel*-based, due to the fact that models cannot use interpolation techniques [85]. However, it is possible to take advantage of parallelism to execute the system in a distributed way.

When the Continuous-Time based technique is used for CPS co-simulation, it raises an integration problem given by the temporal location of events. The time-trigger master-algorithms presented in literature and commonly used do not natively support simulation units that present discontinuity or trigger discrete-event. Due to the widespread adoption of the FMI standard [4] that spreads the use of co-simulation for CPS, several works addressed the event location problem,

Algorithm 1 Master Algorithm based on Gauss-Seidel method. Taken from [85]

```
1:  for su ∈ SU do
2:      uc_[su] := y_[su] := 0                ▷ uc →Inputs; y →Outputs
3:      up_[su] := 0                          ▷ up →Previous Inputs
4:  end for
5:  t := 0
6:  for su ∈ SU do                ▷ Compute the output for each model.
7:      uc_[su] := C_[su]({y_[v] | v ∈ S_[su]})
8:      y_[su] := getOutput(su, uc_[su])
9:      up_[su] := uc_[su]
10: end for
11: while t <= T do
12:     for su ∈ SU do
13:         uc_[su] := C_[su]({y_[v] | v ∈ S_[su]})
14:         doStep(su, H, uc_[su], up_[su])
15:         y_[su] := getOutput(su, uc_[su])
16:     end for
17:     for su ∈ SU do
18:         up_[su] := uc_[su]            ▷ Update previous input
19:     end for
20:     t := t + H                        ▷ Advance time
21: end while
```

proposing different techniques mainly based on the mechanism of step-rejection and rollback.

A simulation unit may reject a proposed communication step size: if the one proposed is larger than the minimum supported, then the SU may reject it [86]. This mechanism is then used also in case, during the internal simulation of the SU, a discontinuity occurs. How the discontinuity is handled then depends on the capabilities and nature of the simulation units. In case the simulation unit encapsulates a CT-based entity, then the discontinuity should be handled by re-initializing the SU [87]. However, the main drawback of this approach is that it can cause a cascade of re-initializations in the system, degrading the overall performance and increasing the simulation time [88, 89]. A technique to re-initialize a SU consists of *rollback* the SU simulation to a previous state in which the discontinuity has not yet occurred. It requires the SU to save its state and restore it whenever necessary. However, the rollback mechanism may not be supported by all simulation units.

In a CT-based simulation unit, the event location is associated with the zero-crossing detector that triggers an event when a continuous signal crosses a defined threshold. Furthermore, those events are typically considered uncommon. However, in a Discrete-Event based simulation unit, events are an important element of the semantics. The methods proposed to locate events are not well-suited due to the high number of rollbacks needed: in particular, all the proposed works to minimize the needed rollbacks are worthless because rollbacks are systematic.

In the context of FMI, several master algorithms were proposed to handle heterogeneous systems such as CPS. A majority of them are variants of well-known Jacobi or Gauss-Seidel methods and dedicated to (continuous-time) system of differential equations [6, 7, 9, 15, 90–92]. Two possible solutions, to approximately locate the instant when an event is detected, are to compute the

Algorithm 2 Master Algorithm based on Jacobi method. Taken from [85]

```
1:  for su ∈ SU do
2:      uc_[su] := y_[su] := 0
3:  end for
4:  t := 0
5:  while t <= T do
6:      for su ∈ SU do                ▷ Compute the output for each model.
7:          uc_[su] := C_[su]({y_[v] | v ∈ S_[su]})
8:          y_[su] := getOutput(su, uc_[su])
9:          up_[su] := uc_[su]
10:     end for
11:     for su ∈ SU do
12:         doStep(su, H, uc_[su])
13:     end for
14:     t := t + H                    ▷ Advance time
15: end while
```

integration step using the minimum step size accepted by the SU [93] or to use the bi-sectional method [94, 95]. The main drawback is that all of them require a rollback mechanism, decreasing the overall co-simulation performance.

Interfaces and Tools A widely standard for co-simulation is the Functional Mockup Interface (FMI) Standard [4]. It is a tool-independent standard that proposes to bundle a simulation unit behind a homogeneous time-driven API. The master algorithm is not part of the standard and it must be implemented. It can *set* or *get* the current value of an exposed variable (according to its direction) by using the standardized FMI API. This API is also used to simulate the model for a specific interval of time specified in the *doStep* method. In the co-simulation mode, each FMU solver decides how many computational steps should be done in that time interval to reach the desired precision.

Conclusion These standards and tools were originally designed for continuous-time co-simulation. In fact, in the FMI standard, we find many features and properties to improve co-simulation results and performance based on numerical solver capabilities (*e.g.* state variables and derivatives of signals). With the increasing use of co-simulation for CPS, several works introduce proposals to support different formalism other than CT-based, such as discrete-event based simulation units. For instance, [11, 19, 28] propose to update the current FMI API to natively support different formalisms such as Discrete-Event. However, these changes break the compatibility with the co-simulation tools that supports FMI 1.0 and FMI 2.0 and they must be accepted by the FMI community to become part of the standard.

In the next subsection, we introduce the characteristics of Discrete-Event (DE) co-simulation along with standard interfaces and tools use for DE co-simulation.

3.1.2 Discrete-Event Based Co-simulation

Continuous-time models are appropriate to design and analyze physical models but they are not well suited to design, simulate and analyze a large collection of models or complex systems due to their size. Systems as networks, software, CPU, and hardware design present some phenomena that cannot be expressed

using a CT-based system. In particular, problems such as *synchronization*, *mutual exclusion*, *parallelism*, *contention*, and *scheduling* cannot be defined and studied using differential equations. In this section, we first introduce Discrete Event (DE) simulation, and then we describe the collaborative simulation approaches that exploit the Discrete Event semantics.

The simulation unit time, which evolves internally in the model, may not be synchronized with the *real time* or *wall-clock* time, which is the time that elapses in the real world while the model executes. Model time can advance faster or slower than the wall-clock time. In some contexts, such as Hardware-In-the-Loop simulation driven by DE algorithms, the algorithm must take into account the delay introduced by the simulation: if the computer simulates the system fast enough, the simulation must "wait" the real world time to synchronize the execution. Moreover, if the simulation time is too slow compare to the *wall-clock*, then it is not possible to perform a Hardware-In-the-Loop simulation. In a Discrete-Event simulation, time advances not by emulating the *wall-clock* (*i.e.* represented as a continuum) but by the changes of system's states at discrete instants in time.

More precisely, a Discrete-Event Simulation (DES) describes the behavior of a system by defining how the internal state evolves according to a discrete sequence of events. An event is an instantaneous occurrence that changes the state of the system at a specific point in time, called timestamp. The timestamp has a scope limited to the simulation unit. An event can produce other events with the same timestamp or with a greater timestamp. In contrast to a continuous-time simulation, a discrete-event simulation unit is not allowed to change its state between two successive events. The simulation is then carried on by increasing time and updating the model's state accordingly (or when necessary).

In Figure 3.3, we illustrate the main concepts of a simulation algorithm of a standard DE simulator. The main mechanism for advancing simulation time and ensuring that events occur in the correct chronological order is based on the idea of an event list, called Future Event List (FEL). The FEL contains all event notices for all events that have been scheduled to occur in the future. The chronological order implies that all events must satisfy $t \leq t_1 \leq t_2 \leq ... \leq t_n$ with $n \in \mathbb{N}$ where t is the current simulation time. The scheduling of future events implies that activity will push, on the event list, the next associated event. A *global clock* can then be used to synchronized the events across the system.

In a loop, the simulation algorithm then takes the first event from the list, sets the current simulation time to the time in the event, pulls it from the list, and performs the set of actions associated with the event. An *action* can then generate new events that will be pushed into the event list. Their associated timestamps must be equal to or greater than the current simulation time. When the event list is empty or the end of the simulation time is reached, then the simulation is terminated.

If an event with a larger timestamp was executed before a smaller one that would schedule this bigger timestamped event, a logic error would occur. These types of errors are called **causality errors**. In a single DE simulation, the time management is ensured by a local clock to which every event refers for their timestamp. In a system where multiple parallel DE simulations execute, a local clock does not guarantee a consistent timestamp for all the events.

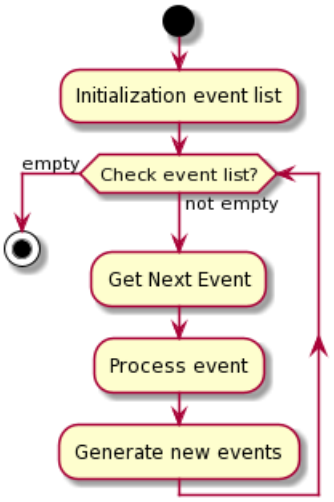


Figure 3.3: DE simulator activity diagram

Co-simulation of Discrete-Event Simulation Units The objective of the DE co-simulation is to correctly simulate the DE-based simulation in a collaborative way. In a Parallel Discrete Event Simulation (PDES), also called Distributed Simulation [96], an independent Discrete Event simulation runs on a parallel machine (*i.e.* computer). The main concern is the time synchronization among the components of the system: when multiple parts of a DE system execute in parallel, they may trigger events sporadically or at a different rate. In a PDES, a possible solution is to use a global simulation clock shared among computers. A *lock-step* execution ensures that at each *tick* of the simulated time, each event list on the computers is checked and the events due in time are executed. However, the overall execution may suffer from poor performance due to the fact that two events rarely have the same exact timestamp [96]. In fact, if no events occur at the same timestamp, then the simulator must stop at every occurrence of an event, checking the event list and executing the scheduled events. This process must be done in sequence in order to prevent causality errors.

A possible solution to speed-up the simulation is to allow a concurrent execution of events that happens at a different simulated time. It allows simulating parallel events of different computers. Usually, the simulation runs as a physical process on a single processor: a typical approach is to map a physical process to a logical process (LP) and each LP can independently simulate.

In this parallel execution contest, to prevent *causality errors*, we define a **local causality constraint**: "*A discrete event simulation, consisting of logical processes (LPs) that interact exclusively by exchanging time-stamped messages, obeys the local causality constraint if and only if each LP processes events in non-decreasing timestamp order*" [97]. It ensures the correctness of the simulation without introducing errors caused by the simulation itself (*i.e.* by avoiding causality errors). The causality constraint is sufficient but not always necessary to guarantee that no causality errors occur [97]. One of the main challenges in PDEVS is to concurrently execute logical processes ensuring correct simulation results. It is possible to achieve this goal by providing a formal algorithm to synchronize the execution of each logical process with the rest of the system.

Synchronization refers to the correct execution of the DE component in a distributed system ensuring that repeated experiments of the execution with the same set of input produce the same results [97]. Time management algorithms can be divided into two main categories: *conservative* and *optimistic synchronization*. These approaches were developed to execute distributed or parallel simulation units on a distributed network or multiprocessors platforms.

The conservative approach strictly prevents the possibility of any causality error by determining if it is safe to process an event. The first algorithm was developed by [98]: in a network that ensures that messages are received in the same order as they were sent, each LP can send its non-decreasing timestamp within a message. Messages are then organized in a First-In-First-Out (FIFO) queue and executed. Events are then scheduled according to the FIFO queue. The local simulation starts consuming the event with the lowest timestamp: local events are then scheduled within the LP in a dedicated event list. When the messages queue (on every LPs) becomes empty, then the LP stops and it is in deadlock. To avoid it, messages with *NULL* events are used. The *NULL* event cannot generate new events or update the internal state of the system. *NULL* messages are a key element of the *lookahead* concept: if an LP is at simulation time T and it ensures that outgoing messages in the future will have a timestamp at least of value

$T + L$, where L is the lookahead period. However, this approach results in a high number of *NULL* messages, and in a high computational overhead [97].

The optimistic approach uses a *detect and recovery* strategy: causality errors are allowed to happen but, when they are detected, a recovery mechanism (*i.e.* rollback) must be executed. The optimistic approach has two main features: a high level of parallelism and the synchronization is more transparent to the application than the conservative approach. However, due to the rollback mechanism in case of violation of causality constraint, it may result in a cascade of rollbacks that impacts the overall performance. One of the widely used algorithm, called Time Warp (TW), was developed by [99]: it allows that each LP independently advances its own simulation time and, when a causality constraint violation is detected then Time Warp performs a rollback, restoring the state previous to the violation, and re-computes the events that violated the temporal constraint [97]. The main problem with this approach is that I/O operations cannot be always undone with a rollback. A solution to this problem is to use a Global Virtual Time [100] that defines a lower bound on the timestamp on every future rollback, allowing to discard data written before the Global Virtual Time and the saved states affected.

The main drawback of the optimistic approach is the high cost due to state-saving and rollbacks [100]. In CPS co-simulation context, simulation units cannot always provide a rollback mechanism and it impacts the usability of this approach in the scenarios where there are those SUs. On the contrary, the conservative approach does not require a rollback mechanism due to the *lookahead* mechanism that prevents a SU to go ahead in time if it depends on other SUs.

One of the first discrete-event co-simulation formalisms used to co-simulate DE models is the Discrete Event System Specification (DEVS) [101]. In particular, it defines a coordination interface used by simulation units to communicate with their external environment (*e.g.* Master Algorithm) that will be presented later in this chapter (see section 3.3). In [102], authors define an atomic DEVS (see Figure 3.4) specification as a structure

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

where X is the set of inputs port through which external events are received, Y is the set of ports through which internal events are sent to the external environment (*e.g.* a connected models), S is the set of states, $\delta_{int} : S \rightarrow S$ is the internal transition function which specifies to which next state the system will transit after the time given by the time advance function has elapsed, $\delta_{ext} : Q \times X^b \rightarrow S$: is the external transition function which specifies how the system changes state when an input is received, where $Q = (s, e)|s \in S, 0 \leq e \leq ta(s)$ is the total state set and e is the time elapsed since last transition. X^b defines the collection of bags over X. $\lambda : S \rightarrow Y$ is the output function that describes the output events according to the current state of the model and ta is the time advance function that controls the duration of internal transitions (during which the model does not change its state).

The DEVS specification structure is defined on a single component, called *atomic* DEVS model. In a co-simulation, the different *atomic* DEVS models are coupled using the *coupled* DEVS formalism (see Figure 3.5): it describes a DE system composed by *atomic* DEVS models as a network of coupled components. As defined in [103], a coupled DEVS N describes the structure of a system in terms of

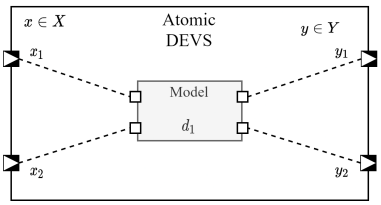


Figure 3.4: Simple example representing an atomic DEVS component.

interconnection between the atomic DEVS model. It is then defined as follow:

$$N =< X, Y, D, M_d | d \in D, EIC, EOC, IC >$$

where

$$X = (p, v) | p \in InPorts, v \in V_{X_i}$$

defines the set of input ports and values,

$$Y = (p, v) | p \in OutPorts, v \in V_{Y_i}$$

defines the set of output ports and values, D defines the set of models identifiers used to address each model using a unique identifier,

$$EIC = ((N, ip_N), (d, ip_d)) | ip_N \in InPorts, d \in D, ip_d \in InPorts_d$$

defines the set of External Input Couplings,

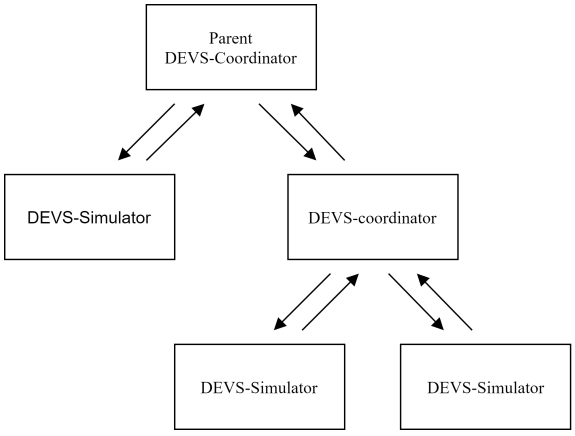
$$EOC = ((d, op_d), (N, op_N)) | op_N \in OutPorts, d \in D, op_d \in OutPorts_d$$

defines the set of external output couplings,

$$IC = ((a, op_a), (b, ip_b)) | a, b \in D, op_a \in OutPorts_a, ip_b \in InPorts_b$$

defines the set on internal couplings.

Based on these definitions, the closure under the coupling of DEVS allows to prove that an atomic model is equivalent to coupled model: this property enables hierarchical and modular modeling of DE systems. In particular, as shown in Figure 3.6, it is possible to compose hierarchical DEVS models (represented as DEVS-coordinators) to coordinate multiple enclosed DEVS simulators.



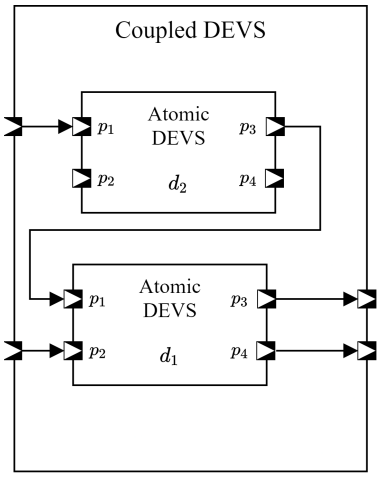


Figure 3.5: Simple example representing a coupled DEVS component.

Figure 3.6: Overview of the hierarchical simulators in DEVS.

An extension of the DEVS formalism for parallel execution is represented by Parallel DEVS (P-DEVS) [104]. The structure of a P-DEVS model is the following:

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

where X defines a set of input events, S defines a set of sequential states, Y defines a set of outputs events,

$$\delta_{int} : S \rightarrow S$$

represents the internal transition function,

$$\delta_{ext} : Q \times X^b \rightarrow S$$

defines the external transition function, X^b is a set of bags over elements in X with $\delta_{ext}(s, e, \emptyset) = (s, e)$, where $Q = (s, e)|s \in S, 0 < e < ta(s)$, and e is the elapsed time since the last state transition,

$$\lambda : S \rightarrow Y^b$$

specifies output function,

$$ta : S \rightarrow R$$

represents the time advance function.

Approaches that support P-DEVS proposed both sequential [105] and parallel [106, 107] coordinators for simulating atomic and coupled model, respectively. In sequential execution, each component executes sequentially its internal transitions: if internal transitions trigger external transitions on other models, all the transitions occur at the same simulation time [108]. The operational semantics of a simulator is described in [104] as an abstract simulator. The simulator computes the next state of the underlying coupled model until its end condition is satisfied. In particular, it computes the set of atomic DEVS models that have events on their event list; it then computes its outputs by executing the *output function* for each connected model. Output events are then available in the corresponding ports. Each atomic DEVS model must determine which transition to execute (internal or external). The simulator then executes in parallel all the transition functions (internal or external). Finally, each atomic DEVS computes its next internal transition, defined by its *time advance function*.

In parallel execution, such as proposed in [106], the operational semantics is defined by implementing a parallel abstract simulator based on the Chandy-Misra-Bryant algorithm [109, 110]. It guarantees a deadlock-free co-simulation and ensures the causality constraint.

Interface and Tools Discrete-event based co-simulation standards and tools are proposed to execute systems where events are first-class citizens. DEVS is one of the first event-based standards for co-simulation. DEVS proposes both a homogeneous interface event-based with which simulation units communicate and standardization for the Master Algorithm used to synchronize the execution of the co-simulation.

Another standard for real-time distributed co-simulation is the High Level Architecture (HLA) standard [71]. As DEVS, it proposes a homogeneous interface event-driven and the guideline to develop the master algorithm. However, it does not provide any implementation for both the interface and the master algorithm. Due to its distributed nature, the HLA standard is mostly supported by network simulators such as OMNET++ [111] and NS3 *.

Conclusion In our proposition, one of the main goals is to reduce to the minimum the number of required rollbacks during the execution of the co-simulation. The application of a conservative approach helps to reduce the overall required rollbacks using the *lookahead* concept that prevents the Cyber SUs to execute themselves ahead of time. Their execution will be then upper bounded by the safe timestamp at which they can execute without providing outputs or consume inputs. The details of this approach are presented in the next chapter 4.

* https://www.nsnam.org/

In the next subsection, we analyze the main challenges that we address in CPS Hybrid co-simulation.

3.1.3 Hybrid Co-simulation

Cyber-Physical Systems design requires to reason on the different interactions between environment (*e.g.* plant) and software (*e.g.* cyber controller). Different behaviors are exhibited by the heterogeneity of components: a cyber (or digital) controller can be represented using a DE formalism, meanwhile, the plat can be described using Ordinary Differential Equations. However, the complexity of the entire system, as the ensemble of plant, controllers, sensors, actuators, and software, is such that is difficult and impractical to detail it [112].

Systems that combine discrete-event and continuous-time systems are called *Hybrid Systems*. This notion is traditionally used by the control community to indicate discrete and continuous dynamics. In particular, a hybrid system has a continuous-time evolution and drastic changes. The changes correspond to the change of state in an automaton in response to external events [113]. *Hybrid systems* are often described using dedicated formalisms [112] and languages, such as Zelus [114] that mixes discrete-event and continuous-time behaviors by describing hybrid systems using a single language, for both expressing the physical models as Ordinary Differential Equations (ODEs) and cyber models as Data-flow formalism. In a co-simulation context, expressing models using a single language prevents the heterogeneity of the languages and tools used to develop CPSs but the white-box approach forces us to eventually disclose the IP of models. Our hypothesis on co-simulation is to use a black-box approach to *prevent* IP disclosure, making these languages not suited for co-simulation.

In this subsection, we focus on the co-simulation of black-box simulation units and on the main challenges due to the heterogeneity of CPS.

Hybrid Model Integration More and more, both CT- and DE-based co-simulation approaches proposed mechanisms and techniques to integrate the other formalism. We can divide the approaches into two main categories:

- CT-based co-simulation with all SUs encapsulated as CT simulation unit (*i.e.* a DE-based SU will use a CT-based interface to communicate with the master algorithm);
- DE-based co-simulation with all SUs encapsulated as DE-simulation unit.

Moreover, based on the main concepts of the co-simulation (the master algorithm, the interface, and the simulation unit), it is possible to further divide the approaches based on their proposition and the concerned level.

In the first category, every SU must conform to the CT-based co-simulation interface and, consequently, to the master algorithm semantics. Several approaches proposed techniques and mechanisms to integrate DE-based SU. For instance, [12] proposed to encapsulate every model not conforming to the CT formalism, into a CT-based SU providing a semantics adaptation between the inner semantics and the CT semantics.

In [115], they proposed to modify the CT-based master algorithm to take into account the internal time and events exploiting a dedicated variable. However, it needs to build a specific wrapper or modify the internal model in order to take

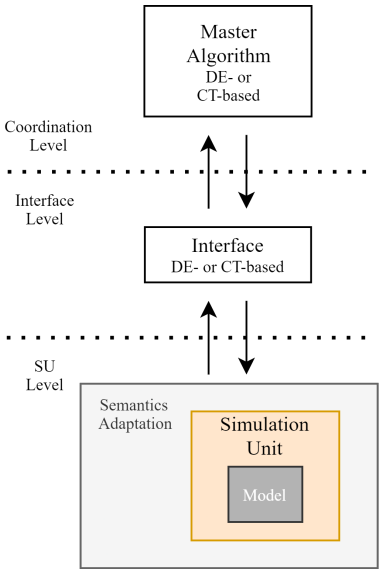


Figure 3.7: Main concepts of co-simulation: the master algorithm, the interface, and the simulation unit.

into account the semantics changes of the communication protocol between the interface and the master algorithm.

Other works, such as [11, 73, 116], proposed to adapt the interface by adding the support for different semantics directly to the CT-based interface. In this case, the semantics of the API change according to the semantics of the added API, and a dedicated master algorithm is needed to take this change into consideration. For instance, in [11], they proposed a dedicated master algorithm based on the different simulation unit semantics.

In the second category, all the SU must conform to the DE-based co-simulation. [117] proposed a DEVS&DESS (Differential Equation System Specification) [118] integration to embed Differential Equation-based simulation unit within DEVS simulations.

In our proposition, we address only the coordination level, without introducing any modification to the underlying interface or wrapping the simulation unit using a semantics adapter. This approach allows reusing existing interface standards (*e.g.* FMI, HLA, or DEVS) without requiring to provide a semantics adapter or change/update the API in use by the simulation unit. However, the generation of the master algorithm must take into account the heterogeneity of the system and the different semantics that compose it. The Model Coordination Interface is a dedicated interface to coordination used to expose a partial view of the syntax and semantics of the simulation unit in order to exhibit the minimal set of elements needed to define the master algorithm, as described in Chapter 4.

According to the DEVS semantic, the *getNextInternalEventTime()* function must return the time of the earliest scheduled internal event in the model. In the DEVS specification, the returned time corresponds to the minimum of:

- timestamp of the next internal event scheduled in a discrete-event component;
- timestamp of the current time plus the communication step size;
- timestamp of the next state-event.

The authors assume that it is possible to implement a rollback functionality to go one single integration step back. Getting the first two dates is trivial as they are a priori known. Things get more complex for the state-events: because of the numerical resolution of the ODE model, state-events can only be detected after each integration step of the FMU, and their localization in time can only be approximated. They perform an exploration of the FMU: thus, the FMU will be always ahead in time compared to the actual simulation time. When a state event is detected, they approximate the timestamp of the event by using a bi-sectional method. A similar approach is used by DACCOSIM [73] to find the time of a state change. Differently, DACCOSIM uses a sequential approach to the date of a state-change. A limitation of both solutions is that some state-changes might go undetected; for instance, if a Boolean value changes twice during the exploration, the change will not see its value as modified.

Event Location One of the differences between CT-based and DE-based co-simulation approaches is the role of an *event*. If we can agree on its standard definition where an event represents an action or occurrence that happens, its use in the co-simulation is different. In CT-based co-simulation, an event is triggered when a variable drastically changes its value, for instance when a discontinuity occurs, or the value has reached a certain threshold. An event is not considered as a first citizen of Continuous-Time but a behavior that can eventually occur

during simulation. On the contrary, in DE-based co-simulation, the event is the first citizen of the simulation. It cannot be considered as a rare occurrence but must be considered as ineluctable and common.

CT-based co-simulation approaches, mostly based on the FMI standard, propose to perform CPS co-simulation using a Time-Trigger Master Algorithm [11, 72, 73, 116, 119]. Such co-simulations involve both DE-based and CT-based simulation units.

A Time-Trigger Master Algorithm does not correctly support the integration of heterogeneous Simulation Units, as explained in the remainder of this section. Typically, CPS co-simulation also contains *Cyber* simulation units which are not based on Continuous-Time but rather on Discrete-Event like, for instance, simulation unit wrote using a Hardware Description Languages to describe digital hardware, or simulation unit written in General-Purpose Languages to describe software. These simulation units usually embed so-called piece-wise constant data where sampling creates bias (see Figure 3.9). As shown in [11, 18, 72, 120], such bias may invalidate the results of the co-simulation.

For example, Figure 3.10 shows the temporal evolution of a variable that changes its value instantaneously between two communication points. It is the case, for instance, if the model represents a Cyber component and it performs a variable assignment. The resulting trace shows that the change can trigger an associated event that will be detected only at the end of the step. From the external point-of-view (the point of view of the Master Algorithm and the rest of the models of the system), the event was triggered only at t_{i+dt}: the introduced delay between the instant when the event was triggered and the instant at which the event is detected creates an artificial delay due to the MA semantics used to communicate with the component.

A possible solution is to use the bisectional method [94] (Figure 3.11) to find an approximation of timestamp, or the smallest interval, at which the event was triggered. Like for other approaches. this method is intrinsically slow due to the several rollbacks needed to approximate the event location. By using this method, the compromise between the accuracy and the simulation time should be taken into consideration: if high accuracy is needed then the simulation time increases due to the time spent to locate the timestamp when the event was triggered.

While rollback mechanisms can be improved to reduce the time to locate the event [84, 121], other component behaviors can degrade performances. For instance, as shown in Figure 3.12, a Cyber model can have a periodic behavior in which a task is executed periodically. Between two activations of the task, the cyber model does not perform any activities or operations. Nevertheless, the semantics of the Time-Triggered MA imposes to keep the minimal communication step size imposed by the activity period: the communication between the MA and the model cannot take advantage of the specific semantic of the model to reduce the communication. An alternative can be the adjustment of the step size but the information of the inactivity period is usually hidden into the component and not accessible externally.

Modeling of Time The real-world phenomena happen as a continuous flow of behaviors. To describe these phenomena, the flow is then abstracted as time. The concept of time is an intrinsic element to describe physical phenomena behavior. To better describe the reality, physical models are usually based on a common

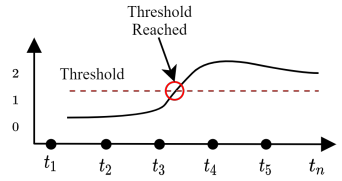


Figure 3.8: Event associated to the threshold reaching instant.

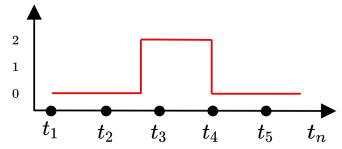


Figure 3.9: Piecewise-constant data.

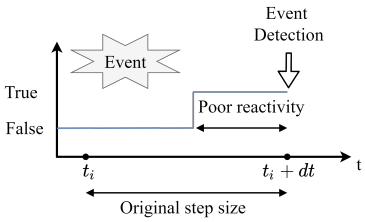


Figure 3.10: Event detection problem using FMI 2.0 Standard. Taken from [18].

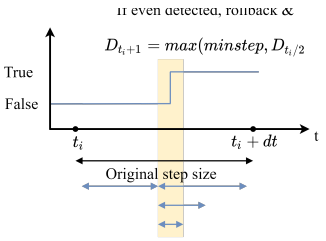


Figure 3.11: Rollback for event detection using FMI 2.0 Standard. Taken from [18].

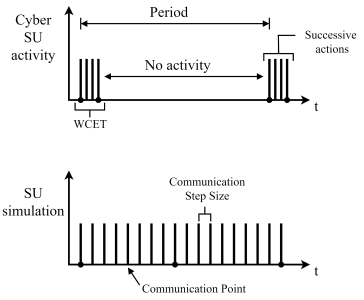


Figure 3.12: Communication impact on performance using a time-triggered API. Taken from [72].

resolution for the model of time: the Newtonian time. Time is represented as a Real number that advances uniformly in a non-decreasing way.

Unfortunately, Real numbers are represented using a floating-point approach. The main critical aspect of using such representation is appearing when performing arithmetic operations. [122] demonstrates how IEEE floating-point operations [123] introduce rounding errors. This is a particular problem, especially in hybrid systems where only Newtonian time is used, for equality tests: two floating-point values can result to be equal even if a ϵ value is added to one of the two [28].

Additionally, the Newtonian time is not well-adapted to simulate discrete-event systems. For instance, given two DE models that periodically trigger two events with the same period. The time representation should ensure that these two events will appear simultaneously to a third observer. Instead, the Newtonian time does not ensure the simultaneously of events and their determinism. The inner inaccuracy of the software representation of the Newtonian time introduces quantization errors [8, 28].

In order to mix Discrete-Event and Continuous-Time simulations, [124, 125] propose a model of time for hybrid co-simulation: the *superdense time*. A superdense time value is represented by a timestamp defined as a tuple (t, n), where t is the *model time* and n is the *microstep* number. The model time t defines the actual time at which an event occurs. The microstep n is then used to provide an index of the sequence of events that occur at the same time t. Two events happen to be *weakly simultaneous* only if given two tuples (t, n_1) and (t, n_2) have $t == t$ and $(n_1 \neq n_2)$. The order among events is then defined lexicographically [28]: $(t_1, n_1) < (t_2, n_2) iff (t_1 < t_2) \vee ((t_1 = t_2) \wedge (n_1 < n_2))$. An event is then represented by a tuple (v, ts) where v is the value of the event and ts the timestamp (t, n) at which the event occurs. This representation allows to add the notion of event at the Newtonian time, ensuring the causality constraint for events. Another approach proposes to overcome the problem of representing real numbers as floating-points, instead by representing the time as an integer [28]. However, this representation is not supported by the widely used co-simulation standard such as FMI [4] or DEVS [101].

Interfaces and Tools In the hybrid co-simulation context, Ptolemy II [14] is a modeling and simulation environment based on an actor-oriented approach for heterogeneous systems. It is based on the concept of Model of Computation which defines the underlying formalism of the simulation unit (*e.g.* Discrete-Event (DE), Continuous-Time (CT), Dataflow (SDF), or Process Networks(PN)). It addresses the heterogeneous model integration problem by providing a semantics adaptation between the different Model of Computations in the form of *directors*.

Conclusion In this section, we illustrated the main challenges in hybrid co-simulation. Several approaches and tools proposed to include hybrid co-simulation by supporting it using CT-based standards, such as FMI Standard, or DE-based ones, such as HLA. The limits of both formalisms cannot natively support the co-simulation of a heterogeneous system. Based on the proposed frameworks that categorize the approaches based on the level they addressed, we illustrate the state-of-the-art and the limits that we identified and addressed in this thesis. As shown in Figure 3.7, most of the approaches proposed solutions that We go further by proposing an approach that takes into account the semantics

of the simulation units by expressing a dedicated master algorithm on the specific scenario of the system and then, generating a parametrized master algorithm capable to exploit the exposed characteristics of each semantics to perform correct coordination. We will show in section 3.5.1 the meaning of *correct* in the context of hybrid co-simulation.

3.2 Domain-Specific Languages for Co-simulation

From the '90s, Software Engineering and System Engineering proposed several Domain-Specific Languages that focused on binding different computational entities together in order to perform a collaborative computation. They support the communication across different software processes by defining a coordination model (*i.e.* relationships among variables and data streams of the different models) [126].

A co-simulation can be seen as a heterogeneous software architecture where simulation units are represented by the software components, and the exchange of data among the simulation units is represented by the interactions between components.

In this section, we introduce some specific DSL used to describe the composition of a system. They were introduced in the context of studies about Software Architecture, known as Architecture Description Languages. We then illustrate some Coordination Languages used to define the behavior of the interactions that occur among components.

3.2.1 Architecture Description Languages

Software architecture aims to separate computation (*i.e.* components) and interactions (*i.e.* connectors) in a system. The architecture of a software system defines a high-level view as a collection of interacting components. More precisely: "*An architecture (of a Software-Intensive System) is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution*" [127]. Instead of describing the system using informal notations, such as boxes and arrows diagrams, to visualize the inter-relationships among the various components, the system software engineering community proposed well-defined formal approaches. For several years, the Software-Intensive Systems and Architecture fields implement ADLs to design software and hardware architectures [21, 128]. ADL specifications can then be used to generate executable models that enable design automation of tasks such as design space exploration, simulation, compilation, synthesis, test generation, and validation [129].

As identified in [25], such languages help to:

- Clarify structural and semantics difference between components and interactions;
- Reuse and compose architectural elements;
- Identify/enforce commonly used patterns.

The expressiveness needed to define an architecture resulted in several ADLs to share some basic elements, such as:

- *Components* : Encapsulation of a procedure, an object or a (formal) abstraction of its behavior;
- *Component Interfaces* : Explicit characterization of the component;
- *Connectors* : Behavioral description of the interactions among interfaces: a connector can represent a large variety of interactions (*e.g.* procedure call, event broadcast or database queries) of different complexity ranging from the identity function to complex protocol specification. [25].

Moreover, [130] defines three main criteria that an ADL must satisfy:

1. *decomposition* : each unique interface has only one corresponding component in the system;
2. *interface conformance* : each component must conform to its behavioral and syntactical interface;
3. *communication integrity* : each component must use connectors to interact with another component of the system;

In the following part of the section, we focus on the ADLs that respect these criteria, analyzing the basic elements (components, components interfaces, and connectors) and their role in these languages. In particular, we focus on the *component interface* and the *connector* elements.

Component Interface While the ADL community agrees on the definition of a Software Architecture as a set of components and connections among them, there are two styles of architecture [130]:

- **Object Connection Architecture** : It is composed by *object-oriented interfaces* and connections. An *object-oriented interface* is a set of features that must be *provided* by components conforming to the interface. A feature is a function that must conform to the name and signature specified in the interface. The component contains the actual implementation of the function. For instance, C++ classes are considered *Object-oriented interfaces*. The connection is then defined object-to-object: the direction defines which object uses (*i.e.* calls) the function defined on the other object;
- **Interface Connection Architecture** : It defines connections among components based on the interfaces which specify both *provided* and *required* features. A connection is defined between a *required* feature and a *provided* one. In contrast with the *Object Connection Architecture*, all the connections are between interface features. A mechanism to defined connections is then required: name matching (*e.g.* with a similar semantics as Hardware Description Language (HDL) for pin connections), or pattern triggered reactive rules, such as in Rapide [131].

Most of the actual ADLs are based on the Interface Connection Architecture. It constrains the components of the system to satisfy the three ADLs criteria of *decomposition*, *interface conformance*, and *communication integrity*. These properties allow using components that conform to the interface while keeping their internal behavior hidden (*i.e.* a component can be implemented as a *black-box* which exposes only the required and provided features thorough the interface, as in some co-simulation approaches such as FMI [4]).

Connector Usually, programming languages define interactions among different components by using procedure calls or accessing shared data. Connectors are types [21] that can be used to define domain-specific interactions. Approaches like AADL [132] or Clara [133] (to cite only two of them) proposed built-in connector and component types. Providing built-in connector types reifies interesting interactions, usually according to a specific domain. For instance, the Clara ADL is dedicated to real-time systems, consequently it proposed built-in connector types like *Rendez-vous*, *Mutex* or *Mailboxes*. This reification of coordination helps the coordination tasks by providing relevant domain-specific constructions. It also limits what can appear in the interaction and is a step towards providing domain-specific reasoning.

Other approaches introduced a notion of user-defined type [134–136]. These ADLs can then be specialized to a specific domain by the creation of domain-specific types. A connector type is usually defined by a set of roles and a glue specification. Roughly speaking, a role represents a formal parameter that is used by the specification of the glue. The glue specification specifies how the activities of the roles are coordinated. This glue can be specified more or less formally depending on the domain need. For instance, in Wright [135], the glue is specified in a variant of CSP [137], and in Reo [136] it is specified by the composition of dedicated primitives. The connector types are later on instantiated and the roles are bound to the actual interfaces of the instances of components. Other approaches define a connector as a component, such as the notion of "connection component" in Rapide [131], or a class, such as the notion of "Association Block" in SysML [138].

The Rapide language [131] is supported by a framework composed of four main elements: a typed language, an executable architecture definition language, the specification language, and the concurrent reactive programming language. The executable architecture definition language allows describing constraints and reactive rules based on events and their associated behaviors. The main feature of this framework is the possibility to use alternative programming and specification languages in cooperation with the type language. The heterogeneity of the languages on the specification of the coordination model can be exploited to compose a coordination model that is based on the syntax and semantics of the languages used to describe the internal component. This idea will be better illustrated in the next Chapter 4.

Conclusion The clear distinction between the computational aspect (components) and the interactions (connectors) allows the development of heterogeneous architecture in which the components conform to different languages and semantics. While the components are built by developers and system engineers, the specification of the interactions is defined by a system designer. In the first case, several general-purpose languages and domain-specific languages exist but, in the second case, the need for a formal specification has pushed the System Architecture Community to propose dedicated languages called ADLs. They identify in the *connector* the first citizen of the languages [139]. ADLs allow the definition of the interactions between two models by instantiating and binding connectors.

Moreover, from the modeling environment point-of-view, many of them are only textual and could appear less appealing for a software developer or system designer from other domains than only those for which the ADL was created [140].

The main limitation is found in the expressiveness of the behavior of the connector. Most ADLs provide only well-understood and formally defined connectors. Meanwhile, coordination languages allow system designers to define the behavioral semantics of connectors using given basic structures, such as in Reo [136], or using primitives to coordinate the overall execution in such a way as to bring out the actual coordination model [75].

3.2.2 Coordination Languages

In contrast with the ADLs, Coordination Languages focus on defining the semantics of connectors for execution purpose, with little support on specifying

the relationship between components and describing the overall architecture of the system. Conjointly with ADLs, Coordination Languages proposed to explicitly specify an executable architecture by specifying how models interact and by using an external model called, in this thesis, *coordination model*. Coordination language approaches focus on the development of parallel and multilingual systems by separating the computation concerns from the coordination concerns. They provide several dedicated models and languages for the specification of the interaction between different parts of the system. According to [126], "*Coordination is the process of building programs by gluing together active pieces*". In [141], authors highlighted that the design of coordination languages must address the following issues: identification of the entities to coordinate, architecture style of the coordination, and protocols/rules of coordination.

Coordination languages may handle run-time entities like processes, threads, actors, services, or agents. In this context, independently of the languages used to define a coordination model, the entity must exist and be implemented in order to build a coordination model. The entity may not be originally developed to be executed in a distributed context: the developer may or may not have used software best practices for a distributed environment such as critical section protections. Coordination languages provide a formal specification to concurrently execute it in a distributed context, with respect to its original semantics. In contrast with the *Interface and Object Connection Architecture* for ADLs, CLs target mainly entities that do not have any method or function that an external entity can call or execute.

Due to the fact that it was not meant for concurrent execution, an entity has not any methods to exchange data, for instance using a messaging pattern through send/receive method calls. It means that if an entity requires to communicate with external entities, it has to use blocking I/O operations on its variables. If we suppose that the evaluation of a variable implies reading its value, it means that in the original design the value exists at the time of the variable reading. Using a blocking I/O prevents the entity to process an absent value (*i.e.* dirty value), ensuring that the value is the most up-to-date and available (*i.e.* it is present at the time of the reading).

Similar to the *Interface Oriented Architecture*, CLs constraint the access to the variables as the exclusive way to access data from the external. The exposed variables are then called *ports*. For instance, in Manifold [74] a port is uni-directional and is part of the definition of the component while in BIP [76], in particular, the version with data-aware interactions [142], a port has a type and a list of data variables parameters. The port element plays an important role in the coordination model: a connector is based on the notion of ports for both data transfers and interactions between components. Both ADLs and CLs use the notion of connector: it represents an interaction that can be simple (*e.g.* assignment of value from a source port to a target port, procedure call, or event broadcast), or complex (*e.g.* definition of a communication protocol).

The main goal of coordination languages is to support a clear separation between the computational aspect and the communication concerns. However, not every coordination languages respect a clear distinction between computation and coordination aspects. Coordination languages can be classified by the level of separation achieved into two main categories:

1. *Exogenous* languages: A component does not contain any code that is used to define a specific protocol that coordinates its execution with other entities of the system;

2. *Endogenous* languages: The separation of concerns is not well-reflected at the level of the source code, a component uses primitives to perform inter-component communication from within the component.

For instance, Linda [75], integrates its provided coordination primitives into the source code using methods proposed by some libraries of the target component language that implement the Linda model *e.g.* C-Linda[†], and Javaspaces [143]. Exogenous languages lead to reusable coordination code, independent from the specificities of the component defined. For instance, Reo [136] provides primitives that define the coordination of the system outside the scope of the component.

Moreover, several style of models of coordination were proposed depending on the semantics of the computation: data-driven (*e.g.* Linda [75]), channel-based (*e.g.* Reo [136]), or component-based (*e.g.* BIP [76]). Our main focus is on the languages that support data-sharing or data-aware coordination models. In CPS co-simulation, data plays a central role during the execution but all the works done on coordination languages were not taken into account to improve the coordination model of the co-simulation. In our proposition, we want to take advantage of the studies and the main concepts on coordination languages to integrate them to better define a correct co-simulation coordination model. In the literature, we studied the main languages focused on data coordination. In particular, the two most important languages are Linda [75] and Reo [136]. The former is considered the ancestor of approaches that implements the idea of generative communication. This idea is based on the interactions among independent and autonomous components that can be expressed as a space orthogonal to computation [144]. This space is then reserved as a working space for the processes where it is possible to read and write data to or from other processes as a distributed system [145]. The latter enables a clear distinction between communication and computation models: communication should not rely on primitives used by components but defined as an independent communication model.

Linda [75] is the first coordination language based on a data-driven tuple-based approach. It has the notion of *tuple space* in which data, called *tuple*, are stored and shared among processes. *Tuple* is a record with typed fields and can be accessed using atomic operations for synchronization and data exchange. In particular, Linda is based on four atomic operations: `out` for writing, `in` for withdrawing, `read` for reading tuples in or out the tuple space, and `eval` for spawning new processes [146]. This model can be used to develop other forms of communication, like one-to-many broadcast operations or many-to-one aggregation, and synchronization, like semaphores or synchronizations barrier. The idea of using a shared data space to coordinate different activities can be found in the Distribution Data Service [147]. It is based on a publish/subscribe model for real-time systems, delivering the data at the right place at the right time. The data-flow is specified by the intent of publishers/subscribers to produce/consume only a specific type of data. The promoted decoupling in communication is due to the asynchronous approach which does not require any information about the sender, the tuple-space, and tuple insertion time. The shared data space concept allows components of distributed and parallel systems to read and write information as in a blackboard [145]. It enables collaboration among independent and autonomous software systems to achieve a so-called *generative communication*.

[†] `https://www.comp.nus.edu.sg/~wongwf/linda.html`

On the other hand, Reo [136] is a coordination language wherein graph-like structures express concurrency constraints among multiple components. These structures consist of a composition of channels and nodes. By using channel elements, it is possible to build a more complex link, called connector. A channel in Reo has exactly two ends, and each end either accepts data items, if it is a source end, or offers data items if it is a sink end. Each channel defines a well-defined behavior *e.g. sync* (a value is simultaneously propagated to every target and the operation is atomic) or *FIFO* (a channel represents a buffer with a certain capacity and the values are propagated asynchronously depending on the target behavior). Moreover, a channel has a type for its behavior in terms of a formal constraint on the dataflow through its two ends. The data-aware implementation is based on a Constraints Automata [148]. Each constraint defines a different basic behavior, allowing to specify synchronous and asynchronous transitions. One or more boolean condition on the data flow permits to exchange of data globally only if local constraints are met.

The language is still actively maintained and developed [‡]. The main benefit of REO is the connector expressiveness: using the elementary channels, it is possible to create connectors that take into account the semantics of the models of both ends. However, the construction of new connectors requires a deep knowledge of the semantics of channels to ensure that the resulting connector meets the given coordination requirements (*e.g.* propagate an input when an event is triggered by the simulation unit).

Another exogenous language is BIP [76, 149]. A BIP system consists of three layers: Behaviour, Interaction, and Priority. The behavior layer encapsulates all computation, consisting of atomic components processing sequential code. Ports form the interface of a component through which it interacts with other components. BIP represents these atomic components as Labeled Transition Systems (LTS) having transitions labeled with ports and extended with data stored in local variables [149]. The second layer exploits BIP interaction models to define the component coordination. For each interaction among components in a BIP system, the interaction model of that system specifies the set of ports synchronized by that interaction and the way data is retrieved, filtered, and updated in each of the involved components. In the third layer, priorities impose scheduling constraints to resolve conflicts in case alternative interactions are possible. BIP data-aware implementations is based on the *interaction expressions* [150]. It describes the control and data flow between a set of ports. The data flows only if a boolean condition is satisfied.

Lately, coordination languages have grown interest in their use in CPS co-simulation. The adoption of the FMI Standard [4] in the CPS community, has shown limitations especially for the configuration and integration of several FMUs. In an attempt to tackle this problem, the FMI consortium proposed a textual representation called System Structure and Parameterization (SSP) [151] which provides an XML schema of the system as a set of connections among FMU. It allows to express the topology of the system but the limited expressiveness of connectors does not allow the definition of any coordination model. Other approaches focus on giving a possible coordination model by defying the actual semantics required by each FMU to be correctly simulated. For instance, HintCo [152] framework proposed the Hint Language that uses *hints* to build the coordination model between two models by choosing and adapting statements in which behavioral patterns are defined on the exposed variables. Hints are then

[‡] http://reo.project.cwi.nl/reo/

translated into semantics adapters, having the same limitation of the semantics adaption approach illustrated in section 3.1.3.

Conclusion In this subsection, we have presented approaches that identified the need for the connector notion to enable the integration of heterogeneous components. Each approach proposes a DSL to define the set of interactions among models. The main advantages of the coordination languages are:

- Changing the model behavior or the connector semantics does not affect other elements of the system;
- The development of each component can occur concurrently and independently from each other;
- A connector can define a protocol that is independent of the actual implementation of the components connected;
- The enrich semantics of connector allows to specify and ensure the correctness of the system's execution.

Both ADLs and Coordination Languages are based on the concept of *connector* that explicits the interaction between two or more components. In this thesis, we focus on systems that implement an *Interface Oriented Architecture* since usually a co-simulation is composed of components that hide their internal behavior using a black-box approach.

As for the connector, for which each language proposes its own implementation with different capabilities, properties, and semantics, the interface used by each language reflects the purpose of each language. Moreover, different levels of abstraction can be provided, for instance, ADLs require a high-level description to represent the system architecture with support for the actual semantics of the interactions, for instance in Rapide [131] the semantics of connectors can be expressed using CSP [153].

To express a correct and soundness coordination model that takes into account the semantics of the models under coordination, a coordination language should allow using elements of the syntax and semantics of the language with which the model is defined. In particular, the coordination model must take into account the synchronization points at which, during the execution, the model can exchange data with the external environment (*e. g.* other models) in a safe way. For instance, in CT-based models, these points are identified by the communication points or, in DE-based models, these points are identified as the synchronization of events between two or more models. A possible solution to express synchronization points without the need to expose the entire model is through an interface. In particular, an interface must contain the minimal set of information needed to express a correct coordination model.

In the next section, we illustrate the role of interfaces in the co-simulation context.

3.3 Co-Simulation Interfaces

In the Model-Driven Architecture, components are abstracted as interfaces. An interface defines a set of properties and features that define a specification of the internal behavior. For instance, in an object-oriented interface, the interface defines which set of features (*i.e.* methods and attributes) the model must provide to be conformed.

Model-Driven Engineering encourages the re-usability of components in a plug-and-play way: a component should be easily substituted with another model with the same capabilities but with a different implementation. To achieve that, a coordination model should be defined between components using only the information on the interfaces.

In the context of CPS co-simulation and our hypothesis of black-box, it is important to find a balance between expressiveness and IP protection: information exposed on the interface must be such to express a correct coordination model while, at the same time, not too expressive to disclose the internal mechanisms and algorithms of the model.

In this thesis, we focus on two level of abstractions: *runtime* and *model*. At *runtime* level, the interface exposes attributes, parameters, and methods that are used during the actual execution; at *model* level, the interface exposes conceptual elements of the models such as properties, events, or ports.

In the next subsection, we illustrate the main concepts of the Runtime Coordination Interfaces for co-simulation. In subsection 3.3.2, we illustrate the Model Coordination Interfaces and their properties for co-simulation purposes.

3.3.1 Runtime Coordination Interface

During the execution of the co-simulation, the Runtime Coordination Interface (RCI) exposes a set of functions, properties, and attributes conforming to the semantics by which external entities can interact with the underlying model. A runtime behavioral interface is usually implemented by an Application Programming Interface (API). An API describes how to interact with the model by giving a set of methods and attributes that can be called or accessed at runtime.

In the context of co-simulation, two widely used standards aim to specify an interface that components must conform to: the Functional Mockup Interface (FMI) and the HLA (High Level Architecture).

The Functional Mockup Interface (FMI) is a standard that promotes a tool-independent co-simulation and model exchange of models or subsystems developed using different modeling tools. It was created within the ITEA2 MODELISAR project [4, 67, 68] and it is currently maintained by Modelica Association [154]. The FMI Standard proposes two different modes: **FMI for Model Exchange (FMI-ME)** and **FMI for Co-Simulation (FMI-CS)**. A component that implements the FMI is then called Functional Mockup Unit (FMU). It consists of a single zip file package with the extension "*fmu*". It contains three main elements, either for Model Exchange, for co-simulation, or for both:

- **Model Description**: A XML file contains information that is needed for model integration such as the declaration of the exposed variables of the FMU. These variables can then be used to get or set the actual internal

variables of the FMUs, according to the specified direction (*i.e.* input, output, or parameter);

- **C-functions**: A source or binary file must implement the functions defined in the standard. It provides a set of functions that execute model equations (FMI-ME) or setup and run the internal model (FMI-CS). In case of release as a binary, it is possible to embed platform-depend executable for each platform supported;
- **Documentation files**: Documentations, model icons, tables, or maps can be included in the package to support users. Other files such as libraries or DLLs dependencies can be embedded.

In the FMI for Model Exchange, the modeling environment exports the model as a C-code in a form of an input/output block. Only the model is then exported (without any dedicated solver) as an algebraic-, discrete-, or differential-equation description with time-, step-, and state-events [4]. The resulting model is then a passive FMU: an external solver must solve the set of equations inside the FMU (see Figure 3.13). The accessibility of the mathematical model of the FMU makes the ME approach unsuitable for exchanging models between different companies due to the disclosure of the IP.

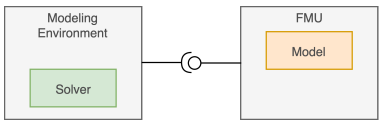


Figure 3.13: FMI for Model Exchange. Taken from [4].

Instead, in FMI for Co-simulation, the FMU is a black-box that acts as an active component: the FMU contains both the model description and an embedded simulation engine or solver (see Figure 3.14). In an industrial context, FMI for Co-simulation is a particularly attractive and useful approach that gives the possibility to embed compiled code into the FMU, keeping the source code in-house. It ensures to protect of the Intellectual Property of the model while sharing it with suppliers and vendors. The standard defines only the APIs that an FMU must implement and constrains on the order of calls defined as a state machine (see Figure 3.15), but it does not specify any algorithm, called Master Algorithm (MA), to coordinate the co-simulation.

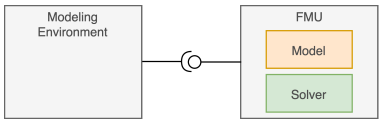


Figure 3.14: FMI for Co-simulation. Taken from [4].

The FMI interface is a C-based header file that defines the signature of the functions that the FMU must implement. Each of the two variants (FMI-ME and FMI-CS) has its own set of signatures defined on the specific requirements of each variant. However, the set of methods that handle the exchange of data from and to the internal component is in common:

```
1  /* Getting and setting variable values */
2  fmi2Status fmi2GetRealTYPE(fmi2Component c,
3  const fmi2ValueReference vr[], size_t nvr, fmi2Real    value[])
     ;
4  fmi2Status fmi2GetIntegerTYPE(fmi2Component c, const
     fmi2ValueReference vr[], size_t nvr, fmi2Integer value[]);
5  fmi2Status fmi2GetBooleanTYPE(fmi2Component c, const
     fmi2ValueReference vr[], size_t nvr, fmi2Boolean value[]);
6  fmi2Status fmi2GetStringTYPE (fmi2Component c, const
     fmi2ValueReference vr[], size_t nvr, fmi2String  value[]);
7
8  fmi2Status fmi2SetRealTYPE   (fmi2Component c, const
     fmi2ValueReference vr[], size_t nvr, const fmi2Real    value
     []);
9  fmi2Status fmi2SetIntegerTYPE(fmi2Component c, const
     fmi2ValueReference vr[], size_t nvr, const fmi2Integer value
     []);
10 fmi2Status fmi2SetBooleanTYPE(fmi2Component c, const
     fmi2ValueReference vr[], size_t nvr, const fmi2Boolean value
     []);
11 fmi2Status fmi2SetStringTYPE (fmi2Component c, const
     fmi2ValueReference vr[], size_t nvr, const fmi2String  value
```

```
    []);
```

As listed in Listing 3.1, the common functions allow to provide and retrieve the current value of inputs and outputs from the component. The functions are then specific to the actual type of the variable: for each allowed type in FMI, there is a corresponding function that handles all the variables with that type in the component. A major limitation imposed by the API is the limited expressiveness of types: more complex types, such as objects or tuples of values, cannot be natively used. Due to the implementation in C, the function takes an array of references and its size: a value reference is a unique ID that identifies an input or output in the component.

Listing 3.1: Excerpt of FMI standard co-simulation and model exchange API.

The function that is specific for FMI for Co-simulation is then listed in Listing 3.3.1.

```
/***************************************************
Types for Functions for FMI2 for Co-Simulation
****************************************************/
/* Simulating the slave */
typedef fmi2Status fmi2DoStep    (fmi2Component c,
fmi2Real       currentCommunicationPoint,
fmi2Real       communicationStepSize,
fmi2Boolean    noSetFMUStatePriorToCurrentPoint);
```

The *fmi2DoStep*, or abbreviated as *doStep*, method provides a homogeneous interface to ask to perform a simulation step of the underlying model. If the size of the simulation step is too large to be computed using a single integration step, the internal solver can perform multiple integration steps until the requested simulation step size. The FMI Standard does not provide any MA definition. Instead, it provides a formalization on the calling order of the functions (as shown in Figure 3.15) to constrain the calling sequence of the functions. In particular, the *doStep* function is responsible to perform a simulation step on the FMU, given a *communicationStepSize* or Δt (a non-negative real number). The FMU can then execute the step proposed or reject it if not compatible with its acceptable step size or an event internally occurs.

The Master Algorithm is in charge to orchestrate the simulation by calling the API functions for each FMU to advance their internal time, retrieve and provide outputs and inputs, and synchronize the internal time of every FMUs with the global time. In the literature several MAs were proposed, each one with some specific capabilities, for instance: ensure determinism and consensus of the size of each time step [4, 116], respect the error bounds of numerical approximation algorithms [84].

CPS co-simulation is composed of cyber and physical components. FMI supports natively the physical components but it lacks supporting cyber components with a dedicated API. To better support the concept of event, few works proposed an extension of the current API: [18] proposed two new functions called *fmi21DoStep(stepSize, nextEventTime)* and *fmi21GetNextEventTime(currentTime, stopTime, eventTime)*. The former allows the FMU to stop its execution on the first unpredictable event (*i.e.* asynchronous events) and return the *eventTime* timestamp at which the event was triggered. The later is used for predictable events (*e.g.* periodic events). Both methods enable the communication of the date of the next to know time event, enabling the MA to adapt the communication step size Δt accordingly.

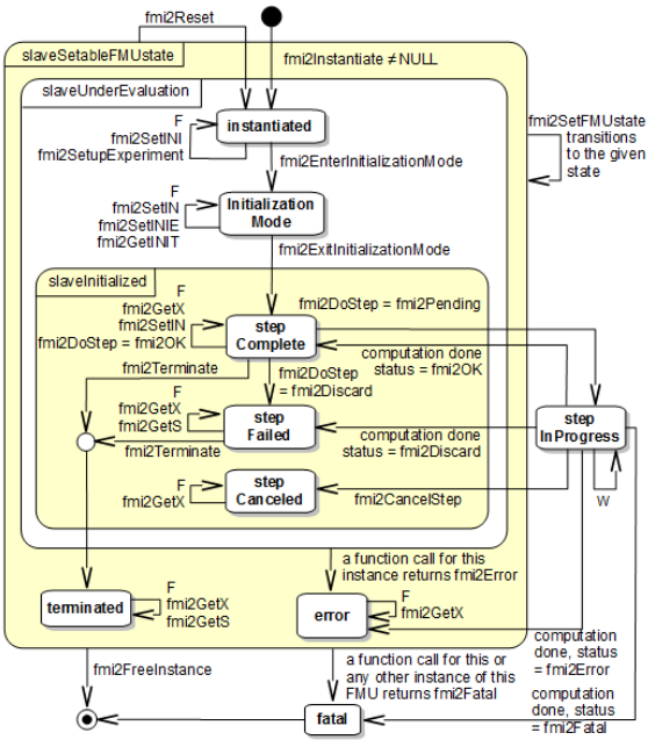


Figure 3.15: FMI for Co-simulation calling state machine. Taken from [4].

In [9], authors introduced in FMI the notion of *super-dense time* to locate the changes of a signal happening at the same instant, modifying the semantics of the *doStep* method to accept a $\Delta t = 0$. Then, they introduced a set of functions, *fmi2SetHybridXXX* and *fmi2GetHybridXXX*, to handle the event type. In this case, according to the definition of *event* in [8], which stated that events are instantaneous, they are able to retrieve events only if present during the execution.

Another popular standard for co-simulation is the High Level Architecture (HLA) [71], an IEEE standard for distributed computer simulation systems, originally created by the Defense Modeling and Simulation Office (DMSO). It is an event-based interface standard for software-centric co-simulations. Components of the distributed simulations are called *Federates*. The federates that cooperate together under guidelines and a defined object models form a Federation. Federates communicate with each other using a common infrastructure called Run-Time Infrastructure (RTI). The RTI represents a Federation execution backbone and provides a set of services to manage the communication the time and the data exchange between Federates.

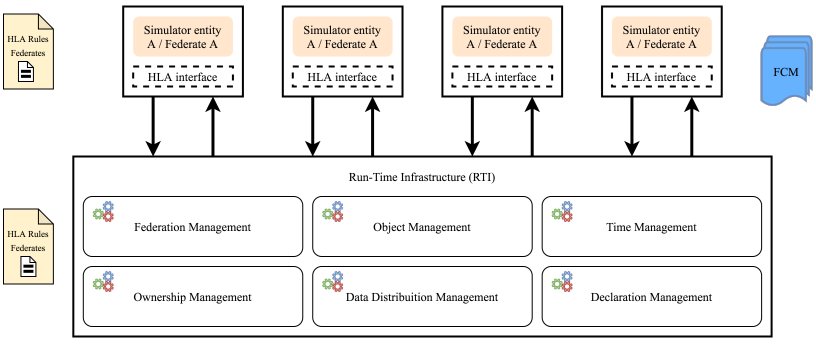


Figure 3.16: Overview on the HLA infrastructure. Taken from [155].

Federates interact using services proposed by the RTI (see Figure 3.16): Federation,

Object, Time, Ownership, Data Distribution, and Declaration Management. To interact with each other, federates can use a publish-subscribe approach, in which they send information to the federation to inform about an intention (*i.e. publish*), or receive information from other federates (*i.e. subscribe*). The content of messages is represented by objects. HLA defines two classes: an Object Class and an Interaction Class. The Object class contains the run-time data shared with the federation in the form of object-oriented data. The Interaction class data contains information on sending and receiving among federates. These objects are implemented using a XML format [156].

One of the current implementations of the HLA standard is Portico [§]. It is an open-source implementation of the Run-Time Infrastructure written in C++. In Portico, the Ambassador is the real interface between the model and the external environment. It means, the Runtime-Infrastructure represents the coordination engine and the federate/Simulator Entity represents the actual executable model. The RTI Ambassador and the Federate Ambassador are the two classes in charge to exchange data between the RTI and federates. So, the model has two ambassadors, one in the RTI and the other one for the federate.

As in FMI, HLA does not provide any coordination algorithm, and the architecture of the distributed system, letting its implementation be tool-specific. Differently from FMI, HLA does not provide any constraint for the actual implementation leading to different implementations that may not be interoperable and cannot be used in a co-simulation [103]. In fact, Co-simulation requires that different implementations can communicate together but each implementation has its own set of interfaces and communication protocols that are not compatible with other HLA implementations.

Some works proposed to integrate FMI and HLA: the FMI standard is based on a time-trigger master algorithm that is well-suited to co-simulation CT-based models, while the event-driven approach used in HLA is suited for DE-based models such as software. [155] proposes to integrate an FMU in an HLA Federation introducing a dedicated HLA component that acts as an orchestrator, which implements the Master Algorithm. This integration shows the same limit that the FMI has: events are managed only at the end of the time steps. In [92], authors proposed to integrate the telecom network simulator OMNET++ [111] with FMI, interconnecting FMU modeling continuous-time systems, but it used only constant time steps and the proposed Master Algorithm did not manage events between communication points.

In the discrete-event context, MECSYCO [106] proposes a Java/C++ API to support the Discrete Event System Specification (DEVS). In particular, they identified five methods to implement the DEVS model presented in the previous subsection (see Listing 3.2).

```
1  public void initialize();
2  public void processInternalEvent(double time);
3  public void processExternalInputEvent(String port,
     Event<?> anEvent);
4  public double getNextInternalEventTime();
5  public Event<?> getExternalOutputEvent(String port);
```

Listing 3.2: Runtime Interface signatures provided by MECSYCO to support the DEVS model interface.

The API is aligned with the definition of P-DEVS model, given in subsection 3.1.2:

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

[§] http://porticoproject.org/

Each input and output is identified with an unique string identifier called *port*. The $\delta_{int} : S \rightarrow S$ internal transition function is implemented by the *processInternalEvent*: for each internal state s, it must define which is the next state. By definition, every state has at most one next state, to prevent ambiguity, not allowing internal concurrency or nondeterminism [157]. The method *processExternalInputEvent* implements the external transition function $\delta_{ext} : Q \times X^b \rightarrow S$: is the external transition function which specifies how the system changes state when an input is received, where Q is the total state set where $Q = (s, e)|s \in S, 0 \leq e \leq ta(s)$ is the total state set, e is the time elapsed since last transition and X^b defines the collection of bags over X. It will process the incoming simulation timestamped event *anEvent* on the given port that has received it.

MECSYCO reflects the semantics in DEVS providing a DE API to encapsulate discrete-event based model. Some works proposed to integrate the FMI Standard, witch its time-trigger semantics, into the DEVS formalism to enable a co-simulation between DEVS-compatible models and FMI-compatible models: in [103], authors proposed to wrap the FMI model into a DEVS model that handles adaptation between the DE paradigm and the CT model. However, the proposition has the same limitations introduced in subsection 3.1.3: if an event occurs between two communication points, it will be localized in the upper communication point, and new inputs are taken into consideration only at the successive communication point, resulting in temporal inaccuracy that may invalidate the co-simulation.

In all the studied runtime interfaces, the API provides a specific set of methods and attributes that reflect specific semantics (*i.e.* only DE-based or CT-based). In the CPS co-simulation context, some approaches extended the API to support different semantics by adding new methods for each new semantics. This methodology increases the number of methods and attributes of the API by the number of different semantics composing the co-simulation scenario. Moreover, with the increasing usage of DSL in CPS development, adding and maintaining APIs could be time-consuming and difficult. In our proposition, we illustrate our approach to propose a general interface that exploits polymorphism to handle different semantics using a reduced API.

3.3.2 Model Coordination Interface

In software engineering, an interface can be used to restrict access to resources. For instance, the Application Programming Interface (API) of an operating system provides access to the underlying hardware avoiding direct access that can lead to malfunctions or instability of the overall system. In this case, the API abstracts the underlying behavior to expose a set of attributes and properties that defines how an external entity (*i.e.* in the case of the OS API, it is represented by applications or users) can interact with the underlying model. In a Model-Driven Architecture, the *Model Behavioral Interface* abstracts the behaviors of the model by providing a set of elements that defines how external entity (*e.g.* simulators, tools, or users) can interact with the internal model. Used in conjunction with MDE tools, it allows, for instance, automated analysis and verification, checking semantics properties detection of vulnerabilities, and generation of test cases.

Respecting the definition given in [130], a Model-Driven Architecture is composed of three main elements: *interfaces*, *connections*, and *constraints*. Interfaces represent the models of the system while connections and constraints define how the models may interact. Connections and constraints are then expressed using the

elements expressed on the interface. In this case, the Model Behavioral Interface can restrict access to its elements, providing only the minimal set of elements needed to define connections and constraints. The resulting interface is then called *Model Coordination Interface*.

For instance, the Coordination language Rapide [131] conforms to the Interface Connection Architecture: the coordination model relies on the information defined on the interface to specify connections among interfaces. In Rapide, an interface can be defined before the actual implementation of the component, allowing it to use as an early prototype. The abstraction of the component behavior is then carried out by the definition of an abstracted behavior on the interface itself. It specifies the set of features that component provides and features it requires from other components, and a set of *behaviors*, called *reactive rules*, that define how the interface should react to the received events. The expressiveness of the language used to define the reactive rules must take into account the inner semantics of the component. Usually, reactive rules are used to implement the behavior of a component without the need for actual implementation. Unfortunately, the specification of the component behavior on the interface may expose the internal component implementation which results in Intellectual Properties disclosure.

In the context of co-simulation, a more *protective* approach is adopted by the Functional Mockup Interface (FMI) Standard [4]. The proposed interface is composed of two components: an API C-header file, containing the signatures of the methods the internal components must conform to, and an XML model description file. The file, called `modelDescription.xml` contains static properties and information on the FMU and the model. In particular, the main elements it contains are:

- `DefaultExperiment`: It defines the property of the co-simulation execution such as start and stop time, and the preferred step size;
- `UnitDefinitions`: All the units used in the model must be previously defined;
- `ModelVariables`: The exposed variables are defined as `ScalarVariables`. Each variable is associated with its name, unit, value reference, causality (*i.e.* direction), and variability (*e.g.* continuous, discrete, or constant);
- `ModelStructure`: Used for equations based models. It defines outputs, derivatives, and initial unknowns that can be used to solve the internal model.

The FMI Standard 2.0 was designed to support a dynamic system that presents continuous and differentiable time signals. In the context of CPS, in which co-simulation requires the interaction of heterogeneous components conforming to different semantics, the behavioral semantics of the components may be exploited at the model-level to retrieve more information on the correct use of the component itself. In [18], the authors formalized different types of signals that may be present in a co-simulation with heterogeneous components. For instance, they identified hybrid signals that may be considered along with continuous and differentiable ones:

Continuous & piecewise differentiable The Figure 3.18 show a signal that is present at each time $t_i \in \mathbb{R}^+$, continuous in $\mathbb{R}^+$ but it is not differentiable in some points

$$t_i : \lim_{\varepsilon \to 0} f'(t_i - \varepsilon) \neq \lim_{\varepsilon \to 0} f'(t_i + \varepsilon)$$

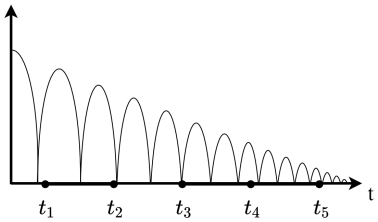


Figure 3.18: Continuous & piecewise differentiable signal. Taken from [18].

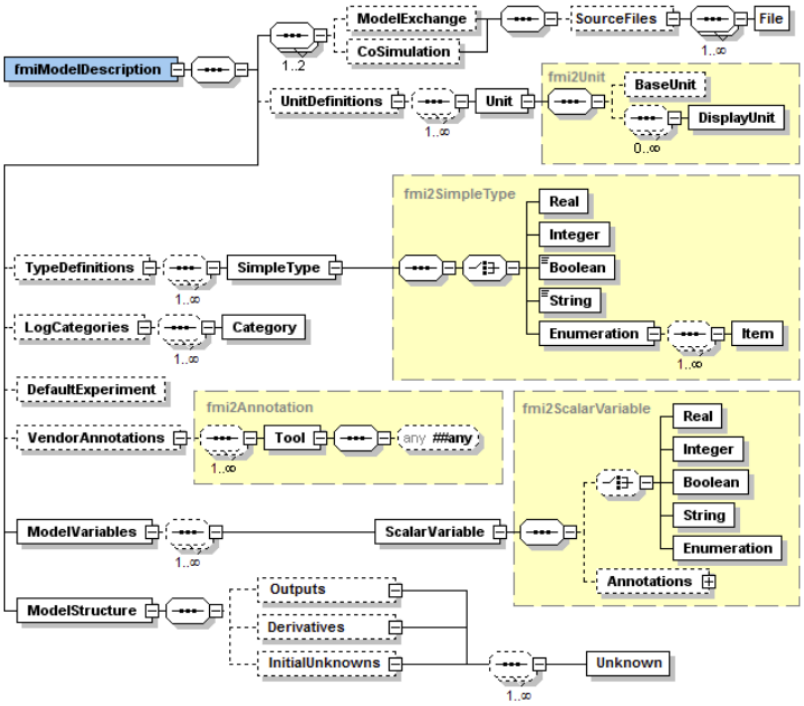


Figure 3.17: Overview on the Model Description Schema. Taken from [4].

Piecewise constant The Figure 3.19 show a signal that is present at each time $t_i \in \mathbb{R}^+$, constant on disjoint and continuous time slots I^i such that $\cup_i I^i = \mathbb{R}^+$ with a discontinuity appearing at each time slot switch.

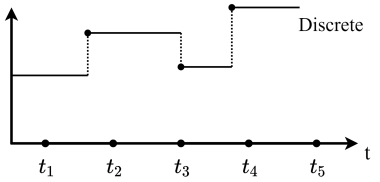


Figure 3.19: Continuous & piecewise differentiable signal. Taken from [18].

Piecewise continuous & differentiable The Figure 3.20 show a signal that is present at each time $t_i \in R^+$, not continuous in R^+ and not differentiable at some points

$$t_i : \lim_{\varepsilon \to 0} f(t_i - \varepsilon) \neq \lim_{\varepsilon \to 0} f(t_i + \varepsilon)$$

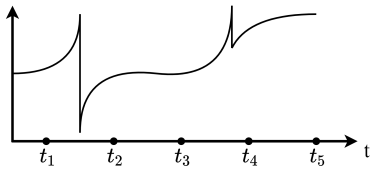


Figure 3.20: Piecewise continuous & differentiable signal. Taken from [18].

Discrete event The Figure 3.21 show a signal that is present for a set of definition D being a discrete time set $t_i \in D$ with $D \subset \mathbb{R}^+$. These discrete signals generate an *event* at each discontinuity.

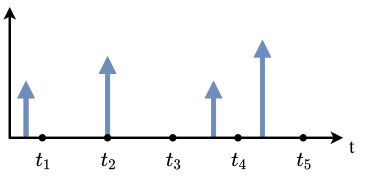


Figure 3.21: Discrete event signal. Taken from [18].

The study on the type, or *nature*, of signals was then used to improve the current FMI Standard 2.0, adding a set of C functions to support that new natures [18]. In particular, the authors were interested in capturing the value changes that could happen during the execution and exposing the change as an *event*. We must precise that the event that we are talking about is different from the discrete event that we defined in subsection 3.1.2: in fact, the discrete events that we find in DE-based models are not considered rare as in the case of [18] but they are triggered in a systematic way. Nevertheless, in [18], the nature of the signals was not abstracted as a property at model-level nor defined in the *modelDescription* file. In our proposition, we aim to expose through the Model Coordination Interface the *nature* of the signal, or more general of the *variable*, to partially disclose the underlying semantics.

In the DE system co-simulation context, one of the widely discussed model coordination interfaces is the DEVS specification (see in subsection 3.1.2). A DEVS atomic model is described using the 6-tuple structure

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

, while a DEVS coupled is defined as follow:

$$N =< X, Y, D, M_d | d \in D, EIC, EOC, IC >$$

These two tuples represent the model-level interface for a DE component. The interface is then a set of acceptable events by the internal model, constrained by the current state.

In other approaches, such as Ptolemy II [14], the model interface is composed of a set of external ports and parameters. A port can then be connected to the external ports of the model itself or to the ports of other models. External parameters are used to determine the value of the parameter's component encapsulated in the model interface. The coordination interface is then defined as a set of coordination points (*i.e.* Java methods []) that any models should provide. The resulting generic interface provides a homogeneous view of all the models, independently of their behavioral semantics, that can be exploited by an actor-based coordinator, called *director*. Each director implements a set of common Java methods (*e.g. initialize()*, *prefire()*, *fire()*, *postfire()*, *wrapup()*). The actual implementation of these methods obviously relies on the semantics of the model.

Approaches like [25, 158] proposed to reify elements of the behavior semantics within a behavioral interface. In particular, based on a formal representation of executable languages, the authors proposed to define a relationship between the *Domain-Specific Actions (DSA)*, that represent data and functions that represents and modify, respectively, the execution state of the model, and the *Model of Computation (MoCC)*, that represents causalities and synchronization of the system [25]. This relationship is represented by using a *Domain-Specific Event (DSE)*. The model interface is then composed by a set of *DSEs* and *constraints*. Based on that, the concurrency and time-related aspects are then abstracted and explicitly exposed. Moreover, in [159], authors go further by proposing a DSL, called *xDSL*, that allows to explicit a behavioral interface specification for any given language (that conform to the Ecore [59] metamodeling language). However, these approaches are strongly based on the concept of events and they do not have native elements that support the specification of data-aware coordination model or time-related coordination such as we find in CT-based approaches.

Conclusion In this section, we illustrated the main concepts and implementations for an Interface. In particular, two types of interface are discussed: model coordination interface and runtime behavioral interface. The Model Coordination Interface exposes a partial view of the internal syntax and semantics of components to enable the specification of coordination models based on its semantics-aware elements. We then illustrated some examples from Coordination Languages, CT-based co-simulation standards, and DE-based co-simulation formalisms. Each interface exposes elements of the internal model based on their domain: for instance, FMI exposes properties typical of CT-domain such as derivatives, variables, unknowns. Instead, the DEVS-based model interface exposes DE-domain related elements such as events and states. Domain-specific elements may be then used by a coordination model to define semantics-aware connections. These approaches are well-suited for co-simulation of models that conform to their semantics (*i.e.* the FMI standard implements a time-trigger semantics to handle CT-based models while DEVS or HLA implement event-trigger semantics to co-simulation DE-based models). However, if used with models that do not conform with the corresponding semantics, a semantics gap is introduced. To avoid it, it is required to expose through the interface the internal semantics

of the model while preserving the IP. This limitation will be tackled in the next chapter 4 by introducing a Model Coordination Interface capable to partially exhibit the semantics of the model using the *nature* of the exposed variables.

3.4 Distributed Co-simulation

A Cyber-Physical system is by nature composed of different parts, possibly distributed across devices, computers, or networks. Furthermore, even its co-simulation can be distributed across different simulators hosted on different computers or networks. It is important to understand the different topologies and mechanisms of a distributed system to deeply understand the impact of the coordination on co-simulation performance. A possible definition of a distributed system was given by [160] *"A distributed system is a collection of independent computers that appears to its users as a single coherent system"*. In a distributed co-simulation, several computers or, to be more precise, *Simulation Unit*, collaborate to achieve a consistent simulation. In co-simulation context, some approaches proposed to distribute the execution of the co-simulation across a network [41, 106, 119, 161, 162]. We analyzed the approaches based on the three main standards and formalism for co-simulation: FMI, HLA, and DEVS.

An important aspect of a distributed system is its organization. In this thesis, we used the definition provided by [163] to review approaches by the architecture they are using: centralized, decentralized, and distributed.

Centralized architecture Traditionally, in a centralized architecture, nodes are organized using a *client-server* topology (see Figure 3.22), where the central node, called *server*, stores data and serves the connected nodes, called *clients*. In a co-simulation, the role of the server is played by the simulation unit which replies to the requests of the *central coordinator or Master Algorithm*, which plays the role of the client. The connection between the server and clients represents the communication between the coordinator and simulation units.

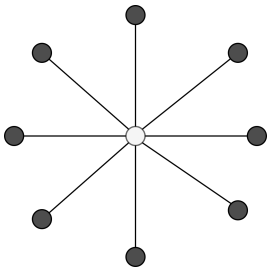


Figure 3.22: Centralized network.

Maestro [119] is an FMI-based framework that supports distributed co-simulation using a RESTful web service API, written using a combination of Java, Scala, and C. It was developed in the context of the INTO-CPS project [¶]. Maestro implements a Jacobi-based iteration Master Algorithm, called *Co-simulation Orchestration Engine (COE)*, to simulate the overall system [119], exploiting its property to execute continuous-time models in parallel. The co-simulation is controlled by the *VCS* and it is in charge to compute the next step size for the system under co-simulation. In case one of the FMUs discards the proposed step size H, then it must return the amount of the proposed step that was able to execute. The *VSC* then rolls back the other FMUs according to the retrieved value. The synchronization between CT-based FMUs and DE-based FMUs is carried out by exploiting the FMI API extension `getMaxStepSize`, proposed by [9], to get the largest step size that the SU can perform at that given point in time. However, this approach still requires rollback capabilities for certain SUs as they need to execute themselves to know their next possible step size. If a causally connected FMU requires a smaller step size, then the FMU needs to rollback its execution. Moreover, the required FMI extension is not part of the FMI Standard [4], delegating its development into the FMU.

Coral [41] is an FMI-based co-simulation software developed by the ViProMa project, developed in C++. It implements a centralized client-server architecture. The main element of the infrastructure is the *slave provider*, which runs on the machine along with the FMU handling its loading, execution, and exchanging data with the master. The *master* will then carry on the execution relying on the Coral software to communicate with the FMUs. The communication layer is

[¶] https://into-cps.org/

implemented using the ∅MQ middleware. The current version does not support any mechanism to support DE-based FMU co-simulation. The authors developed Coral because of the lack of solutions that meet their requirements: a model-driven framework, easy-to-use and that provide support for the main standards in the industry. These requirements were also addressed by our work by providing an open-source implementation that meets the requested requirement.

Differently from the previous approaches, the FMU-Proxy [161] provides an infrastructure to execute remote FMUs using the Remote Procedure Calls (RPC) paradigm. The RPCs are implemented by a proxy-server that wraps the FMU and exposes an FMI-CS API. The proxy-client controls the co-simulation by calling the remote procedures over HTTP and encoding messages with JSON schema. It allows defining an FMI Master Algorithm by using languages that support RPC. This approach results in a strictly centralized architecture where the RPC must be up-to-date with the underlying FMI-based SU and it does not support DE-based SU co-simulation.

Another communication middleware is the Distributed Co-Simulation Protocol (DCP) [‖] developed in the context of ACOSAR project [**]. It proposes a standard to distribute models of different platforms, focusing on the integration of real-time and non-real-time systems. It is composed of three main elements: a data model, a Finite State Machine, and the communication protocol. The communication protocol supports high-level protocols such as TCP, UDP, CAN, Bluetooth, or USB. Since it does not allow the definition of an actual communication protocol that can be implemented by a Master Algorithm, its use is dedicated to defining high-level interactions between real-time and non-real-time systems. The only open source implementation is *DCPLib* [††], implemented as a C++ library.

A centralized architecture allows to quickly implement co-simulation solutions. Several libraries and frameworks ease the setup, configuration, and execution of networked co-simulation relying on standards, such as FMI [4]. The two main drawbacks of a centralized approach are scalability and overall performance. The central node that acts as the coordinator can easily become the bottleneck due to the high number of messages exchanged with the nodes that host simulation units. Moreover, the FMI-based approaches may suffer the high number of messages exchanged on the network, as previously discussed in Section 3.1.3.

Decentralized architecture The centralized architecture shows its limitation using a single central node causing a bottleneck for the communication. A possible solution is to have multiple central nodes that handle a sub-network: the load will be then distributed across the multiple servers, allowing to potentially reduce the total load on each server (see Figure 3.23).

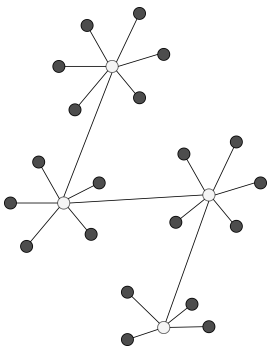


Figure 3.23: Decentralized network.

DACCOSIM (Distributed Architecture for Controlled CO-SIMulation) [73] is an FMI-based co-simulation framework based on a decentralized architecture. The interaction with the FMU is handled by the JavaFMI library [164], which provides a set of tools to work with the FMI Standard. The DACCOSIM NG version [162] is based on the same concepts and architecture as the first version [73]. As shown in Figure 3.24, the co-simulation is divided into multiple computational nodes. Each node has a local Master Algorithm that handles the co-simulation of the FMUs on its node. A global Master Algorithm is then in charge to synchronize the overall co-simulation among the local Master Algorithms. The use of a hierarchical

[‖] https://dcp-standard.org/
[**] http://www.acosar.eu/
[††] Last check in November 14, 2020 on https://dcp-standard.org/tools/

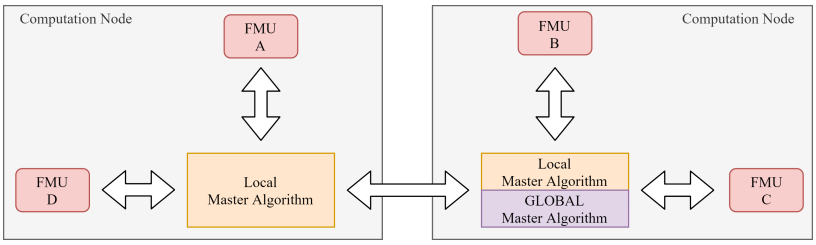


Figure 3.24: Overview of the distributed architecture provided by DACCOSIM.

approach aims to reduce the simulation time overhead due to the communication between two FMUs (that could be assigned on two different nodes). If two or more FMUs are closely coupled, they may require a high number of communication points to exchange data. To reduce the number of exchanges between two different computational nodes, DACCOSIM distributes FMUs according to their network use. Daccosim NG includes a DSL [162] to design the co-simulation topology by defining the relationships between the exposed inputs and outputs of the FMUs, and to configure the overall co-simulation (as shown in Listing 3.3). Based on this information, DACCOSIM distributes the FMUs, keeping the FMUs that communicate with a higher rate on the same node, as shown in Figure 3.24. The expressiveness of the connection specification is very limited: it is possible to define only the topology and the sample rate for each pair of FMUs.

```
1  FMU fmu_A "fmus/A.fmu"
2  Output fmu_A x2 Real
3  Input fmu_A x1 Real
4  FMU fmu_D "fmu/D.fmu"
5  Output fmu_D x1 Real
6  Input fmu_D x2 Real
7  Connection fmu_A.x2 fmu_D.x2
8  Connection fmu_D.x1 fmu_A.x1
9  CoInit 10 1.0E-6
10 ConstantStepper 0.5
11 Simulation 0.0 100.0
```

Listing 3.3: DACCOSIM DSL specification

The decentralized architecture improves the overall performance by distributing strongly coupled nodes closer. However, the performance may be inconsistent due to misconfiguration of the system or not properly optimized. For instance, the specification of the topology of the system does not take into account DE-based communications: the number of times an event is triggered is known only at runtime and cannot be always computed at static time.

Distributed architectures Centralized and decentralized architectures present two main disadvantages due to central nodes: throughput bottlenecks and single-point-of-failure. The central node is the central point of exchange of data and may suffer poor performance when heavily used by the rest of the nodes to communicate. Moreover, if the central node (or nodes) fails, the whole execution fails as well. The idea behind a distributed architecture is to avoid bottlenecks by eliminating any single-point-of-failure (*i. e.* central nodes). A node communicates directly with the interested node(s), without relying on a central node to exchange data with the others.

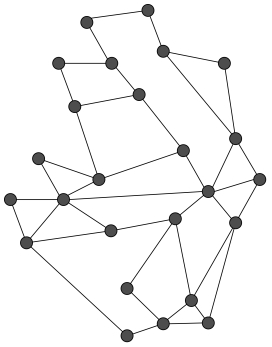


Figure 3.25: Distributed network.

DE-based co-simulation approaches such as HLA [71] standard proposed a native distributed architecture. However, the HLA standard does not provide any implementation of the Runtime Infrastructure, which must provide services to support the coordination of federate operations and data exchange. Several

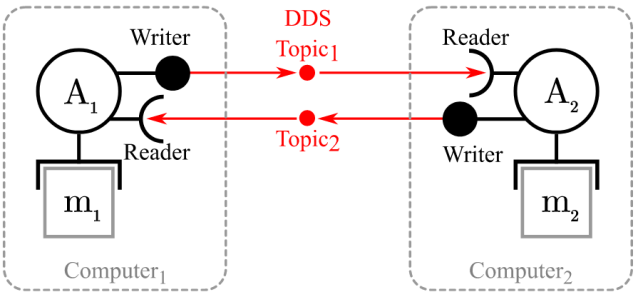


Figure 3.26: Overview of a MECSYCO co-simulation distributed system. Picture from [103].

works proposed their own implementations, supporting different languages and standard versions.

To the best of our knowledge, there is no FMI-based co-simulation approach that implements a distributed architecture. However, MECSYCO [106] implements a distributed architecture as a discrete-event framework which supports also FMI [103].

MECSYCO [106] is a DEVS wrapping framework for the multi-paradigm co-simulation of a complex system. MECSYCO is based on the AA4MM (Agents & Artifacts for Multi-Modeling) paradigm [165] to represent a heterogeneous co-simulation as a multi-agent system. In this context, the Simulation Unit is embedded into an agent. The data exchanges between agents correspond to the data exchange between executable models. Then, the dynamics of interactions among agents correspond to the actual co-simulation of the system. The current implementation is implemented in Java and C++ [#]. The messages are in the JSON format, in particular, they rely on the OpenSplice implementation of the OMG standard DDS (Data Distribution Service). To receive and send data, a coupling artifact relies on two components called *reader* and *writer*, as illustrated in Figure 3.26. Given the implementation using DDS, which uses a publish-subscriber pattern, the writer component plays the role of a publisher while the reader component acts as a subscriber. Each component defines a specific DDS topic on which to publish or subscribe for messages.

The parallel co-simulation is based on Chandy-Misra-Bryant (CMB) master algorithm [109, 110] that guarantees deadlock-free and ensures the causality constraint. Differently from the two architecture described before, in a distributed architecture each node implements itself the algorithm that we previously called *master algorithm*. It has a strong impact on the coordination of the overall system because, in absence of a single node, that plays the role of the master algorithm that orchestrates the co-simulation and where nodes are seen as passive entities, the "*master algorithm*" emerges from the interactions between nodes.

The main advantage of this architecture is the distribution of the co-simulation algorithm across each node (or agent). It overcomes the limitations of centralized and decentralized approaches removing the bottleneck represented by the central coordinator. Consequently, the communication load is usually distributed across links. However, some downsides should be taken into consideration, such as the distribution of the nodes is a key element for overall performance (*e.g.* strongly connected simulation units must be located as close as possible), and the implementation can be hard and difficult to deploy and maintain.

[#] http://mecsyco.com

Conclusion In this section, we discuss the three main architectures use to distribute a co-simulation across a network: centralized, decentralized, and distributed. A centralized architecture has the main benefit to be simple to develop and easy to understand the communication between the central node and the other nodes. On one hand, associated with the FMI standard to perform co-simulation, this architecture is well adapted to the development of master algorithms based on the Time-trigger mechanism used in accordance with FMI compatible simulation units. On the other hand, the main limitation regards the intensive use of the central node due to the transfer of data among nodes that must take place through the central node. A decentralized approach avoids the intensive use of the central node, by grouping strong coupled nodes, and using a hierarchical topology of master algorithms, optimizes the overall co-simulation performance. This architecture still has a central, event if decentralized in some parts, master algorithm. A distributed approach goes beyond a central master algorithm, by delegating the synchronization and data exchange to the nodes, and optimizes the communication between the various SUs by allowing a direct connection between two nodes.

In this work, we want to take advantage of the advantages of a distributed system, both in terms of decentralization of the master algorithm and optimization of the overall communication among nodes. Since a distributed system presents some challenges for its development and its configuration, in our proposition we present a framework capable of automatically generating the corresponding run-time infrastructure that actually runs the co-simulation. Moreover, the distributed master algorithm idea is improved in the proposed algorithm by the definition of scenario-specific coordination constraint (*i.e.* coordination model) between two or more simulation units in order to optimize the data exchange, and by introducing an extension to the concept of events capable to handle simulation unit with different semantics in addition to Discrete-Event or Continuous-Time Simulation Units.

In the next section, we conclude the state-of-the-art and discuss the three research problems that we address in this thesis.

3.5 Conclusion

In this section, we introduce the three research questions based on the limitations found in the literature. In particular, we propose for each section presented in this chapter a research question that addresses the problems highlighted before.

3.5.1 Correctness of Co-simulations

The major limitation of the current state-of-the-art approaches is that we do not have enough information on the simulation unit. It makes the specification of correct coordination difficult to achieve. As introduced in [166], in order to define correct coordination, the simulation unit needs to exhibit its capabilities and the usage of its input and output ports. For instance, using the FMI Standard, an FMU does not expose all the required capabilities to be correctly coordinated **but** the MA can ensure some capabilities needed to perform a correct co-simulation.

In this work, *we consider a coordination algorithm is correct if it does not introduce any delays or loss of information while interacting with the simulation units*. Consequently, delays and information loss that appear when using a time-triggered API on a piece-wise constant data are considered incorrect (see Figure 3.27). Three important things must be noticed at this point. First, sampling a piece-wise constant value can make sense and does not necessarily introduce a major problem; however, this should be done on purpose and not be the result of an inappropriate API. Second, there exists in many API (e.g., the FMI standard [4]) the possibility to avoid such delay, typically by roll-backing the simulation to a previous state and trying to locate the actual value change. This can be done only if the simulation can actually be rolled-backed; also this is costly in terms of simulation-time. Finally, third, it is worth noticing that the problem is broader than the simple illustrative case. As illustrated in [17], the coordination algorithm can have an impact on the correctness of the system.

The core of the problem was identified in several papers: it is not appropriate for any simulation unit to communicate only through a time-triggered or event-triggered API. In the literature, some approaches have proposed to extend some existing APIs to fix a particular problem. This was for instance the case in [18] where they proposed to add a new parameter to the FMI time-triggered *doStep(Δt)* function. The new parameter is $nextEventTime$, a placeholder to store the time at which unpredictable events occurred. [11] went further by proposing to extend the FMI API with new functions that simulate until input and output ports are respectively ready to be read or just written. Finally, the new features of FMI3.0 for hybrid co-simulation tries to aggregate such propositions (see chapter 5

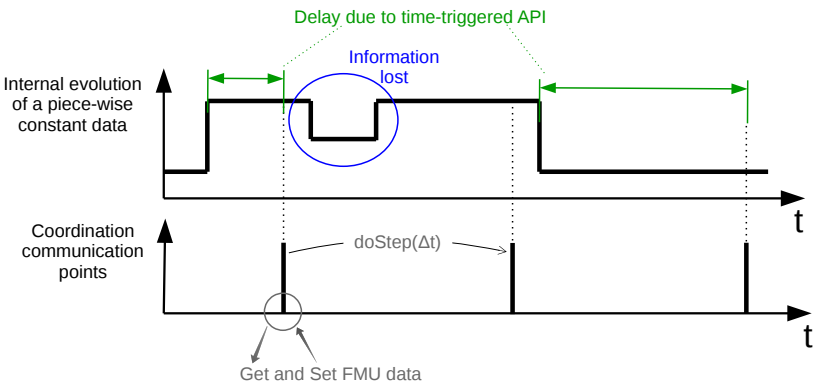


Figure 3.27: Temporal inaccuracy and delays of a TT co-simulation coordination with a DE SU.

of FMI3.0 development version https://fmi-standard.org/docs/3.0-dev/#fmi-for-hybrid-co-simulation).

However, in all these related works, the problem is not handled in its generality and they make specific cases of something that should be straightforward. In order to speak *correctly* with a simulation unit, you should be aware of its behavioral semantics and adapt the way to realize the *doStep* accordingly.

3.5.2 Research Problems

The collaborative simulation focuses on the orchestration among different simulation units that represent different parts of the same system, in order to better understand the emerging behavior of the system. A simulation unit is a, usually black box, executable entity, which may for instance encapsulate a model and its solver, a binary executable process, or a proxy to a hardware device. The orchestration is of prime importance because it defines the instant when a simulation unit exchanges data with other simulation units, *i.e.* when it synchronizes its internal time in order to produce or consume data at the right timing.

Consequently, several orchestrations (mainly based on the FMI Standard) were proposed. A majority of them are variants of well-known Jacobi or Gauss-Seidel methods and dedicated to (continuous) system of equations [6, 7, 9, 15, 90–92]. All the proposed algorithms implement time-triggered coordination that does not correctly support the integration of heterogeneous simulation units as found in Cyber-Physical Systems. Typically, CPS co-simulation also contains *Cyber* simulation units which are not based on continuous-time models but rather on discrete-time/ discrete event models like for instance models specified in a Hardware Description Languages to describe digital hardware, or models specified in General-Purpose Languages to describe software. These simulation units usually embed so-called piece-wise constant data where sampling creates bias. For instance, Figure 3.27 shows that time-triggered sampling can create either information loss or delays. As shown in [11, 17, 18, 72, 120], such bias may invalidate the results of the co-simulation. Consequently, we consider a coordination as *correct* when not timing bias like the ones of Figure 3.27 are introduced.

Research Question 1 *What coordination algorithm can be used to study emerging behaviors of a heterogeneous CPS without introducing timing bias?*

The problem in Figure 3.27 is not really related to the sample rate but rather to an inappropriate communication schema between the simulation unit and the coordination. One can increase the sample rate but it decreases the performances and reduces the delays without removing them. Also, when the possibility is offered by the co-simulation framework, one can roll-back the simulation units to a previously saved state and try to locate when the value actually changed; this is, however, (when available) a costly procedure that decreases overall performances.

The main problem comes from the black-box nature of simulation units, which does not allow to determine the correct communication schema between a simulation unit and the coordination. To thwart such problem, several works on component-based architecture description languages and coordination languages proposed to exhibit in an interface partial information from the (black box) components [21, 22, 167]. For instance, in the context of event-based systems

[131] relies on an interface for coordination, composed by the set of acceptable events, to define the synchronization relationships among components. Such interface gives enough information on how to coordinate the component without disclosing the internal model, *i. e.* without problematic IP violations. In the context of CPS, we addressed in this thesis how to exhibit, in a coordination-specific interface, an abstraction of the simulation unit behavioral semantics which is suitable to define an appropriate communication between the simulation unit and the coordination.

Research Question 2 *What is the abstraction of the behavioral semantics of simulation units which is sufficient to specify an appropriate communication schema between different simulation units without disclosing their intellectual properties?*

Then, the exposed behavioral semantics and syntax can be exploited to define a coordination algorithm. Even though coordination languages, such as [22, 76, 136], proposed sophisticated techniques to define correct and efficient coordination among software components, their focus on discrete-event coordination makes the approaches unsuitable to the coordination of CPS simulation units.

Additionally, as illustrated in [6, 12, 168] and by the *glue* proposed in [76, 131, 136], defining an appropriate communication schema is specific to the interaction between two simulation units and can not be inferred only from the formalism(s) initially used to specify the two simulation units. It implies that a dedicated interaction model must be provided for each set of coupled simulation units. Finally, in order to simplify the co-simulation process, the definition of a coordination model that takes advantage of the information defined on the simulation units should be amenable to the automatic generation of the co-simulation execution.

Research Question 3 *What language constructions are required to define a correct coordination model between coupled simulation units, so that it can automatically result in a distributed co-simulation?*

The next Chapter addresses these three research questions and describes our proposition. In section 4.2, we address the research question 2 by proposing a Model Coordination Interface that exposes a partial view of the syntax and internal semantics of the simulation unit and reduces their expressiveness for coordination purpose. In section 4.3, we address the research question 3 by proposing a Domain Specific Language that exploits the information defined on the interface to specify a coordination model specification tailored to the exposed syntax and semantics of the simulation units. Finally, in section 4.4, we address the research question 1 by giving a distributed coordination algorithm used for the actual co-simulation. The algorithm is parameterized by the previously defined Model Coordination Interfaces and Model Coordination Specification.

Proposition 4

4.1 Introduction

In this thesis, we want to take into consideration the three aspects previously identified and ease the writing of correct coordinators based on the semantic properties of simulation units. To remind the reader, it appears that:

1. the writing of the coordinator is more and more complex because it must take into account the characteristics of the data it conveys;
2. the characteristics of the simulation units, the behavioral semantics of the used languages, and their implementation by a solver are not always accessible;
3. the topology between the different interconnected simulation units is usually not exploited.

Considering all of these aspects are often neglected but it has been shown to condition the correctness of the co-simulation. Inspired by works on the Architecture Description Languages, discussed in subsection 3.2.1, Coordination Languages, discussed in subsection 3.2.2 and heterogeneous frameworks [23–26], we propose to define a Model Coordination Interface that exhibits enough information to ensure that the interaction between different simulation units can be correctly specified. We also propose a structured dedicated language to specify the different interactions between the simulation units. From such description, it is possible to automatically generate an appropriate coordinator (*e.g.* Master Algorithm in FMI [4]) and its underlying co-simulation infrastructure.

The characteristics of each simulation unit are key elements to define the interactions with the other simulation units. A possible solution is to exhibit them through an interface tailored to the semantics of the underlying model. In contrast with the white box *Model Coordination Interface* defined in frameworks like Gemoc*, this proposition exposes only a partial view of the syntax and semantics of the simulation unit. The interface is tailored specifically to share only the elements necessary to coordinate the execution and communication among the simulation units. The goal is to help to protect the Intellectual Property contained in the simulation unit. We proposed to provide a higher abstraction level about the semantics to ease the reasoning. We divided the interface into two main elements:

- *Ports* defined the exposed variables of the simulation unit and their attributes such as name, type, direction, and data-nature;
- *Temporal reference* used by the model representing which temporal referential the model is using. In the context of heterogeneous systems, each model can be simulated using a different temporal referential *e.g.* time, angle, or distance. For example, in a fuel engine, the camshaft angle is used to define when to open/close valves or to actuate the fuel pumps. The execution semantics for this model is not based on time but on the

* http://gemoc.org/

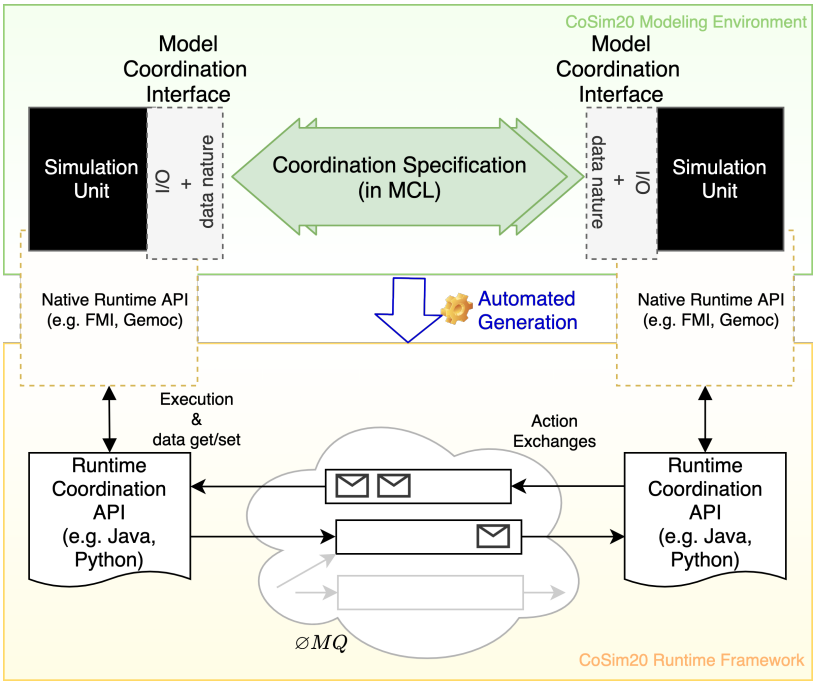


Figure 4.1: Overview of the proposition.

angle. The actions depend on the movement of the camshaft and not on the physical time;

- *Set of simulation unit properties* denoting SU-wide behaviors (*i.e.* rollback capabilities or saved states which allow restoring a previous state if necessary). On one hand, these properties enable the generation of a more sophisticated that takes into account some additional properties of the simulation unit. On the other hand, we do not yet have exploited these properties in the current version of our implementation.

The most important element of the interface is represented by the data-nature. It characterizes the value of the port according to its behavior during the execution of the model. Thus, different natures have different behaviors during the execution and then the coordination needs to take them into account. By using such information, the designer can decide the way to coordinate different data. For instance, the way to coordinate *transient* data (*e.g.* ported by an event) is handled differently than the way to coordinate *continuous* data (that may be sampled periodically). Other information is added to the data according to their nature. For instance for *piecewise-continuous* input data, an annotation is added to specify if internally, the executable model is using interpolation or extrapolation (like explained in [12]). Another example of such annotations are *events* associated to *piecewise-constant* data, like for instance the $update$ event represents the instant at which a data is assigned or the $readyToRead$ event represents the instant before the data is actually read[†]. Based on the primitive ones, *e.g.* Double, Integer, it will be possible to extend the interface allowing the definition of new complex data types, *e.g.* Electric Current, Acceleration.

The resulting interface provides enough information to ensure that a system integrator has the knowledge that is required to correctly coordinate the different executable models, as studied in [27]. Simulation units enclose as black-boxes different models that conform to modeling languages with different syntaxes and semantics. To explicit the semantics without *opening* the black-box, we need to abstract it as a property on the interface. Based on the abstracted semantics,

[†] This is directly inspired from [11, 158]

the system integrator must relate the different semantics of the simulation units to determine how to integrate them. We introduce a specification to express the relations between simulation units as coordination models.

The Coordination Model Specification explicitly defines the interactions and constraints between different Coordination Interfaces. It is composed of a set of *Connectors*. Each connector is in charge of defining *when* and *how* one or more data are conveyed from a model to another. In order to define interactions, we structured a connector through three different elements:

- *Interaction* specifies how data are actually exchanged between ports and which transformation should take place here *i.e.* unit alignment between Meter and Feet;
- *Triggering Condition* defines the instant at which the interaction must be realized. The instant can be expressed as a periodic condition (*e.g.* every 5ms) or as an aperiodic condition (*e.g.* event, internal variable usage [11]);
- *Timing Constraint* specifies a relation between the temporal references of the models.

To validate our proposals, we have developed an Integrated Development Environment (IDE) to support the specification of Model Coordination Interfaces (MCI) and Model Coordination Specifications (MCS). For this reason, we proposed two Domain-Specific Languages (DSL) called MCILang and MCL to define the MCI and MCS respectively. We provided a formal syntax, both abstract and concrete, using Ecore and Xtext, respectively. We then provided a textual and visual environment with few facilities such as auto-completion, connection helpers, and an FMI 2.0 compatible importer. Based on these DSLs, we provide an automatic generation process to create a dedicated coordinator based on the specified system. In particular, we developed a framework capable of coordinating several distributed entities without the need for a centralized coordinator. At runtime, this framework encapsulates the SU, according to its MCI, with a dedicated wrapper tailored to its semantics. The wrapper contains a semantics parametrized API to handle the execution of the model and a communication module to interact with the rest of the system.

We discuss our proposition starting from the introduction of the model coordination interface in section 4.2, we illustrate our model coordination specification in section 4.3, and then, finally, in Section 4.4, we introduce the proposed distributed coordination algorithm and its automatic generation.

4.2 Model Coordination Interface

In a co-simulation environment, we can identify two main execution phases: in the first one, the defined coordination model is applied and determines which model has to execute; the second one, the model executes according to its internal semantics. In the last phase, we can identify the need to interface with the model to get information on its internal semantics, on the exposed variables that the environment has access to, and on the required communication point, at which data propagates between the model and the environment. The goal of the proposed interface is to answer the research question 2:

> *What is the abstraction of the behavioral semantics of simulation units that is sufficient to specify an appropriate communication schema between different simulation units without disclosing their intellectual properties?*

We were inspired by existing approaches, such as the FMI standard [4] and ADL languages, as we discussed in subsection 3.2.1, for structural information (*e.g.* exposed variables and their types). We then adapted the *data nature* concepts proposed by [120] to add semantics to each variable that better fits the coordination purpose of the API. The semantics reflect the inner semantics of the simulation unit. The proposed Model Coordination Interface (MCI) consists in a set of *ports* and their direction (input or output), a *temporal reference* and a set of *simulator capabilities*. It is interesting to notice that each *port* represents a point of interaction between the internal variable of the model and the external environment *i.e.* the coordination algorithm. A port is then categorized using a data-nature that defines different ways to exchange data. The Model Coordination Interface allows defining in the interface the correct way to handle the data. For instance, if a *piecewise-constant* port is handled using time-trigger coordination, two errors can occur: the data are delayed in time according to the size of the step, or if the value changes during the step size, the change is not registered and the updated value is lost. Exploiting the data nature of the port, we can then define the more appropriate interaction semantics and give the expected interaction.

4.2.1 Port

Inspired by the existing interfaces and components, the set of *Ports* is the main element of the Model Coordination Interface. It provides a well-defined and formal abstraction to communicate with the underlying SU. A port is an abstraction of the variable value and its properties such as its identifier, direction, and type, but also co-simulation properties such as initial value or the SU internal variable name.

The direction property can assume the values Input *e.g.* the SU can read the value assigned but it is not allowed to write to it, or Output *e.g.* the SU is allowed to make assignments (writings) to it but it is not allowed to read from it. A bi-directional port (*i.e.* assignments and readings are allowed to such a port simultaneously) is not supported since the current co-simulation tools do not support it at runtime. For instance, it is not possible to define it using VHDL [169] or Modelica [170].

By looking at the various existing approaches and by developing various systems, we realized that the information which is important from the coordination point of view depends on how and when the inputs and outputs are internally used (*i.e.* read and/or write) by the simulation unit. Such information is strongly linked to the *data nature* of the simulation units inputs/outputs. The different natures of

the data found in CPS have been introduced in several approaches but are nicely summed up in [8, 120], illustrated in subsection 3.3.2. We then introduce the notion of data nature as a property of the port and we adapted such specification ending up with the data nature defined in the next subsection. In particular, the data natures proposed by [120] were used to define an associated event at each discontinuity of the signal and which kind of signal is compatible with their proposition. Otherwise, we propose the data nature as port property to reflect the inner semantics of the signal and not only if it can support an event-based API. Exposing the data nature of a variable enables it to be exploitable outside the internal model, in particular for coordination purposes. We illustrate how is possible to exploit it in the next subsection.

4.2.2 Data nature

By looking at the various existing approaches and by realizing various systems, we realized that the information which is important from the coordination point of view depends on how and when the inputs and outputs are internally used (*i.e.* read and/or write) by the simulation unit. Such information is strongly linked to the *data nature* of the simulation units inputs/outputs. The different natures of the data found in CPS have been introduced in several approaches but are nicely summed up in [8, 120]. We adapted such specifications and ended up with the data nature defined later.

The data nature defines how data is internally used in simulation units, independently of the internal model itself. It provides partial information about the behavioral semantics of the simulation units with respect to a specific input or output.

Before explaining how we used them, a description of the different data natures supported by the proposed MCI is provided:

Continuous A variable is defined as *continuous* if its value is present for all $t \in T$, continuous and differentiable at any points of its range of definition *e.g.* the value of the physical temperature of a room. The continuous data nature is usually associated with nonlinear systems' model *e.g.* Ordinary Differential Equation. In case the data nature is *continuous*, then there is no real coordination point in time of interest. What is important is to sample is data fast enough to avoid losing quick variation (*i.e.* Nyquist-Shannon law should actually hold).

Piecewise-continuous A variable is defined as *piecewise-continuous* if its value is present at each instant but it is not continuous and not differentiable at some points. For instance, two classical sources of discontinuity are a multi-mode model [171] and internal discontinuities due to the differential equation system *e.g.* Bouncing ball. In addition to the previous case, if the data nature is *piecewise-continuous*, then it may be important to be noticed of any discontinuity in the signal.

For instance, a representative example is a Bouncing Ball (Figure 4.2). Every time the ball touches the floor, a discontinuity appears: from a coordination point of view, this information can be then used by the system integrator to define a coordination model that takes into account only the discontinuities occurring without asking the most updated value all over the parabolic curve. However, the coordination model depends on the scenario we want to define. For instance,

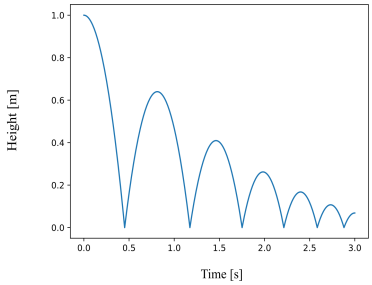


Figure 4.2: Example for the datanature *piecewise-continuous*: a Bouncing Ball trajectory.

in the case the simulation unit representing the behavior of the bouncing ball must be integrated with a second SU that reacts every time the ball reaches its maximum height at each bound, then the coordination model must change accordingly. The system integrator must then sample the signal with the most appropriate sample rate to detect the maximum height, with a certain degree of inaccuracy.

Piecewise-constant A variable is defined as *piecewise-constant* if its value is present at each instant but it presents discontinuities and the value is constant between two discontinuities.

Associated with the direction of the data, the data-nature as defined above gives information on important points in time at which coordination should be done. Typically, if the output of a simulation unit is declared as *piecewise-constant*, then the coordination model is interested in the points in time at which data are written; so the corresponding value can be updated in the connected simulation unit input port(s). On the other direction, if it is declared as an input, the coordination model takes into account the points in time at which data are read. The last case is quite interesting because it enables the coordination of the I/O operations of a simulation unit with the rest of the system, according to its internal semantics.

This type of data-nature enables to define on the port specials events that represent *interesting coordination instants* on which the model can synchronize its execution to exchange data from or to the external [11, 158].

Algorithm 3 *Piecewise-constant* generator example.

```
1: while true do
2:     if nonDeterministicFoo() then
3:         var input = i;
4:         if nonDeterministicBar(input) then
5:             o = f(input);
6:         end if
7:     end if
8: end while
```

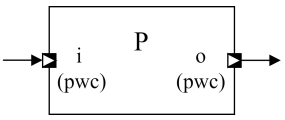


Figure 4.3: Example of a *piecewise-constant* generator.

For example, we take a simulation unit as represented in Figure 4.3 that implements the program implemented in Listing 3. The program reads the input i according to the condition represented by a non-deterministic function $nonDeterministicFoo$. Once it reads the value, it then computes the result only if the input i reaches an unknown threshold. If the threshold is reached, the program can then compute the result that depends on i and write on its output the value. Once computed, the result of the computation is written to its output variable o. The program then interacts with the other SU by reading or writing from/to the exposed variables i and o declared on the interface. The *piecewise-constant* data-nature is given by the internal and external assignment to these variables. In particular, at line 3, the program reads the input i and it assigns its value to the internal variable $input$. The knowledge of this particular behavior, related to the data-nature, exposes a possible coordination model that would not be possible without this information. In this case, the *piecewise-constant* data-nature gives the possibility to coordinate the SU by relying on the semantics of the assignment: the i value must be present at the time when it is evaluated for the assignment. In the complementary case for the output port, in the line 5, the result of the function f is assigned to the variable o, defined as an output port. The resulting data-nature will be then defined by the internal assignment: the

value is present at each instant but the variable o presents discontinuities caused by the assignment.

Transient A variable is defined as *transient* if its value is present only at a specific point in time and absent at other points in time. Transient data are usually associated with the notion of event or signal as, for instance, in synchronous languages. Furthermore, the data-nature of a port is defined as *transient* when its value exists only for an instant or it is associated with an internal SU event. In this case, the *interesting coordination points* in time are the instants when the data is present. For example, the Figure 4.4 shows a simple Timed Finite State Machine (TFSM) reading the variable x every second and it triggers the event e_1 after 5 seconds when it reads a value over a predefined threshold.

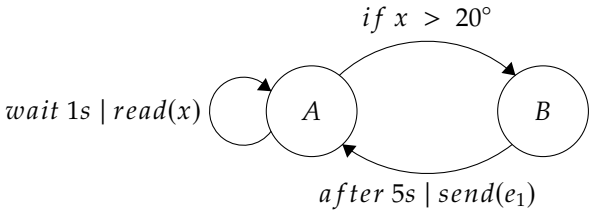


Figure 4.4: Simple Timed Finite State Machine (TFSM) representing a sensor that triggers an event when a predefined threshold is reached.

All in one, the data nature gives hints to the coordination of what is the event of interest on the data. Of course one can decide knowingly to sample *piecewise-constant* data, aware of potential problems as illustrated in subsection 3.1.3.

4.2.3 Temporal references

Due to the heterogeneity of formalism participating in the system, the time representation can be different across the simulation units. In a Cyber-Physical System, time may have different representations, depending on the semantics of the simulation unit and its formalism. Even if physical time (with or without relativity effects) is mainly used to model CPS but *time* can also be *logical* or *user-defined* (*e.g.* semantics driven by the distance or angle), discretized, superdense or even from a different magnitude like if we consider for instance the angle of a camshaft [172]. Allowing the coordination model to align different temporal reference is important to ensure the correctness of the simulation. For example, the Global Positioning System (GPS) uses to keep physical time synchronized between the satellite and the receiver device. Due to the relativity effects, the satellite embedded atomic clock experiences time slowdown. So, the clocks embedded by each satellite must tick about 38 microseconds faster than those on earth. If this effect is not taken into account, GPS computation would lose 10km of accuracy each day [173].

The time can also be *multiform*, where different times progress in a non-uniform way and independently from any reference to physical time. Inspired by the time model of MARTE [174], the model behavioral interface also defines the temporal reference of the model, *i.e.* the way time is encoded and the dimension it refers to (typically the physical time but possibly other dimensions like an angle or distance). This is particularly useful in case we are managing polychronous systems [175] (*i.e.* system whose temporal referential can come from different dimensions). For example, the angle is used to breathe out exhaust gases and take in the air for the next cycle. To perform these actions, it is necessary to open/close valves and actuate the fuel pumps thanks to an actuating mechanism that is triggered by the movement of the camshaft. The execution semantics for

this model is not based on time but on the angle. The actions depend on the movement of the camshaft *e.g.* the angle reached by it.

4.2.4 Simulation Unit Properties

The simulation unit may embed an executable model and its simulator. In the coordination context, it is useful to expose a set of properties and capabilities of the underlying simulator. A *capability* denotes a behavior of the simulation engine (or simulator) associated with the simulation unit. For instance, [87] identified a set of requirements to improve the co-simulation based on additional information on simulators, like for instance the input extrapolation method performed by the model [176] or the support of rollback mechanism. Such information could then be used during the generation of the co-simulation runtime.

We tried to identify which capabilities a simulator should support to enable a correct co-simulation of the model. For instance, the rollback capability allows to retrieve a previously saved state of the SU and restore it. In co-simulation, this technique requires the SU to keep a state of its states at different points in time and to restore it even if not explicitly requested by the external entity that is controlling the co-simulation (*i.e.* the coordinator or Master Algorithm).

For instance, the rollback capability can be used to compute scheduling that takes it into account. However, these properties are not currently exploited by the implementation of the framework but they represent important extensions for a future release.

4.2.5 Implementation

Any Model Coordination Interface has the same structure: providing the minimal set of elements required to co-simulate the simulation unit. For this reason, we developed a declarative DSL to formalize the specification of the Model Coordination Interface using the previously introduced concepts. This DSL is called Model Coordination Interface Language (MCILang).

In the following paragraphs, we illustrate the abstract syntax and the concrete syntax of the language, and its implementation as an Eclipse Plug-in to provide an Integrated Development Environment (IDE) to help its adoption and ease the definition of the specification thanks to some features such as an FMI model description importer.

Abstract syntax The abstract syntax of MCILang is defined by using an Ecore metamodel (see Figure 4.5). A more detailed figure is available in Appendix A.1. The root element is the *Interface*. An interface must refer to an executable Simulation Unit. At this moment, we support two different simulation units: FMI, exported as an FMU, and Gemoc Executable Unit, exported as a stand-alone JAR ‡ file. The *SUPath* supports both types of simulation units and it is used to define the absolute path of the actual simulation unit. Then, it exposes a partial view of the semantics and syntax of the SU defining three main parts:

‡ Java ARchive

- *Ports*: The set of ports contains some information used later to define the coordination specification. In particular, it exposes the type and the data-nature of the variable, its temporal reference, and its direction. If any temporal reference is defined, model temporal reference is used.;
- *Model Temporal Reference*: The Model Temporal Reference exposes the temporal reference of the SU, shared to all the variables, and used later to define the synchronization among the coordinated models. For the moment, it is a fixed enumeration encoded in the language of the interface but it would be interesting to allow the definition of different temporal references by the user at the model level. The temporal reference could for instance be expressed using MoCCML that allows the semantics specification of different temporal references [177];
- *Simulation Unit Properties*: This field is used to define the specificity of the model, in particular, the properties that can be exploited to define a dedicated coordination algorithm. It is possible to specify the supported properties specified in the subsection 4.2.4, such as rollback support.

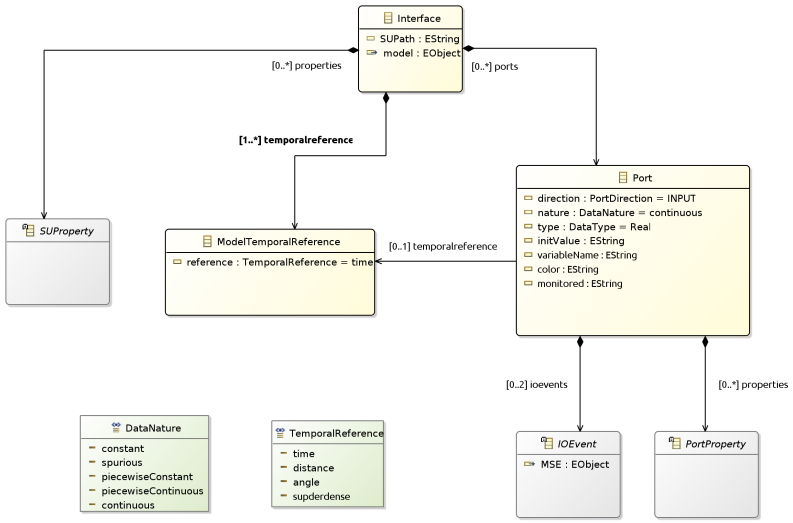


Figure 4.5: A partial overview of the class diagram for the MCILang language.

Following the classical definition of port in ADLs, a port is an abstraction of an exposed variable of the simulation unit. A set of shareable variables of the model are exposed through the interface enabling access from the external environment. To interact with the external environment or other models, the model should expose all the variables that can be modified by the external. Each port contains a set of attributes and properties. More precisely, a port is a variable that can be externally read or set, according to its direction.

We defined it as a *Port*, conforming with the definition given in Section 4.2.1. A *Port* is a structured variable or structure representing a variable and its properties. Properties express a particular behavior or additional information on the port. The port is composed by:

- *Value*: associated with the Runtime Data, it represents the current value of the variable;
- *Set of properties*: They define one or more properties on the port;
- *Direction*: It defines the direction of the port if it can be considered as an input or an output;
- *Type*: It defines the type of the value *e.g.* Boolean, Integer;
- *Data-nature*: It defines the nature of the value *e.g.* constant, continuous, or piecewise-constant.

The IOEvent represents the link between the state of the model (*e.g.* runtime data) and the elements of the interface (*e.g.* port) in order to exploit the internal semantics for coordination and communication purposes. The provided abstraction allows accessing the runtime data of the model only at specific points. The IOEvent can be defined according to the direction and data-nature specified for the Port. The direction defines which IOEvents can be specified: an IOEvent *Updated* if the port direction has the value *output*, IOEvent *ReadyToRead* if it has the value *input*. The IOEvent *Updated* means that, during the last step, the value of the port was updated. The fresh value is now available as the current value of the port. Therefore, the IOEvent *ReadyToRead* means that, during the last step, the internal computation requires reading the value from the Port to execute the next computational step. The current implementation supports the specification of IOEvents only in the case the port has a *piecewise-constant* data-nature.

As defined for the port, it is possible to define the properties of the simulation unit. A property denotes a specific behavior of the simulation unit and it is not associated with a specific port. A SU property can be defined as follows:

- *Rollback*: The simulation unit is able to restore a previously saved state at specific points in time and restore a previous state if requested;

Making explicit the specification of these properties will help to generate the dedicated coordinator. According to the specified properties, the coordinator executes the models in a specific order that permits the exploitation of the different specified properties. For example, if a model exposes the capability to perform rollbacks, it is used by the generator to generate a dedicated coordinator that executes this model before a model that cannot perform rollbacks. In this way, it is possible to avoid useless rollbacks and increasing accuracy without decreasing the performance.

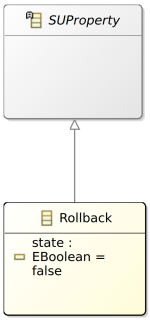


Figure 4.6: Class diagram of the model properties in the MCILang language.

Concrete Syntax The concrete syntax of MCILang is introduced by describing the MCIs of the running use-case simulation units along with their intended meaning. In particular, we introduce two MCIs that define the concepts defined in the abstract syntax. In the first example, we build the MCI for the model of the Timed Finite State Machine presented in Figure 4.4. The model presents two ports: an input port x used to retrieve the external temperature and the output port e used to send the event when the threshold is reached. The resulting abstracted model is then presented in Figure 4.7 and the corresponding MCI is described in Listing 4.1. In the first port (see line 4), given the continuous nature of the temperature value, we define it with the *continuous* data-nature. If any initial value is needed, it can be specified using the attributes *initValue* with the value encoded as a string. In the output port e (see line 11), we define the port as a *transient* data-nature due to the internal event. $SUPath$ attributes define the absolute path where the actual simulation unit is hosted.

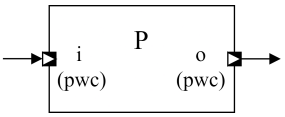


Figure 4.7: Example of a *piecewise-constant* generator.

```
1   Interface Controller {
2     SUPath "path/to/the/simulation/unit"
3     ports {
4       Port x {
5         direction INPUT
6         nature continuous
7         initValue "0.0"
8         type Real
9         monitored true
10      },
11      Port e {
12        direction OUTPUT
13        nature transient
```

```
14            type Real
15            ioevents {
16              Triggered
17            }
18            monitored true
19            plotterColor "Color.GREEN"
20          }
21        }
22        temporalreferences {
23          ModelTemporalReference t {
24            reference time
25          }
26        }
27      }
```

Listing 4.1: Excerpt of the *Controller* interface conforming to the model illustrated in Figure 4.19.

In addition to the previous MCILang elements, we introduced few elements more related to the analysis of the co-simulation results. These elements are taken into account for logging and displaying variables and their runtime values. In particular, we add:

- `monitored`: Boolean variable that enables the logging of the value of the variable. Enabling it will result in the generation of the associated code in order to log and display the obtained results;
- `plotterColor`: String variable that defines which color to display for the associated variable. It is possible to define colors present in the Java class *Color* [§] or define a new color by using the construct `new Color(float r, float g, float b)` provided by the Java class *Color*.

The resulting configuration will be used to generate the associated source code that handles the logging to a textual file, one per variable, and a logger that shows the values at runtime. In Listing 4.1, we use these two attributes to monitor both ports (producing the corresponding traces during the simulation) and, in the case of the port e, we define the color of the trace by specifying its color. The concrete syntax is supported by the IDE that helps to define the MCI providing some facilities such as completion and auto-filling in case the underlying simulation unit is FMI2.0 compatible.

Integrated Development Environment The project MCILang is part of the Gemoc Studio as a set of Eclipse plugins. The MCILang abstract syntax has been developed using Ecore and the textual concrete syntax has been developed using Xtext. It provides some facilities to create and edit the specification.

Some facilities exist to allow easier and faster development of a dedicated interface for some widely used standards such as FMI Standard [4]. The current implementation of MCIL allows to specify the path for an FMU (Functional Mockup Unit) as a property of MCI and retrieve the FMI interface and the executable model. The importer takes into account the list of exposed variables and their properties to generate the corresponding representation in MCILang.

The IDE supports the import of the Functional Mockup Unit by automatically creating the set of exposed variables defined as Input or Output. It creates each port from a single variable semi-automatically defining some properties: the unique name, type, direction, and data-nature. In this thesis, the data nature cannot be inferred automatically due to the black-box approach that prevents a formal approach to precisely retrieves the underlying semantics for the unit.

[§] https://docs.oracle.com/en/java/javase/11/docs/api/java.desktop/java/awt/Color.html

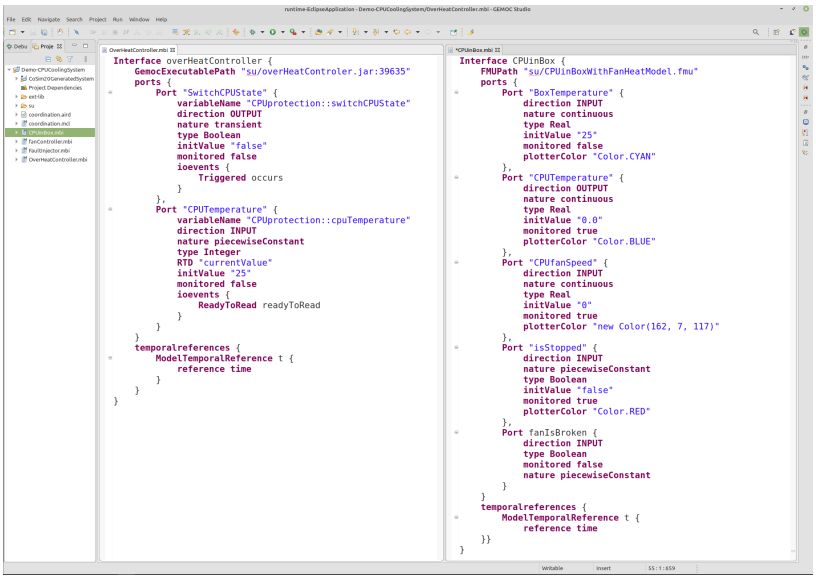


Figure 4.8: The MCILang textual editor.

Moreover, hierarchical FMU can contain different executable semantics that is hidden from an external point of view. Without any knowledge of the simulation units inside the FMU, the data nature cannot be automatically inferred.

The current implementation of the language and the associated IDE can be found on the public repository hosted by Inria [¶] at this link: `https://gitlab.inria.fr/glose/model-coordination-language`.

In this section, we have illustrated the main concepts of the Model Coordination Interface, by giving its elements and characteristics. We have then given implementation as a DSL called MCILang, with which it is possible to expose the elements required to correctly coordinate the simulation respecting its internal semantics. Using the data-nature property of the variables, the semantics can be exposed as a property of each variable without disclosing the Intellectual Property of the simulation unit.

Once defined the interface for each simulation unit, we need to define how the different simulation unit interacts: we will introduce a dedicated language capable to take advantage of the exposed properties and semantics of the simulation unit to define a coordination model tailored on its semantics.

[¶] `https://www.inria.fr`

4.3 Model Coordination Specification

A co-simulation execution requires defining a coordination model in order to exchange data among the simulations unit and to ensure time synchronization. To improve the coordination model, we propose to take advantage of the specific semantics of each simulation unit by relying on a Model Coordination Interface capable to expose a partial view of the syntax and semantics of the simulation unit. In particular, we introduce the concept of the data-nature to abstract the internal semantics without disclosing its internal mechanisms and preserving the IP. Furthermore, it is possible to exhibit synchronization points using Model Specific Events called *IOEvents* for certain data-nature such as the *piecewise-constant* data-nature.

To take advantage of the information defined on the Model Coordination Interface, we propose a specification based on MCI called *Model Coordination Specification (MCS)*. The goal of MCS is to enable a semantics-aware coordination model that we addressed in the research question 3:

> *What language constructions are required to define a correct coordination model between coupled simulation units, so that it can automatically result in a distributed co-simulation?*

Its role is to explicitly define which type of interactions must occur among the simulation units. The coordination model between two simulation units is then specified as a single element called *Connector*. All the inter-SU interactions are possible only by connectors. Consequently, a Model Coordination Specification contains a set of Connectors, each one defining a particular interaction between two or more models. In an interaction, at least two SUs must participate to be considered valid.

A connector is defined based on the semantics and syntactical elements defined in MCI. More precisely, a connector defines three main concepts: the actual *interaction*, defined as a directed-graph topology of the system, the *triggering condition*, defined as a boolean expression that defines when the interaction must be triggered, and the *synchronization constraint*, defined as a temporal constraint that must hold before to perform the interaction. The resulting connector is a tuple defined as follows:

$$C_i =< I^i, TC, Sync >$$

with $i \in \mathbb{N}$ and where C_i is the i-th connector of the Model Coordination Specification, the I^i is the set of interactions of the connector C_i that are realized when the triggering condition TC occurs ensuring the synchronization constraint $Sync$. In the next subsections, we discuss these three main concepts that composed a *connector*.

4.3.1 Interaction

A connector defines the interactions between two or more MCIs and their semantics. It embeds one or more interactions that occur between MCIs. An **Interaction** represents the assignment of the value from a source port o to a destination port i. The source port o must be an output port while the destination port i must be an input port. As represented in the Figure 4.9, each simulation

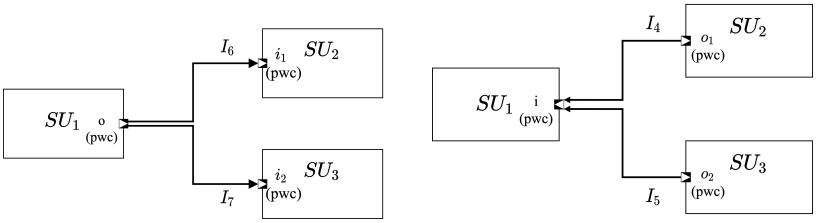


Figure 4.10: Two possible topological maps: 4.10a shows an allowed topological map; 4.10b shows a topology where multiple ports assign their value to a single port. The latter topological map is not allowed because a destination port must have only a source port.

exposes a port, SU_1 exposes an output port o while the SU_2 exposes an input port i. The assignment of the value from o to i is represented as follows

$$I_1 : o \rightarrow i$$

where I_1 is the assignment and $\rightarrow$ defines the assignment direction. The graphical representation for interaction is given by a directed edge that connects two ports according to their direction.

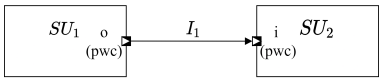


Figure 4.9: Simple system with two SUs and a connector.

Defining the interaction using an assignment can represent a limitation for certain scenarios where a System Engineer may require more expressive language. In the interaction statement is possible to perform deterministic logical and arithmetical operations such as unary operation such as negate a value (it does not affect the current value but only its assignment),

$$I_2 : \neg o \rightarrow i$$

, or using binary operators, such as addition or subtraction of two elements,

$$I_3 : o + d \rightarrow i$$

where d can be another port or a constant.

At the interaction level, some topological maps are not allowed due to the possible concurrency of assignments that will not be taken into account by the semantics of the connector. For instance, we consider the following set of interactions that textually defines the interactions in Figure 4.10b:

$$\begin{aligned} I_4 &: o_1 \rightarrow i \\ I_5 &: o_2 \rightarrow i \end{aligned} \tag{4.1}$$

Due to the possible concurrency between the two interactions, our approach does not take into account this topology. However, a single output port is allowed to assign its value to multiple input ports. For instance, as shown in Figure 4.10a, it is possible to define a system with three SUs where SU_1 can act as a sensor, SU_2 can act as a controller monitoring the values of the sensor, and SU_3 plots the current value of the sensor for debugging purpose.

$$\begin{aligned} I_6 &: o \rightarrow i_1 \\ I_7 &: o \rightarrow i_2 \end{aligned} \tag{4.2}$$

The expressiveness in the interaction statement is reduced at basic operations to ensure the determinism of the interaction. In fact, interactions are meant to exchange data without adding any additional semantics to the data exchange. The semantics will then be added using the other two elements of the connector: *triggering condition* and *synchronization constraint*.

4.3.2 Triggering Condition

Once defined the interactions statement of the connector, we must define its semantics. We divide it by proposing two elements: a *triggering condition* and the *synchronization constraint*. A **Triggering Condition** defines a boolean expression that represents the condition at which the interactions must occur. It is based on the data-nature of the ports participating in the set of interactions I^i defined on the connector

$$C_i =< I^i, TC, Sync >$$

The triggering condition exploits the semantics elements exposed on the MCI. For instance, if the data involved are all of *continuous* nature, the triggering condition should specify a sample rate at which the data exchange occurs. As another example, if an interaction involves a piecewise-constant output port, the triggering condition can refer to each update of this port (instead of using a time-triggered approach). The allowed triggering conditions are aligned with the expressiveness of predicates defined in [178] and aligned with the data nature defined in subsection 4.2.2.

In particular, we propose four different triggering conditions: *Sample rate*, *Event*, *Updated*, and *ReadyToRead*. We illustrate them in the following paragraphs.

Sample Rate We introduce first a triggering condition inspired by the co-simulation techniques for Continuous-Time co-simulation based on the notion of the communication step-size. In particular, we define the triggering condition using the following statement:

```
every <sample rate> <temporal reference>
```

where `sample rate` defines the size of the communication step and `temporal reference` defines which temporal reference the triggering condition is based on. For instance, in Figure 4.11 we represent two SUs that exchange data using

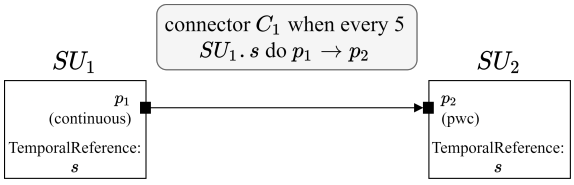


Figure 4.11: Co-simulation specification of two SUs based on a connector that defines a sample rate triggering condition.

a connector C_1: an output port p_1, that exposes a value with a *continuous* data-nature, is connected with an input port p_2, exposing a *piecewise-constant* data-nature. On the connector, we have defined as triggering condition a sample rate with a communication step size of *1 second* (`1 s`). The temporal reference s refers to the exposed temporal reference on the MCI of SU_1. It allows defining a sample rate based on the internal temporal reference of the SU, enabling the use of different temporal references exposed by the SUs of the system.

It is possible to exploit this triggering condition in all those scenarios composed of continuous-time SUs that requires a time-trigger approach to exchange data. If the scenario requires a coordination model based on the notion of *event* then it is possible to define it by using an *event* triggering condition.

Event In case the SU is DE-based and it exposes its IOEvents on its port (thorough its MCI), it is possible to exploit its *event* data-nature. It allows defining a discrete-event connector based on that notion. In this case, the triggering condition statement has the following structure:

```
event on <port> occurs
```

Let us consider as an example, as shown in Figure 4.12, a simulation unit SU_1 that encapsulates a Timed Finite State Machine: it reads a random-generated variable x and triggers the event e_1 is the threshold is reached. Then, the simulation unit SU_2 starts its execution when an event is received on its port p_2. The connector C_2 specifies that when the event e_1 occurs during the simulation of SU_1, the associated value is propagated. The interactions rules (not $p_2 \rightarrow p_2$) specify that when the event e_1 is triggered than the value on the port p_2 is negated and assigned to the port itself. The resulting data-nature of the port will be piecewise-constant due to the change driven by the event.

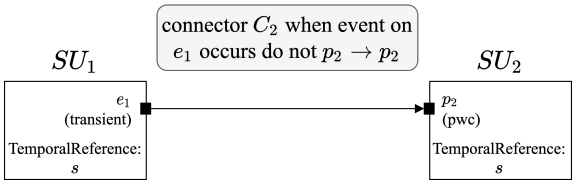


Figure 4.12: Simple Timed Finite State Machine (TFSM) representing a sensor that triggers an event when a predefined threshold is reached.

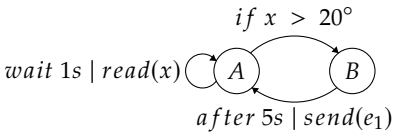


Figure 4.13: Timed Finite State Machine (TFSM) encapsulated in SU_1 in Figure 4.12.

Updated The *updated* triggering condition relies on the exposed *IOEvent* provided by the MCI. The event allows catching the change of the value for the port on which it is defined on. The specification of the triggering condition is then defined as following:

```
value on <port> has been Updated
```

where *port* is the port that exposes a *piecewise-constant* data-nature along with the *Updated* IOEvent (as specified in subsection 4.2.5).

Let us consider a system (Figure 4.15) composed of a cyber model of a wheel encoder driver (SU_1), producing a digital output p_1. A wheel encoder is a sensor that allows tracking the number of wheel rotations (see Figure 4.14). More precisely, in our example, the output of the driver switches from 0 to 1 or conversely each time a 1/36 of revolution of a wheel is done. Signal p_1 is consumed by another simulation unit (SU_2), which computes the actual speed of the wheel according to the time spent between two successive switches of p_1. Consequently, $v1$ is assigned only at specific points in time, depending on the speed of the wheel. This assignment creates a discontinuity, which is neither a rare event nor symptomatic of a specific phenomenon like in models of physical systems.

ReadyToRead Another *IOEvent* that is possible to defined on a port of the MCI is the `ReadyToRead` event: it allows to *pause* the current execution of the simulation unit *before* the reading of the current value of the port and *wait* for the value at the current time of the simulation unit. In the model coordination specification, it is possible to exploit by defining a triggering condition as following:

```
value on <port> is ReadyToRead
```

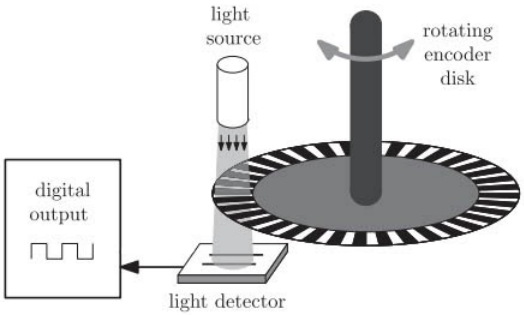


Figure 4.14: Wheel encoder example. The rotating encoder disk has 36 slots. Therefore, the wheel has traveled one revolution every 36 pulses. Taken from [179]

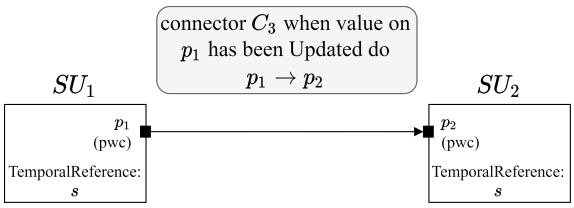


Figure 4.15: Overview of the wheel encoder system defined in Figure 4.14 with the coordination model specification as the connector C_3.

As an example, consider a cyber simulation unit SU_1 that senses the environment (*e.g.* a room, a CPU) and computes the temperature. Usually, in the actual implementation of such a system, the environment is sensed periodically. In Figure 4.16, the cyber simulation unit is represented by the Arduino controller with its connected sensor. We consider here that the environment that provides the temperature evolution is modeled by a physical simulation unit SU_2 whose output is read periodically by SU_1. In Figure 4.16, the physical simulation unit is represented by the surrounding environment. In usual time-triggered co-simulation, the co-simulation period is chosen so that the data obtained by the physical model is *fresh* enough when propagated to the cyber model. There are three drawbacks here. First, the cyber model is called several times to update its input even if this input is not required to be read internally thus wasting simulation time. Second, the physical model is called several times to compute fresh values that are actually not used by the cyber model. Third, there is no synchronization between the actual reading of the input by the cyber model and its update by the coordinator. This can lead to a temporal inaccuracy since the actual reading can occur at the end of a simulation step, *i.e.* without a *fresh* input. In this case, either the designer considers that the freshness of the data is not important (but that can lead to wrong simulation results!) or the designer decreases the co-simulation period and consequently decreases the performance. As shown in the section 3.1.3, increasing the number of communication points between models and coordinators for better accuracy decreases the overall performance and therefore we aim at reducing the number of communication points without reducing accuracy. In the model coordination specification, we can

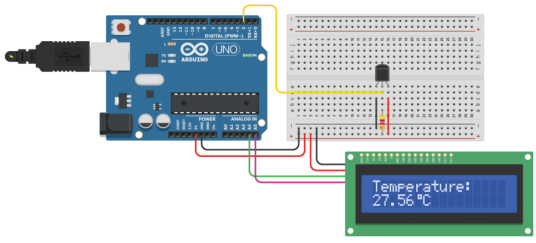


Figure 4.16: Temperature sensor example. The temperature is retrieved from the sensor only at specific point during the execution of the controller code. The system is then composed by two logical components: the environment with a sensor and the controller.

take advantage of the `ReadyToRead` event defining a triggering condition based on it. Figure 4.17 shows the connector defined on the previous system on which SU_1 exposes a piecewise-constant data-nature along with a `ReadyToRead` event. Based on that, the connector specifies that when the simulation unit SU_1 requires to read the value on p_1, the value is retrieved from p_2 at the time required by SU_1: a temporal synchronization will happen if the two simulation units are not at the same time.

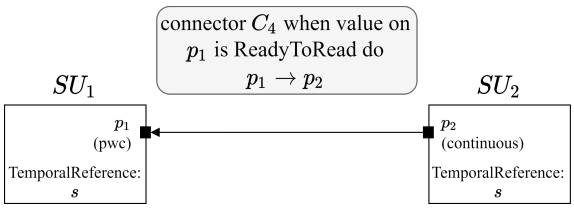


Figure 4.17: Overview of the temperature sensor system defined in Figure 4.16 with the coordination model specification as the connector C_4. The environment model is encapsulated into SU_2 while the Arduino controller model is encapsulated into SU_1.

The exact synchronization between two simulation units can be specified by using the *synchronization constraint* statement.

4.3.3 Synchronization Constraints

Given the previous definition of a connector

$$C_i =< I^i, TC, Sync >$$

we focus on the $Sync$ element of the connector. The **Synchronization Constraint** defines the relation between the temporal references in different MCIs. This relation defines when the actual interaction can be done. Typically, such constraint specifies that the time must be equal when two models exchange some data. However, on one hand, we want to be more expressive, for instance, to encode some time relativity effects between different timebases (for instance between the time on earth and in a satellite) or to support polychronous systems [175], *i.e.* system whose temporal referential can come from different dimensions (*e.g.* a distance or an angle). For example, the camshaft is used to breathe out exhaust gases and take in the air for the next cycle. To perform these actions, it is necessary to open/close valves and actuate the fuel pumps thanks to an actuating mechanism that is triggered by the movement of the crankshaft. The execution semantics for this model is not based on time but angle. The actions depend on the movement of the camshaft and not on the physical time. It is also possible to express quantities, like interval or duration, which is consistent for the models to exchange data, without relying on a perfect synchronization.

On the other hand, it is also important to be able to define the exact condition under which time is considered to be the same in two different models, for instance, if time is encoded as a Float in one model and as an Integer in the other. Such constraints, when respected, ensure a consistent temporal state of the two models, allowing the interaction to occur.

4.3.4 Implementation

Based on the previous definition of the Model Coordination Specification, we developed a language called *Model Coordination Language* (MCL). It is a DSL that allows the definition of a Model Coordination Specification between one or more models exposing the Model Coordination Interface (written in MCILang). It

allows specifying all the concepts described for coordination according to the definition given in Section 4.3.

It enables the specifications of conditions and constraints under which data exchanges have to occur. Inspired by coordination languages like REO [136] and Wright [135], it defines the coordination model as a set of rich connectors between ports from the MCI. The main element of MCL is a *MCLSpecification* that imports Model Coordination Interfaces and contains a set of *Connectors*. At least two Model Coordination Interfaces must be imported in order to define a connector. This section describes the basic syntax and semantics of these rich connectors, showing how they are combined to specify the coordination between simulation units in an unambiguous way.

The current implementation of MCL is an Eclipse plugin in Gemoc Studio, which integrates several facilities in the context of the Eclipse Modeling Framework (EMF). Textual and graphical editors are then provided to ease the specification of the coordination model. Moreover, an integrated automatic process generation makes ease the development and configuration of the entire system using the framework Cosim20. The MCL syntax has been developed using the EMF metamodel facility Ecore, the textual concrete syntax has been developed using Xtext ‖, and the visual concrete editor has been developed using Sirius **. The textual and visual editor are synchronized to ease the creation, modification, and validation of the coordination model.

The resulting implementation is then integrated into the Cosim20 framework. It allows handling both the MCI specifications written using MCILang and the MCL specification developed by using MCL. Furthermore, the automatic runtime generation process is embedded in the MCL implementation and it allows the creation of a runtime co-simulation producing the corresponding source code in Java. Then, it can be compiled and executed to run the actual co-simulation. The runtime co-simulation coordination algorithm and its automatic generation will be presented later in this chapter.

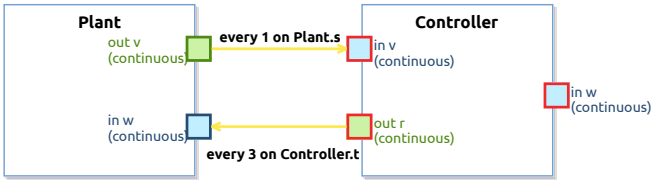


Figure 4.18: Overview of a classic example with a controller and a plant.

A complex system can contain hundreds of simulation units and each one can expose a huge amount of port. Without advanced features to help and ease the definition of coordination mode, this task can be difficult and error-prone. Xtext provides some advanced editing facilities such as context assist auto-completion but is based only on the syntax. We augmented the standard module presented in Xtext with a semantics context assistant that takes into account the information provided in the Model Coordination Interface of each Simulation Unit. We can exploit the properties of the set of ports expressed in the MCI to match elements that fit the current context in which the writer is on. For instance, in Figure 4.18, we have an example composed by two Simulation Units, `Plant` and `Controller`, connected by two connectors. The first connector, named *Feedback*, links the output port `v` of the `Plant` with an input port of the `Controller`. During the specification, the keyword **from** triggers the completion process to retrieve every input port available on the SU `Plant`. Then, when the system designer has chosen

‖ https://www.eclipse.org/Xtext/
** https://www.eclipse.org/sirius/

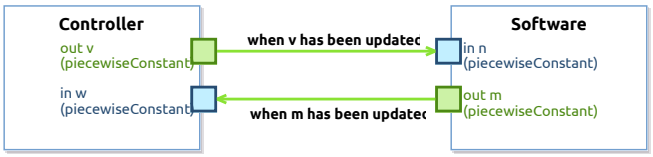


Figure 4.19: Overview of a rejected MCL specification. A topological loop of *initiator* predicate cannot be automatically transformed into a deadlock-free coordination algorithm.

the port, the **to** keyword triggers the completion process that retrieves every compatible port with the one chosen before. The facility eases the connection between ports, avoiding incompatible connections by type, data nature, and direction. It permits the definition of a coordination model that does not present any mismatches. The resulting textual *MCSpecification* is listed in Listing 4.2.

```
1  load "Plant.mci"
2  load "Controller.mci"
3
4  Connector Feedback ( from Plant.v to Controller.v)
5  when every 1 Plant.s
6  do
7  Plant.v -> Controller.v
8
9  Connector Command ( from Controller.r to Plant.w)
10 when every 3 Controller.t
11 do
12 Controller.r -> Plant.w
```

Listing 4.2: Textual specification of the connector.

Further checks are also implemented to reject non-schedulable coordination algorithms based on the current static topology. For instance, Figure 4.19 shows a simple system composed of two simulation units, a *Controller* and a *Software* module, that exchange data using two connectors, as defined in Listing 4.3, where their composition creates a loop where all the triggering conditions are of types *Initiator*.

```
1  load "Software.mci"
2  load "Controller.mci"
3
4  Connector Ctrl (from Software.m to Controller.w)
5  when value on Software.m has been Updated
6  do
7  Software.m -> Controller.w
8
9  Connector Software (from Controller.v to Software.n)
10 when value on Controller.v has been Updated
11 do
12 Controller.v -> Software.n
```

Listing 4.3: Excerpt of the MCL specification with two connectors. Each connector defines an *Initiator* condition on it, closing a loop.

As we introduce later in this Chapter, we use the initiator definition to give an execution priority in the system and configure it accordingly. Moreover, this configuration can produce a graph where:

1. *Node* represents a single simulation unit;
2. *Edge* represents a triggering condition that is specified between two ports.

In this case, we create an edge only if an initiator condition of type *updated*. The resulting graph is then analyzed using Depth First Search (DFS) to detect the presence of a cycle. The DFS for a connected graph produces a tree. There is a cycle in a graph only if there is a back edge present in the graph. A back edge is an edge that is from a node to itself (self-loop) or one of its ancestors in the tree produced by DFS. If a cycle is detected then the system cannot be coordinated automatically due to the lack of information on the priority among the simulation units. For

instance, in Figure 4.19, if *Controller* executes first and goes ahead of *Software* when it receives an updated value from *Sofware*, it must execute a rollback to synchronize its execution (the implicit *synchronization constraint* defined that the internal time of the two SUs must be equal). The lack of any rollback capability for the SU *Controller*, as illustrated in Listing 4.4, does not allows this specification. In the case the SU *Controller* supports a rollback capability, then it is possible to compute a schedule based on this information. *Controller* could execute and when it receives an updated value from *Software*, it rollbacks and takes into account the newly received value.

```
Interface Controller {
  SUPath "/path/to/the/controller"
  ports {
    Port v {
      direction OUTPUT
      nature piecewise-constant
      type Real
      monitored true
    },
    Port w {
      direction INPUT
      nature piecewise-constant
      type Real
      initValue "0.0"
      monitored true
      plotterColor "Color.BLUE"
    }
  }
  temporalreferences {
    ModelTemporalReference t {
      reference time
    }
  }
}
```

Listing 4.4: Excerpt of the *Controller* interface conforming to the model illustrated in Figure 4.19.

In this section, we have illustrated the Model Coordination Specification by defining its elements and by showing the corresponding modeling environment. The MCS is composed of connectors that define the coordination models between two or more simulation units. By relying on the MCI of each SU, the coordination model can be expressed using semantics-aware elements exposed as ports' data-nature. In particular, the connector is based on three main parts: an interaction, a triggering condition, and a synchronization constraint. The interaction defines the connection between two ports by specifying their causality and their dataflow. The triggering condition, defined as a boolean expression, is used to express the set of conditions on which the set of interactions must be performed (*i.e.* the current values must be propagated). To ensure the temporal synchronization among the simulation units, the synchronization constraint defines explicitly the constraint that must hold in order to exchange data between two simulation units. To ease the definition of the MCS, we provide an integrated environment with completion and error check features, preventing the specification of coordination models that brings the co-simulation of the system to deadlock. In the next section, we illustrate the Distributed Coordination Algorithm generated from the previously defined MCS and MCIs.

4.4 Distributed Coordination Algorithm

The Model Coordination Specification and the Model Coordination Interface give coordination at the model level: it describes and connects concepts and elements of models but it does not enable the actual co-simulation of the simulation unit. To execute the co-simulation, it is required to realize the code that executes each model in the system. We called it *Coordination Algorithm* or *Master Algorithm* (as in FMI Standard [4]). In particular, we focus on the third research question:

> *What coordination algorithm can be used to study emerging behaviors of a heterogeneous CPS without introducing timing bias?*

In this section, we show a distributed algorithm built using *static* information *i.e.* topology, causal, timing, and event relationships, and *runtime* information *i.e.* synchronization events, timing, data exchange. Then, the source code implementing the algorithm is generated using coordination information contained in the MCI and MCL specifications. The Algorithm 4 shows the pseudo-code of the Runtime Coordination Interface (RCI) (see Figure 4.1). The RCI contains the distributed coordination algorithm that relies on (1) the native runtime interface of the simulation unit to perform the computational concern and (2) on a distributed communication technology to exchange actions to be performed across the system by other simulation units.

4.4.1 Semantics-aware API

We propose to consider the FMI time-triggered interface *doStep(Δt)* as a specific case where we ask a simulation unit to simulate until a specific predicate characterized by an amount of time spent in the simulation unit[††]. Following the same rationale, the proposed semantics-aware API can ask a simulation unit to execute until a specific coordination predicate holds. The predicate must be expressed according to the information from the simulation unit behavioral interface (typically containing input/output nature, time representation, and solver capability [8, 11, 19, 180]). Consequently, the general form of the proposed doStep API is:

`StopCondition` doStep(`CoordinationPredicate` p)

where p expresses a condition under which the execution should pause, *i.e.*, the condition under which the doStep function returns. For instance, considering the input and output nature as defined in [19] (i.e., continuous, piece-wise constant, piece-wise continuous or transient), the concept of predicate for correct coordination can be defined as shown in the class diagram Figure 4.20.

[††] Note that, in reference to study on Model of Computations [16] that this may be done only for timed simulation units.

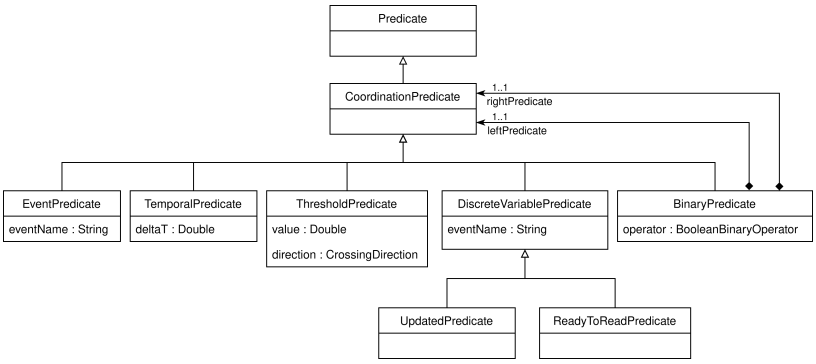


Figure 4.20: Minimal but extendable set of predicates.

If the simulation unit supports only temporal predicates, then it corresponds to the FMI API. However, other coordination predicates have been defined. Here is a brief description of their meaning and their typical use case.

1. `TemporalPredicate` is a predicate that becomes true when the internal time of the simulation unit reaches the value of the predicate. This is the classical FMI predicate.
2. `UpdatedPredicate` is a predicate that becomes true as soon as the referenced variable, which must be a piece-wise constant output, has been assigned. It typically corresponds to the example from Figure 3.27, which can then be managed without data loss, delays, or very small communication step size; *i.e.* in a correct way.
3. `ReadyToReadPredicate` is a predicate that becomes true just before the simulation units actually read the referenced variable. It is typically used if there is a need to provide an input to a simulation unit that actually reads (non necessarily in a deterministic way) this input at specific points in time. Instead of periodically providing the input data (consequently with unavoidable delays), the data is provided only when needed by the simulation unit.
4. `ThresholdPredicate` is a predicate that becomes true when the referenced variable crosses the defined threshold (according to the crossing direction[‡‡]). It is typically used when a simulation unit is waiting for a specific threshold on a value from another simulation unit. Instead of periodically providing the input data (consequently with unavoidable delays) to be tested and possibly using rollback for more precision, the data is provided only when the condition is reached.
5. `EventPredicate` is a predicate that becomes true when the referenced event occurs. While this is in our implementation only used for cyber events, it may also be extended to encompass discontinuities or other kinds of events on (piecewise) continuous signals.
6. `BinaryPredicate` defines the disjunction of other predicates.

Finally, the proposed API also provides the classical function like for instance `loadModel`, `get/set Variable`, `get/set State` and `terminate`.

What is important is the (preliminary) definition of the coordination predicate, which is, according to our experiments, the minimal set of predicates to have an accurate coordination *i.e.* without losing any data, events, or signals. Note that for now, we are only using the disjunction of predicates since it is not clear about the meaning of their conjunction. For instance, existing works about Event constraints suggest using Union or Inf/Sup constraints instead of AND since

[‡‡] It can be either from above to below, from below to above or both.

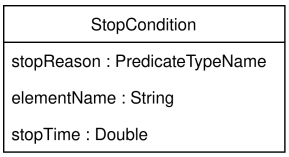


Figure 4.21: Simple `StopCondition`, returned by the `doStep` function.

they intrinsically embed a notion of order which is not existing in the classical Boolean operators [181, 182].

To these predicates, many others could be added like for instance a discontinuity predicate that stops when a discontinuity is detected on a piece-wise continuous variable (see description of the Event predicate). Another more complex predicate could be a Büchi predicate, which is verified when a specific state-based observation occurs. There is no real reason to limit the kind of predicate that can be defined, as long as it makes sense according to the simulation unit execution semantics.

In other words, based on the simulation unit behavioral interface, one can speak about the simulation in terms of predicates that are relevant in the particular simulation units used in the co-simulation. For instance, considering a simulation unit interface of an untimed simulation unit, no temporal predicate can be used. In the same idea, if the simulation unit exposes only (piece-wise) continuous variables, then it should not be possible to refer to these variable updates (since it creates an undesired connection with the internal simulation unit discretization step). In short, the acceptable predicates for a specific simulation unit can be inferred from the simulation unit behavioral interface of such simulation unit. However, it is also important that each tool specifies the predicates it supports.

The value returned by the `doStep` function must allow the coordinator to understand why the simulation was actually paused so that it can do the appropriate action. For instance, if the simulation unit was paused due to an `UpdatedPredicate`, then the variable that has been updated should be communicated to the appropriate simulation unit input (after being sure that the receiving simulation unit is at the same time as the emitting simulation unit, aligning the time if needed). For now, we used a simple form a `StopCondition` but it might be aligned with the `Predicate` class diagram. The Figure 4.21 shows a minimal proposition for a simple `StopCondition`. The *StopReason* is a predicate type defining why the simulation was paused; the *elementName* defines the referenced element link with the stop reason and the *stopTime* stores the internal time of the simulation unit when paused.

Observation 1: This is not clear yet how the link should be made between the name of an exposed variable in the simulation unit behavioral interface and the actual variable inside the model under simulation. For now, we are using qualified names instead of simple names like in the simulation unit behavioral interface. Similarly, for experimental facilities, we are using a Double to encode time in the co-simulation. It does not mean that the time is internally a double (since it may be encoded by super-dense time for instance) but it provides a helpful homogenization of the time from the coordination point of view.

Observation 2: According to our definition, FMI is a specific mold of our interface since it defines only (piece-wise) continuous variables and it does not allow for Threshold predicate injection. Consequently, the only acceptable predicate is a Temporal predicate.

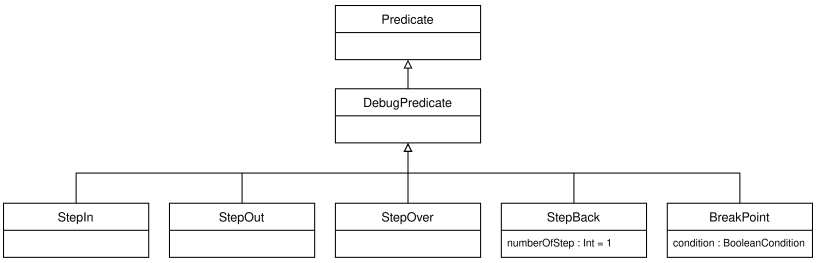


Figure 4.22: Set of *DebugPredicates*.

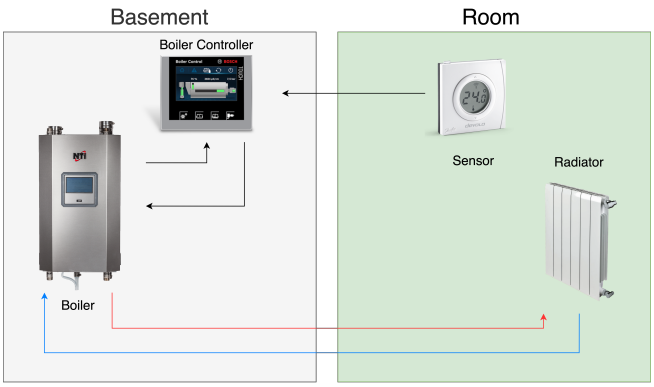


Figure 4.23: Overview of a Home Heating System.

We show in the validation Chapter 5 how this API, implemented for language developed in the GEMOC studio [26], provides a simple way to gain in term of accuracy and performance during the coordination of multiple simulation units. However, in the next subsection, we overview how it can be used for other usages, typically debugging.

Extension of the API for Debugging In this subsection, we show an implementation experimented in the GEMOC studio to use the very same API for debugging. Our goal was to implement the functionality of an API as defined in the usual debugger. We consider this useful for the developer of one simulation unit when she/he wants to debug the simulation unit in the context of the other simulation units. For this reason, we considered that breakpoints are defined with another interface and considered only the way to execute the simulation unit. To define the new use of the interface, we simply defined the necessary Predicate for debugging (see Figure 4.22) and implemented the corresponding management of the Predicate in a wrapper. Furthermore, it is interesting to realize that debugging equational simulation units could use a totally different notion of breakpoint. For instance, one could want to pause the simulation when the derivative of a specific output reaches a symptomatic threshold, in order to check different values in the system and try to understand what actually happens. In this case, Predicates should be defined accordingly.

Once again, we tried to provide an extendable simulation API, focused on co-simulation but suitable for different activities.

4.4.2 Coordination Algorithm

In this section, we present a distributed algorithm generated from the information of the modeling environment introduced above. To introduce using an explicative example, we introduce a simple example of a Heating Home System.

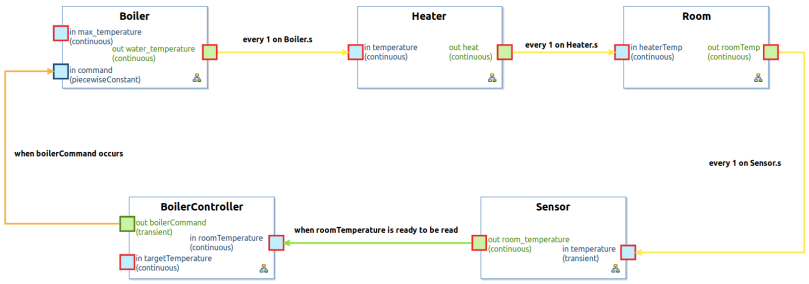


Figure 4.24: MCL Visual representation of the Home Heating System. Each connector specifies under which condition and constraint the interaction must take place.

The main idea is that a connector creates an *Initiator/Follower* relationship on the ports it connects, depending on its triggering conditions. An initiator port initiates the data exchange and chooses the time at which it occurs. On the opposite, a follower port waits for the initiator and does not know about the time at which the data exchange must occur. For instance, the triggering condition of the connector in Figure 4.24 is "**when value on** `roomTemperature` **is ready to be read**". This means that each time the *BoilerController* simulation unit internally reads the *Sensor* variable `roomTemperature`, it pauses just before the reading so that a fresh value can be provided to it (see [178] for details). In this case, the target of the connector initiates the data exchange by querying for a value at a specific point in time (to the related follower ports, *i.e.* the source of the connector). For this mechanism to work correctly, it requires that the simulation unit which exposes the follower ports did not simulate beyond the point in time at which the value is queried. More generally it means that each simulation unit that exposes a follower port should not simulate beyond the minimum point in time of the simulation units exposing the initiator ports. We name this point in time the *Temporal Horizon* of the simulation unit. Rephrased, it means that a simulation unit that has at least a follower port can never simulate beyond its temporal horizon to be able to correctly handle queries from its connected initiator ports. Sometimes, it is possible to take advantage of *deterministic triggering conditions*. For instance, the triggering condition of the *Boiler2Heater* connector on Figure 4.24 is "**every** 1 **on** `Boiler.s`". In this case, statically, the follower ports know when the data exchange will happen (every 1 second of time units of the Boiler simulation units). We refer to such ports as *DeterministicFollowerPort*.

4.4.2.1 Description

The proposed algorithm runs in parallel for each simulation unit (in the runtime coordination interface of Figure 4.1). The list of initiator ports, follower ports, and deterministic follower ports are parameters of this algorithm, generated from the modeling environment (see line 1 in Listing 4). The algorithm also relies on (1) the native runtime interface of the simulation unit to perform the computational concern and (2) on a distributed communication technology to exchange actions to be performed across the system by other simulation units. It means that an initiator port will send actions to be done by the simulation unit of the associated follower port. These actions are stored in the simulation *todo* list, which is a list of actions to be done, sorted according to the time at which action must be done. The next action in the *todo* list is the action with the smallest time. An action is a request to the simulation unit. There are three kinds of action: *publish(data, time)* to ask a simulation unit to publish data on a port at a specific time, *set(data, value, time)* to ask a simulation unit to internally assign a data to `value` at a specific point in time and *updateTH(time)* to update the temporal horizon of the simulation unit.

Before running the co-simulation, it computes, according to the list of its initiator port triggering conditions, the predicate at which it is mandatory to pause the simulation. For instance, if we consider the triggering condition of connector *cpuTemp2* already explained, the predicate requires that the simulation must pause when the cpuTemperature variable is (internally) ready to be read. The conjunction of such predicates required by the initiator ports is then constructed (line 2).

After the initialization and while the simulation is running, the next action in the *todo* list is taken (line 4). Note that, a simulation unit can assign itself actions to do, typically if it possesses initiator ports for which actions to be done are a priori known (*e.g.* to publish a data on a port at every 5 time units).

Each action must be done at a specific point in time, this is the current action time attribute. If the time of the current action is the current simulation time (*i.e.* the time at which the simulation unit is actually paused) then it performs the action (see lines 5 and 7 of Listing 4).

If the current action is in the future, the simulation unit has to check if it can actually simulate or not (lines 8 and 9). For that, we first check, according to the current action, initiator predicates, and deterministic follower ports, what we have to do in the future and how much we can simulate. If we cannot advance in time (the max step size is 0) it means that we have to wait for a new temporal horizon from another simulation unit (*i.e.* we wait for the temporal horizon from a simulation unit connected to the actual simulation unit through an initiator/follower relation) (line 11). Also, before waiting, we have to reschedule the current action by putting it in the *todo* list (line 10).

If the max step size is greater then 0, then it is add to the initiator predicates (line 13) and the simulation is restarted ($doStep$ line 14). When the simulation pauses, the now variable is updated according to the stop condition of the doStep call. This stop condition makes explicit the reason why the simulation stopped. Pragmatically it refers to the part of the predicate that became true and the algorithm can consequently determine the actions to be submitted to other simulation units. For instance, if the stop condition tells that the simulation units are ready to read the $cpuTemperature$ (see connector $cpuTemp2$) then the $publish(cpuTemperature, now)$ action is sent to the simulation unit that contains the CPU temperature variable. Thanks to the temporal horizon mechanism (lines 9 to 11), the time in the simulation unit that will receive the action will be lower or equals to the local now sent in the $publish$ action. Eventually, the emitter of the action will receive a set action with the requested value of the variable at the correct time. Finally, depending on the stop condition reason, the current action may be accomplished or not. If not, it is rescheduled (lines 17 and 18).

If the coordination defined by the rich connector does not violate constraints (see the previous section) then this algorithm ensures that no action in the past will be present in the *todo* list of a simulation unit. In other words, there is no need to rollback. Additionally, as shown in the next section, such an algorithm drastically reduces the number of required communication between simulation units and avoids timing bias like the one in Figure 3.10.

Algorithm 4 Pseudo-Code for the Wrapper Coordination Algorithm.

```
1:  function COSIMULATE(Set<Port>initiatorPort, Set<Port>followerPorts,
    Set<Port>deterministicFollowerPorts)
2:      initiatorPred ← setOwnedInitiatorPredicate()
3:      while simulationIsRunning do
4:          currentAction ← todo.getNextAction()
5:          if now = currentAction.TH then
6:              realize(action)
7:          end if
8:          maxStepSize ← getNextStepSize(action)
9:          if maxStepSize = 0 then
10:             todo.add(action)
11:             waitTH()
12:         else
13:             predicate ← maxStepSize ∪ initiatorPred
14:             stopCondition ← su.doStep(predicate)
15:             now ← stopCondition.time
16:             submitActionsToOtherSU(StopCondition)
17:             if currentAction! = done then
18:                 todo.add(action)
19:             end if
20:         end if
21:     end while
22: end function
```

4.4.3 Implementation

We then discuss the actual implementation of the distributed coordination algorithm. We developed a standalone version using Java 1.8 that can be imported as a library for future reuse.

4.4.3.1 Message-oriented Distributed Architecture

In a distributed architecture, different entities and components (*i.e.* nodes) are hosted on different devices and platforms cooperating to achieve a given objective over a communication medium (*e.g.* pipe, networks, shared memory).

There are three main benefits to implement a distributed system:

- **Scalability**: Each computation happens independently on each node, it is generally easy to add additional nodes and functionality;
- **Performance**: A node can be customized on the needs for specific computational requirements, improving performance for that kind of computation;
- **Geographical Distribution**: Intellectual Property software can be protected by distribution allowing external access to its use without distributes it.

In order to communicate among nodes, two main alternatives exist: Remote Procedure Call (RPC) and Message-Oriented Middleware (MOM). In a system with multiple programming languages, operating systems, and requirements on dynamic deployment and reliability, RPC shows its main disadvantage in general compatibility with several programming languages. It does not allow[§§] to natively implement heterogeneous middleware supporting different programming

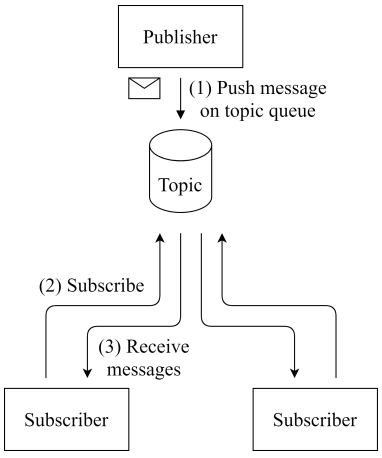


Figure 4.25: Publish-Subscribe model.

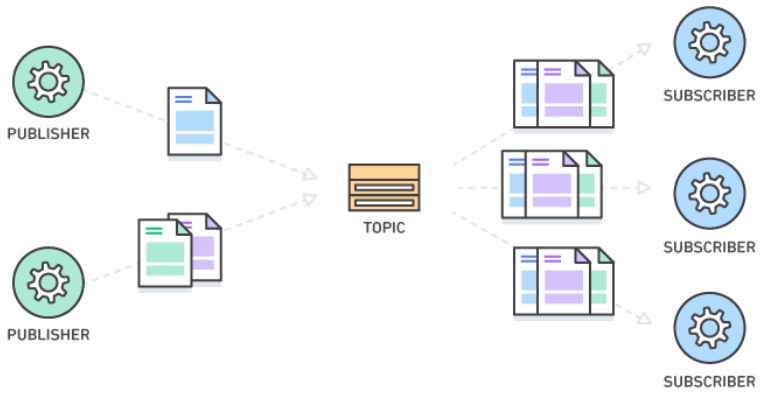
[§§] at the best of our knowledge, at the time of writing this thesis

languages. Meanwhile, the Message Oriented Middleware provides a standard and homogeneous method to implement a native solution in several languages.

Two interaction models are possible: synchronous and asynchronous. In synchronous communication, when a method is called, the caller must stop and wait (suspending its execution) until the method completes and returns. In asynchronous communication, when a method is called, the caller does not need to stop and wait. It allows the caller to continue processing without any regard for the state of the called method. As a requirement, the asynchronous interaction requires an intermediary to handle the exchanged messages and save the received requests. Normally, this role is taken by a message queue.

One of the messaging models available is the publish/subscribe model (see Figure 4.25). It is a mechanism used to propagate data between anonymous entities. The entity that emits the data is called *publisher*. The entity that receives the data is called *subscriber*. It is possible to have a one-to-many mechanism (see Figure 4.26) in which a message is sent to one subscriber or a many-to-many mechanism (see Figure 4.27) in which a message is sent to multiple subscribers at once.

We use a topic-based approach, in which messages are published to queues called "topics" to ground them by logical channels. Each subscriber will receive all messages published to the topic to which it subscribes. The definition of the available topics is delegated to the publisher, it is in charge to create, handle and destroy the available topics to which subscribers can subscribe.



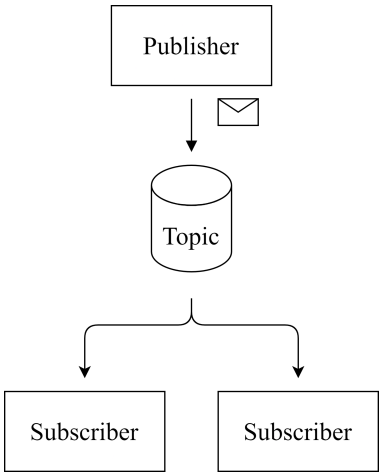


Figure 4.26: Publish-Subscribe one-to-many topology.

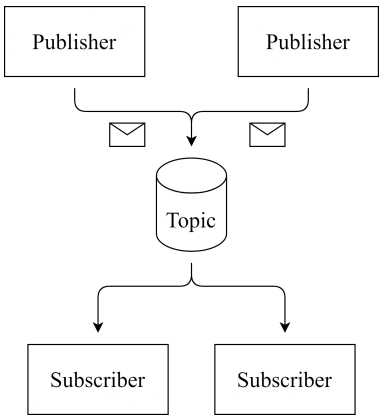


Figure 4.27: Publish-Subscribe many-to-many topology.

Figure 4.28: Publisher/Subscriber overview communication using a topic-based system.

ZeroMQ ∅MQ is an asynchronous messaging open-source library developed by iMatix under the LGPLv3 license with a static linking exception. The library provides a simple and compact socket API. A ∅MQ socket can be used to establish threads, in-process, inter-process, or inter-host (using TCP or multicast) communication. It supports several messaging patterns such as request/reply, client/server, publisher-subscriber (PUB/SUB), and others. The transport layer uses the ZeroMQ Message Transport Protocol (ZMTP) for exchanging messages between two peers over a connected transport layer such as TCP. ∅MQ core is written in C++ but it has bindings and native ports for most modern languages and operating systems.

This feature allowed us to define runtime coordination API in different languages and to abstract the network stack by using ∅MQ distributed queues. Moreover,

since the topology is static and a priori known (defined in the modeling environment), we chose a point-to-point communication to avoid having a router that may introduce a performance bottleneck due to the single-point-of-routing.

∅MQ allows to build messages using different frames. The resulting "multi-part" message is then structured as a single message that will be delivered to the network. The marshaling/unmarshalling processes are then more efficient and the low-latency performance of the library is not compromised. A multi-part message is then composed of two parts: a leading frame that is used as "topic", and a second frame that is used as "content". In a publish/subscribe architecture, the leading frame allows to set "filters" to route the messages only to the right subscribers. The message content structure will be discussed later.

4.4.3.2 Software Architecture

The Cosim20 framework is based on a runtime middleware that eases the development of distributed multi-SUs based on a peer-to-peer communication architecture. The runtime middleware organizes the system as a structured distributed sets of independent software components. The actual runtime middleware is fully developed in Java to achieve some engineering principles as:

- *Portability and easy-to-use*: Cosim20 provides a portable environment that does not depends on third-parties program. It is possible to use the runtime Java library as a dependency on all system supporting Java;
- *Interoperability*: Cosim20 is based on the widely use ∅MQ library, available for a vast number of languages. As a consequence, the developed Java library can interoperate with other Cosim20 Runtime Coordination libraries written in different languages.

Cosim20 provides the library required to execute the coordination algorithm with the basic communication services, and the Java classes required to develop the coordination interface for the simulation units. Each instance of the Cosim20 runtime is composed of a model-specific coordination interface. The set of all the model-specific coordination interfaces is the called *System*. The *System* is in charge to instantiate the model-specific coordination interfaces for each simulation unit and executing them. In case the *System* is in a distributed network environment, is in charge to configure the connection information on the position of each simulation unit. For instance, if simulation units are distributed across a network, the *System* instance should act as a service discovery server and provides the information on the location for each simulation unit. In our case, the automatic compilation process avoids putting in place a service discovery server due to the static configuration of every coordination interface with the needed information (*e.g.* IP address and port on which the coordination interface is available).

In the following paragraphs, we illustrate the runtime architecture of the Cosim20 Java middleware based on the coordination algorithm presented in section 4.4. The developed coordination algorithm can then be developed using other programming languages, depending on the supported ones by ∅MQ library and by the simulation unit. The Java library is composed of three main components:

- *Interface*: It homogenizes an API to communicate with the simulation engine of the simulation unit using a reduced set of methods. In particular, the simulation unit should provide the four methods listed in listing 4.5;

- *Coordination Interface* : It embeds an instance of the SU and the distributed coordination algorithm. It is in charge of the actual data exchange between the SU and the rest of the system using the *Communication Layer*;
- *Communication Layer*: It provides facilities to exchange data among models using a Publish-Subscriber pattern based on the ∅MQ library;

Interface The first component provides a homogeneous interface to handle the execution of the Simulation Unit and to exchange runtime data between the coordination interface and the Simulation Unit. It requires a minimal set of methods to be implemented, in particular, the Java interface (see Listing 4.5).

```
1 public interface Interface {
2   public StopCondition doStep(CoordinationPredicate
      predicate);
3   public Object get(String variableName);
4   public Boolean set(String variableName, Object value);
5   public Boolean simulationIsTerminated();
6 }
```

Listing 4.5: Shows the set of methods that are required to communication with the simulation engine of the Simulation Unit.

The implementation of each method can vary depending on the specific semantics provided by the co-simulation standard or simulator aligning the expected semantics of the methods. In particular, each dedicated interface must be able to implement these abstracted methods following these expectations:

- *StopCondition doStep(CoordinationPredicate predicate)* : This method executes the underlying simulation unit according with the parameter *coordination predicated*. When the predicate is true, a *Stop Condition* is returned to notify the algorithm on the cause of the return;
- *Object get(String variableName)* : This method retrieves the actual value for the specified variable;
- *Boolean set(String variableName, Object value)* : This method updates the actual value of the specified variable with the new value passed;
- *Boolean simulationIsTerminated()* : This method returns true if and only if the simulation unit has terminated its execution;

The Cosim20 runtime middleware already supports two semantically different simulation engines and standards. We developed an FMI 2.0 interface and a Gemoc interface: the first interface allows us to be compatible with a widely use standard and to perform a validation of our approach implementing some use cases, as we see later in Chapter 5, and the second interface allows us to introduce heterogeneous simulation unit developed using DSLs with different semantics and syntaxes. Figure 4.29 shows the two implementations of the classes and the exposed methods. In the FMI interface, we embed the FMU as a simulation unit using the Java library *JavaFMI* [¶¶] to load the FMU and its executable model and to read or write data according to their type using the polymorphic property of Java. It is worth noticing the actual implementation of the *doStep* methods that should match the required predicate from the semantics of FMI.

In particular, in Listing 4.6, we restricted the coordination predicate only using the temporal predicate as the acceptable predicate. The *TemporalPredicate* conforms to the definition given in section 4.4.1. The actual *FMIInterface* class is responsible to accept only the valid temporal predicates. Other predicates, such as Event Predicate, should not be supported by the *FMIInterface* class due to the lack of support by the FMI 2.0 standard. The Δt is then used as a parameter in the *doStep*

[¶¶] https://bitbucket.org/siani/javafmi/wiki/Home

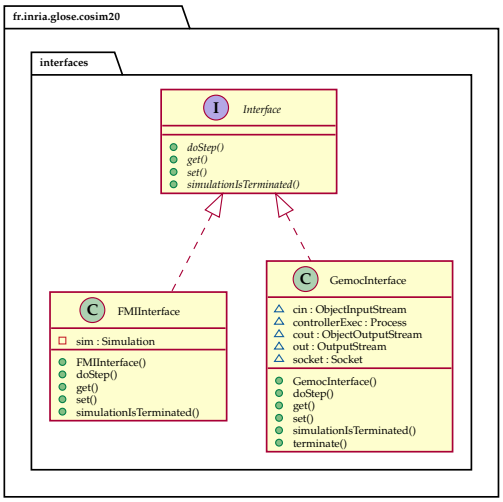


Figure 4.29: Shows an overview of the *Interface* classes.

function of the FMI simulator (line 8). The simulation instance of JavaFMI is in charge to execute the step size on the FMU and returning its new state and return status.

```
1  public StopCondition doStep(CoordinationPredicate
       predicate) {
2  TemporalPredicate tp = predicate.getTemporalPredicate();
3  // Execute the doStep of FMI 2.0
4  Status status = null;
5  if(tp.deltaT == 0){
6    status = sim.doStep(0.0001);
7  } else  {
8    status = sim.doStep(tp.deltaT);
9  }
10 if (status == Status.OK) {
11   return new StopCondition(StopReason.TIME, "", "", ((int)
       sim.getCurrentTime()));
12 }
13   return new StopCondition(StopReason.TIME, "", "",
       ((int)sim.getCurrentTime()));
14 }
```

Listing 4.6: Shows an example of *doStep* method for an FMI 2.0 interface implementation.

The start and end times will be defined later using the *CONFIG* class. Both values are then used to parametrize the simulation of the FMU, defining the start time and the end time. A limitation of the JavaFMI library is for the handling of the set methods: it is not possible to set multiple times a variable without incurring an error. Even if the multiple assignments of a variable are legal in FMI 2.0, it creates the impossibility to update a variable to its current and updated value. For this reason, we introduced a workaround (line 6 in Listing 4.6) that consists to call the *doStep* method with a step size smaller than the accepted one. The step size does not advance the internal time of the FMU but enables to set a variable twice in the same instant of time.

We implemented an interface to support languages developed using Gemoc Studio. It enables us to take advantage of the various semantics developed in Gemoc to validate our approach against the proposed coordination predicates. The proposed implementation is based on a standalone exported model that exposes the API of the Gemoc Execution Engine using a socket used with two streams: one to receive commands and the other to emit the result of the given command. The standalone Gemoc executable is then able to communicate using a protocol defined by Gemoc Studio. As done for the FMI 2.0 Standard,

we developed an interface tailored to the semantics of the protocol and compatible with the proposed *Interface* abstract class. The *GemocInterface* is able to connect to a running model or to launch the JAR file containing the exported executable model. As shown in Listing 4.7, the constructor launches the JAR file (line 3) and then it opens a socket to communicate with the model using two streams. The input stream is used to send command to the model, such as *DoStepCommand* (line 2 in listing 4.8), *getVariableCommand* (line 2 in listing 4.9), and *setVariableCommand*[***].

```
1  public GemocInterface(String path, String host, int port) {
2    try {
3      controllerExec = Runtime.getRuntime().exec("java -jar
       " + path);
4      Thread.sleep(2500);
5      socket = new Socket(host, port);
6      cout = new
       ObjectOutputStream(socket.getOutputStream());
7      cin = new ObjectInputStream(socket.getInputStream());
8    } catch (Exception e) {
9      e.printStackTrace();
10   }
11 }
```

Listing 4.7: Shows the constructor for the *GemocInterface* in the case a JAR, with the executable model and its execution engine, is provided.

```
1  public StopCondition doStep(CoordinationPredicate p) {
2    DoStepCommand doStep = new DoStepCommand(p);
3    try {
4      cout.writeObject(doStep);
5    StopCondition sc = (StopCondition) cin.readObject();
6      return sc;
7    } catch (Exception  e) {
8      e.printStackTrace();
9    }
10   return null;
11 }
```

Listing 4.8: Shows the *doStep* method implementation for the *GemocInterface*.

```
1  public Object get(String varQN){
2    GetVariableCommand getVar = new
       GetVariableCommand(varQN);
3    try {
4      cout.writeObject(getVar);
5      Object varValue = (Object) cin.readObject();
6      return varValue;
7    } catch (Exception e) {
8      e.printStackTrace();
9    }
10   return null;
11 }
```

Listing 4.9: Shows the *get* method implementation for the *GemocInterface*.

Coordination Interface The *Interface* is then used as a homogeneous API to handle the underlying Simulation Unit by the Coordination Interface. The class *CoordinationInterface* implements the distributed algorithm presented in subsection 4.4. It is composed of three main elements: the simulation unit that implements the *Interface* class, the communication layer, and the implementation of the coordination algorithm. As shown in Figure 4.30, each model coordination interface for a specific simulation unit is required to extend the

[***] The implementation of the *set* method is similar to the implementation of the *get* method. Consequently, it is not reported here.

CoordinationInterface inheriting the actual implementation of the coordination algorithm and implementing the methods associated with the interactions that occur. The interactions are group by the type of triggering condition specified on the *Connector*(see section 4.3.4).

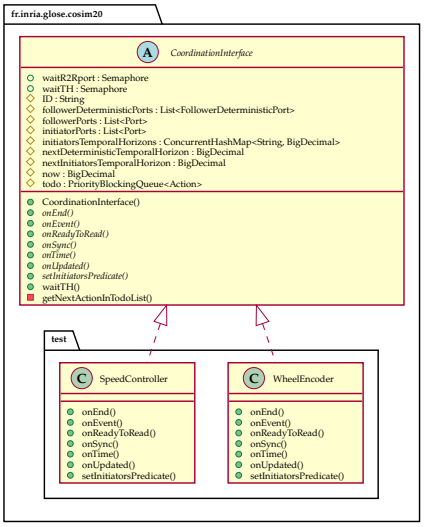


Figure 4.30: Shows an overview of the *CoordinationInterface* class and its extension by two SU coordination interfaces.

An interaction needs at least two ports to exchange the actual value from the *output* port to the *input* port. The communication layer is in charge to provide a transparent and reactive API to exchange data. The implementation reflects our study to minimize the communication among the simulation units without sacrificing the accuracy. For these reasons, we proposed to adopt a publish/subscribe approach to exchange data across the system. It allows us to reduce the number of exchanged messages to the minimum required to perform a correct execution of the co-simulation of the system. Each coordination interface acts as a publisher and as a subscriber, depending on its input or output ports. We decide to associate the publisher entity to the output direction and the subscriber entity to the input direction. The publisher and the subscriber entities are then used by (1) the communication layer to exchange the data value and (2) by the coordination algorithm to perform the coordinated activities of the distributed coordination algorithm. The messages exchanged are formatted according to the JSON format: it allows us to interoperate with other interfaces implemented using a different language. We represent a message using the *Message* class (see Figure 4.31).

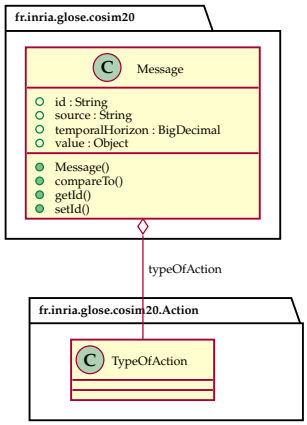


Figure 4.31: Shows the content of a message represented as a Java class.

A message contains five important fields:

- *ID* : Unique identifier used to identify the source port of the message;
- *source* : Unique URI used to identify the source simulation unit. It is used in conjunction with the *ID* to route the message to the correct port;
- *temporalHorizon* : Actual Temporal Horizon of the simulation unit at the instant when the message is sent;
- *value* : (Optional) Resulting value of the requested action. It can be used to store the new value to set or the actual value read from the Simulation Unit;
- *typeOfAction* : Type of *Action* to execute of the simulation unit that receives the message.

A message is then parsed and transformed into an *Action* (see Figure 4.32). An *Action* is an operation that the coordination interface must execute on the simulation unit. An *Action* can derive from a message or from the coordination algorithm (*e.g.* a periodic action is defined in the coordination interface and it does not arrive from an external SUs coordination interface).

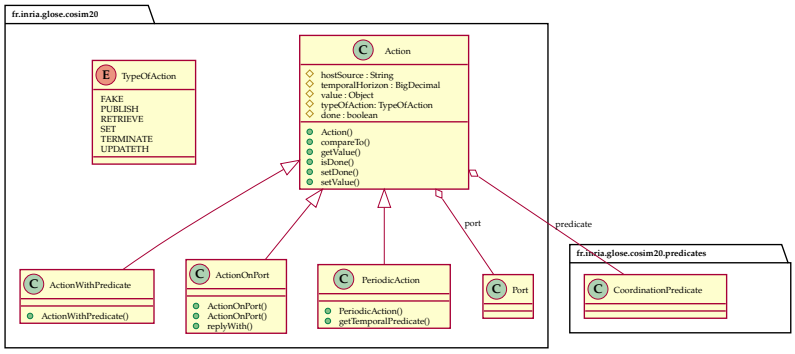


Figure 4.32: Shows the content of a message represented as a Java class.

The action is then pushed into an ordered FIFO queue, prioritized by timestamp, called `todo` list. Then, the implementation follows the Algorithm 4.

A Finite State Machine (see Figure 4.33) is used to implement the communication protocol among the coordination interfaces. In particular, depending on the semantics of some predicates (*e.g.* ReadyToReadPredicate or EventPredicate) and the coordination algorithm.

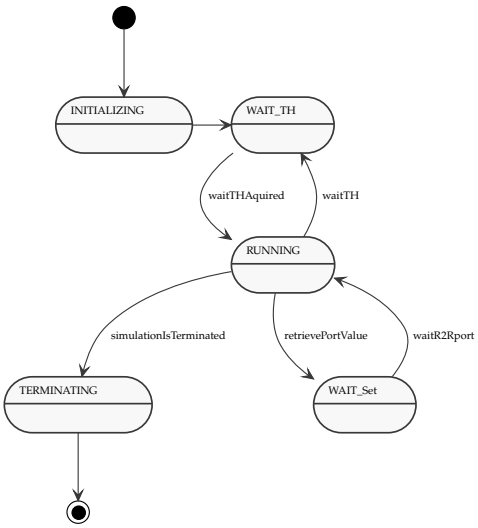


Figure 4.33: Illustrates the Finite State Machine for the Coordination Interface during its execution.

The received actions are then pushed to an ordered-by-timestamp queue that dictates the next action to execute on the simulation unit. In order to respect the synchronization constraint specified in the connector, the first action (the one

with the smallest timestamp) is pulled from the queue and, if the timestamp specified in action holds the constraint defined in the synchronization constraint, then it is removed from the queue and execute. The coordination interface then executes the action: depending on its type, the action can advance the internal simulation of the SU or retrieve the current value of a port.

A possible Java implementation of the algorithm is proposed in Listing 4.10:

```
1  public void algorithm() {
2    Action currentAction;
3    StopCondition sc;
4    CoordinationPredicate actualPredicate;
5    // Initialize initiators predicate
6    initiatorsPredicate = setInitiatorsPredicate();
7    // Update the TH for all local Initiator ports
8    initiatorPorts.forEach(port -> {
9      updateTH(port, now);
10   });
11   // Set the current state to RUNNING
12   state = BIState.RUNNING;
13   // Simulation loop: execute the internal model until the
       simulation is terminated or the co-simulation is halted
14   while (!model.simulationIsTerminated() && state !=
       BIState.TERMINATING) {
15     // Retrieve the next action from the todo list
16     currentAction = getNextActionInTodoList();
17     if (currentAction.typeOfAction ==
       Action.TypeOfAction.TERMINATE  state ==
       BIState.TERMINATING) break;
18     // If the model is already on sync with the time of
       the current action, then it is possible to execute the
       action
19     if (now.compareTo(currentAction.temporalHorizon) == 0)
       {
20       onSync(currentAction);
21       continue;
22     }
23     actualStepSize = computeNextStepSize(currentAction);
24     if ((actualStepSize.compareTo(BigDecimal.valueOf(0.0))
       == 0) //no progress in time
25       && currentAction.temporalHorizon.compareTo(now) > 0
       //nothing to do at that time but waiting
26     ) {
27       todo.add(currentAction); // Reschedule current
       action on the queue
28       // wait that all the port I follow have put their TH
29       waitTH();
30       continue;
31     }
32     // If the action contains a temporal predicate then it
       computes the size of the next safe step size
33     TemporalPredicate nextTimeToStopPredicate = new
       TemporalPredicate(actualStepSize.intValueExact());
34     if (followerPorts.isEmpty() &&
       followerDeterministicPorts.isEmpty()) {
35       nextTimeToStopPredicate = new
       TemporalPredicate(CONFIG.InfOfSimulation);
36     }
37     // If the simulation unit is an initiator, then the
       predicate must include the initilized initiator
       predicate
38     if (initiatorsPredicate != null) {
39       actualPredicate = new
       BinaryPredicate(nextTimeToStopPredicate,
       initiatorsPredicate,
       BinaryPredicate.BooleanBinaryOperator.OR);
40     } else {
```

```
41        actualPredicate = nextTimeToStopPredicate;
42      }
43      // If the currentAction can be reached, execute the
        doStep method and get the StopReason
44      sc = model.doStep(actualPredicate);
45      // Retrieve the current internal time, it will use to
        determine the actual stop reason
46      now = Utils.toBigDecimal(sc.timeValue);
47      if (now.compareTo(CONFIG.EndOfSimulation) >= 0) break;
48      // The timestamp of the current action is aligned with
        the internal time of the FMU
49      if (sc.stopReason == StopReason.TIME) {
        onTime(currentAction, sc); }
50      else if (sc.stopReason == StopReason.EVENT) {
        onEvent(currentAction, sc); }
51      else if (sc.stopReason == StopReason.READYTOREAD) {
        onReadyToRead(currentAction, sc); }
52      else if (sc.stopReason == StopReason.UPDATE) {
        onUpdated(currentAction, sc); }
53      if ((!currentAction.isDone())) {
54        todo.add(currentAction); //was not done this time,
        reschedule.
55      }
56    }
57    // End of the simulation, let the simulation unit
        terminate safely
58    onEnd();
59    // Message all the connected models that the simulation
        is terminating
60    followerDeterministicPorts.forEach(
61      port -> {
62        Message m = new
      Message(Action.TypeOfAction.TERMINATE, port.ID,
      "tcp://" + hostname + ":" + this.port,
      CONFIG.EndOfSimulation);
63        sendMessage(publisher, m, port.ID);
64    });
65    initiatorPorts.forEach(
66      port -> {
67        Message m = new
      Message(Action.TypeOfAction.TERMINATE, port.ID,
      "tcp://" + hostname + ":" + this.port,
      CONFIG.EndOfSimulation);
68        sendMessage(publisher, m, port.ID);
69    });
70    followerPorts.forEach(
71      port -> {
72        Message m = new
      Message(Action.TypeOfAction.TERMINATE, port.ID,
      "tcp://" + hostname + ":" + this.port,
      CONFIG.EndOfSimulation);
73        sendMessage(publisher, m, port.ID);
74    });
75  }
```

Listing 4.10: Implementation of the distributed algorithm in Java.

4.4.3.3 Automatic Runtime Model Coordination Interface Generation

In order to reduce the effort to build a correct co-simulation, we develop an automatic generation process that, starting from the Model Coordination Interfaces and the Model Coordination Specification, is able to generate the corresponding runtime co-simulation. The process is based on the template engine provided by the Xtext framework. It generates the corresponding source code in Java for each

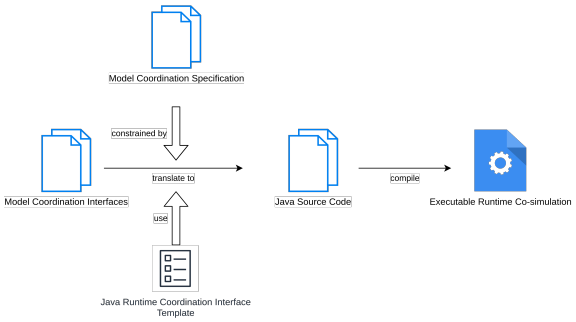


Figure 4.34: Overview on the automatic generation process.

Model Coordination Interface. Figure 4.34 illustrates the workflow for the entire process: starting from the specification of the MCIs and the MCS, it is possible to translate the exposed information into a runnable Java source code using a global RCI template. An RCI template contains some common parts, such as the logical structure. The customization is done using the data expressed in the MCI such as the set of ports and their properties, the set of connectors that link ports, and the topology of the system. A customized RCI is then created for each MCI. Once the process has been completed, it is possible to compile and execute the resulting executable system using the built-in Java compiler of the Cosim20 Studio. An advantage of the template-based approach is also the generation speed: for small-medium size systems, the process can be done in the background having a synchronized generated sources with the loaded coordination model. It enables fast development and verification of the coordination model. Due to the template-based approach, the generation of the source code can be customized using other languages. If an existing implementation of the coordination algorithm exists, then the creation of a template and the customization can be done using as an example the template done for the Java language.

In the next chapter, we validate our proposition by illustrating the use of the Cosim20 framework and the developed IDE on a representative use-case.

4.5 Conclusion

In this chapter, we have presented the three parts of our proposition: the Model Coordination Interface, the Model Coordination Specification, and the distributed coordination algorithm. In the first section, we presented the Model Coordination Interface to partially exposed the semantics and the syntax of the simulation unit. Along with the set of ports and their properties, such as name, direction, and type, we introduced the notion of data-nature: it enables to expose the semantics of the value of the associated port. We developed an Integrated Development Environment to support the MCI specification by giving a DSL called MCILang that eases its definition.

Then, a system designer needs to define the coordination model of the system. Relying on the previously defined MCI for each simulation unit, it is possible to develop a semantics-aware coordination model among the different simulation units. In particular, we introduced the Model Coordination Specification as a set of connectors that provided the coordination "*glue*" between two or more simulation units. A connector is based on three main concepts: *interactions*, *triggering condition*, and *synchronization constraint*. The set of interactions define the dataflow connection among the ports defined on the MCIs, based on their direction. The triggering conditions take into account the data-nature (defined on the interested port) to specify a semantics-aware condition on which the interactions should occur. The synchronization constraint prevents to exchange of data when the two simulation units are not synchronized. It allows defining the synchronization constraint that must hold in order to perform the interactions of the connector.

However, the Model Coordination Specification cannot be executed to co-simulate the system. We provide a template-based generator that, based on the information in the MCIs and the defined MCS, is able to generate an executable specification written in Java. The executable framework is based on a distributed coordination algorithm that exploits the proposed semantics-aware API to execute each simulation unit. In the next chapter, we validate our approach by using as use-case a heterogeneous CPS of a CPU cooling system. We use the proposed Integrated Development Environment to define the Model Coordination Interface for each SU and then to define the Model Coordination Specification. Based on those specifications, we are able to automatically generate an executable co-simulation.

5 Validation

In this chapter, we discuss a representative use-case used to validate our propositions. The validation of the proposed approaches is divided into four parts: the definition of the Model Coordination Interface for every component, the definition of the Model Coordination Specification, the generation of the framework, and the execution and analysis of the results.

We used the management of a CPU temperature as a simple but representative case study. This system is made up of 3 simulation units. *CPUinBoxWithFan* and *fanControler* have been developed in the OpenModelica tool* to respectively define the CPU in a box which is cooled by a fan and the controller of the fan speed (a simple Proportional controller). The heat between the box and the CPU is transferred according to the fan speed. The *overHeatController* has been developed as a state machine in the GEMOC studio†.

We organize this chapter as follows: in the next section we present the CPU Cooling System and we detail its components. We present the corresponding Model Coordination Interfaces. We then illustrate the proposed Model Coordination Specification for the system and the three types of connectors used. Finally, we discuss the results obtained by the co-simulation and we conclude.

* https://openmodelica.org.
† http://eclipse.org/gemoc.

5.1 Use Case: CPU Cooling System

The CPU Cooling System use-case[‡] is composed of three components (see Figure 5.1): a logic controller written in TFSM, a DSL dedicated to modeling timed finite state machine, a fan controller written in Modelica, and a plant modeled as a box composed by a CPU and a fan, written both a single component in Modelica. The resulted system is then constituted by three models written in two different languages that conform to two different semantics. Furthermore, we make a hypothesis that we are dealing with black-box components: the two Modelica models are then exported as two FMUs that embed an executable model. The logic controller is exported as an executable binary that conforms to the Gemoc API for coordination.

The resulting three simulation units are then ready to be used in the CPU Cooling System. In the first part, we define for each SU the corresponding Model Coordination Interface. In the second part, we define three different types of connectors, based on the data nature of the ports exposed in the MCIs. In particular, the first connector defines a time-trigger co-simulation by specifying the most appropriate sample rate to exchange data between the fan controller and the box. In the second connector, we define the coordination model between the logic controller and the box by specifying an event-driven co-simulation on an exposed event. In the third connector, we define the coordination between the box and the logic controller as a feedthrough loop where the controller exposes the requirement to read the most updated value at a specific point in time. In the third part, we generate the corresponding runtime framework based on the specifications written in the first and second parts. Finally, in the fourth part, we execute and analyze the results of the co-simulation.

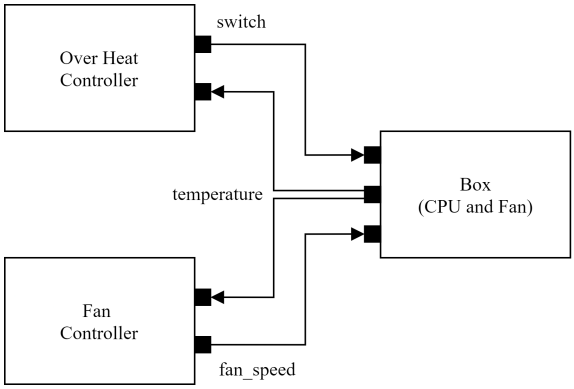


Figure 5.1: Overview of the CPU cooling system used as use case.

Box with CPU and Fan The subsystem composed of a CPU and a fan is embedded into a box and it is modeled using OpenModelica (see Figure 5.2). In the *CPUinBoxWithFan* simulation unit, the CPU is activated as long as the *isStopped* input is equal to false. When activated, the CPU produces heat, which is exchanged with the air of its box more or less rapidly depending on the *fanSpeedCommand* input ($\in$ [0..10] where at 0 the fan is stopped and at 10 the fan is at full speed).

Using the proposed Model Coordination Interface, we define an interface as following:

[‡] the associated code can be retrieved from https://github.com/giovanni-liboni/cosim20-CPU-cooling-system.

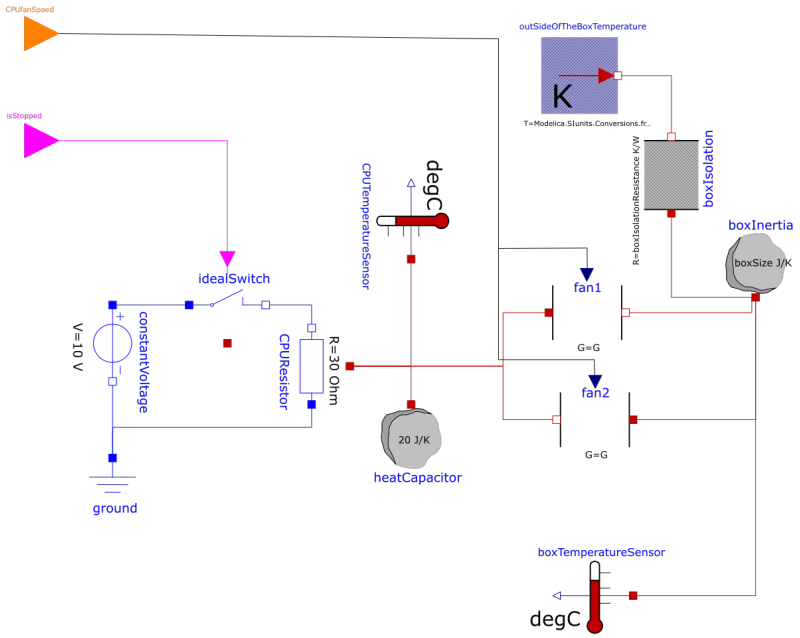


Figure 5.2: Modelica model representation of a box with CPU and Fan.

```
1  Interface CPUinBox {
2    FMUPath "su/CPUinBoxWithFanHeatModel.fmu"
3    ports {
4      Port "BoxTemperature" {
5        direction INPUT
6        nature constant
7        type Real
8        initValue "25"
9      },
10     Port "CPUTemperature" {
11       direction OUTPUT
12       nature continuous
13       type Real
14       initValue "0.0"
15     },
16     Port "CPUfanSpeed" {
17       direction INPUT
18       nature continuous
19       type Real
20       initValue "0"
21     },
22     Port "isStopped" {
23       direction INPUT
24       nature piecewiseConstant
25       type Boolean
26       initValue "false"
27     }
28   }
29   temporalreferences {
30     ModelTemporalReference t {
31       reference time
32     }
33   }
34 }
```

Listing 5.1: MCI textual representation of the model described in Figure 5.2.

In the Model Coordination Interface defined in Listing 5.1, we expose all the information such as the set of ports with their properties (from line 3 to line 28), the temporal reference used in the model (line 30), and the path of the executable model (line 2). According to the Modelica model (Figure 5.2), the list of ports

reflect the exposed variables. For instance, *CPUfanSpeed* and *isStopped* are defined as `INPUT` ports of type `Real` and `Boolean`, correspondingly. The nature is defined accordingly with the use of the variable in the model: the *CPUfanSpeed* defines the speed of the fan and it's directly use to set the speed of both fans in the model. In the other case, the *isStopped* controls an ideal switch and its value should change instantaneously.

Fan Controller The *Fan Controller* model is written in Modelica and developed in OpenModelica. We use the integrated simple continuous controller PID in the Modelica library. As illustrated in Figure 5.3, it represents a PID controller with limited output, anti-windup compensation, and setpoint weighting. The gain of controller k is driven by the input variable Kp. The time constant Ti of the integrator block is set to 0.5. The time constant Td of the derivative block is set to 0.1. The output y is then limited between 0 and 10. The output of the controller is transformed from the *Real* value to an *Integer* value.

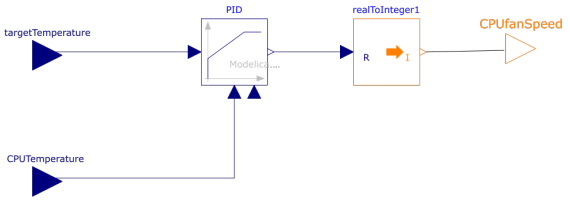


Figure 5.3: Modelica model representation of the Fan Controller.

To use the model in our framework, we export the model as an FMU. Based on the FMI model description, we then automatically generate the corresponding Model Coordination Interface using the importing feature of the MCILang IDE. The resulting MCI of the Fan Controller is represented in Listing 5.2.

```
1  Interface fanController {
2    FMUPath "su/FanController.fmu"
3    ports {
4      Port "CPUTemperature" {
5        direction INPUT
6        nature continuous
7        type Real
8        initValue "25"
9        monitored false
10     },
11     Port "CPUfanSpeed" {
12       direction OUTPUT
13       nature continuous
14       type Real
15       monitored false
16     },
17     Port "targetTemperature" {
18       direction INPUT
19       nature constant
20       type Real
21       initValue "65"
22       monitored false
23     },
24     Port "Kp" {
25       direction INPUT
26       nature constant
27       type Real
28       initValue "5.0"
29       monitored false
30     }
31   }
32   temporalreferences {
33     ModelTemporalReference t {
```

```
34        reference time
35      }
36    }
37 }
```

Listing 5.2: MCI textual representation of the model described in Figure 5.3.

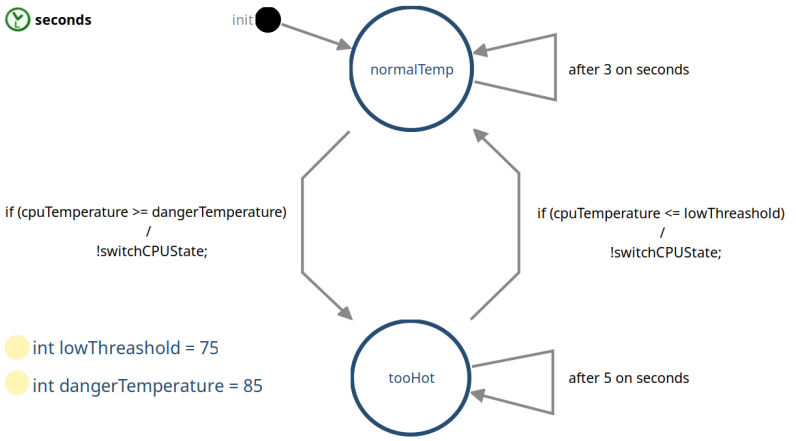


Figure 5.4: Over Heat Controller state machine.

Over Heat Controller The *overHeatController* model is written using a DSL developed in Gemoc Studio. The controller is then defined as a state machine. It monitors periodically (every 3 seconds) the *cpuTemperature* and if it exceeds a specific threshold, the *switch* event occurs and the state machine enters in a new state where it monitors the CPU temperature every 5 seconds. If it goes above a specific threshold, the *switch* event occurs and the state machine enters the first state (see Figure 5.4). The resulting model is then exported as an independent simulation unit.

```
1  Interface overHeatController {
2    GemocExecutablePath "su/overHeatControler.jar:39635"
3    ports {
4      Port "SwitchCPUState" {
5        variableName "CPUprotection::switchCPUState"
6        direction OUTPUT
7        nature transient
8        type Boolean
9        initValue "false"
10       ioevents {
11         Triggered occurs
12       }
13     },
14     Port "CPUTemperature" {
15       variableName "CPUprotection::cpuTemperature"
16       direction INPUT
17       nature piecewiseConstant
18       type Integer
19       initValue "25"
20       ioevents {
21         ReadyToRead readyToRead
22       }
23     }
24   }
25   temporalreferences {
26     ModelTemporalReference t {
27       reference time
28     }
29   }
```

```
30 }
```

The resulting MCI for the *overHeatController* is presented in Listing 5.3. Compared to the interfaces presented previously, there are two new elements: *variableName* and *ioevents*. In line 5, the *variableName* attribute links the internal event *switchCPUState* with the port of the same name and expose through the interface the specific event of type *triggered* named *occurs* (line 11), which will be triggered when the internal event will be internally triggered. In the second case, the port will be linked to an internal variable (line 15) and the associated ioevent will be of type *ReadyToRead* (line 21). In this case, the specific ioevent will be fired before the expression on the guard is evaluated for the two transactions to and from the internal states `normalTemp` and `tooHot`.

Listing 5.3: MCI textual representation of the model described in Figure 5.4.

5.1.1 Model Coordination Specification

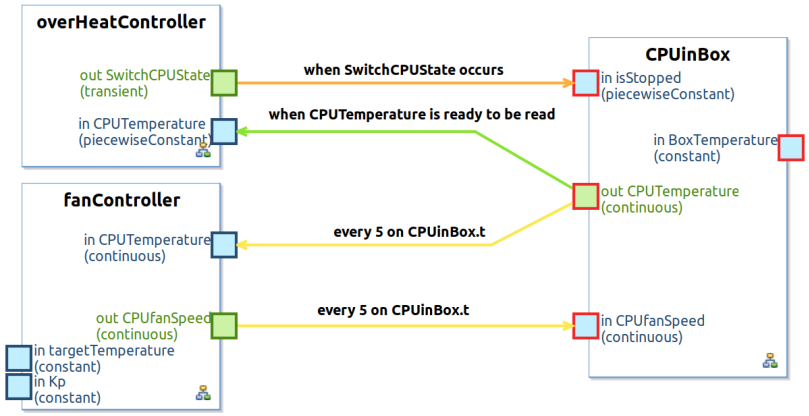


Figure 5.5: CPU Cooling System represented in Cosim20 Modeling Environment. If a port is monitored and logged, a red border is added to the visual representation of the port.

To connect the different simulation units we relied on strategies defined in [11]. Consequently the temperature from *CPUinBoxWithFan* to *overHeatController* is only exchanged when the later simulation unit is ready to read the data. Similarly, the change of the *isStopped* input is only done only when the *switch* event occurs. Between the two simulation units obtained from Modelica, the connectors define classical time trigger communication. The corresponding specification using the proposed language MCL is illustrated in Listing 5.4.

```
1  load "overHeatController.mbi" as ctrl
2  load "fanController.mbi" as pid
3  load "CPUinBoxWithFan.mbi" as plant
4
5  Connector cpuTemp
6    when every 5 plant.t
7    sync pid.t = plant.t
8    do
9      plant.cpuTemperature -> pid.cpuTemperature
10
11 Connector fanSpeed
12   when every 5 plant.t
13   sync pid.t = plant.t
14   do
15     pid.fanSpeedCommand -> plant.fanSpeedCommand
16
17 Connector cpuTemp2
18   when value on ctrl.cpuTemperature is ReadyToRead
19   sync ctrl.t = plant.t
```

```
20   do
21     plant.cpuTemperature -> ctrl.cpuTemperature
22
23 Connector switchCPUState
24   when event on ctrl.SwitchCPUState occurs
25   sync ctrl.t = plant.t
26   do
27     plant.isStopped -> not plant.isStopped
```

Listing 5.4: MCL textual specification for the system in Figure 5.5.

We handled the different cases by using different Predicates in the *doStep* function call. However, one can notice that the coordination algorithm will not be generic anymore but dedicated to the topology of simulation units and the information on the connectors. For this specific use case, the coordination algorithm is provided on Listing 5.5. Lines 4 to 6, the predicate for the *overHeatController* simulation unit is defined as "the variable cpuTemperature is ready or the switch event occurs". Line 7, the *dostep* function is called and in lines 8 to 16 the result of the function is managed. If the simulation was paused due to the variable cpuTemperature which is ready to be read, then a function (*simulateBoxAndFanControl* defined line 19) is called to set the *CPUinBoxWithFan* simulation unit at the same time as the overheat controller simulation unit. Once done, the expected value is exchanged between the FMU. If the simulation was paused due to the occurrence of the *switch* event, then the receiving simulation unit is at the time when the event occurred, so the *isStopped* variable is changed. The temporal connector between the fan controller and the CPU, as defined in Figure 5.5, requires the simulation of both models until a specific point in time. In lines 21 to 36, the simulation units must reach an *expectedTime*. If there is one (or several) intermediate temporal steps in between now and the expected time (i.e., $now\%5 = 0$ in our case), then the simulation units are simulated until this point in time, and data are exchanged as expected.

```
1  public void coSimulate(double endtime) {
2    //now = 0; localIsStopped = false;
3    while (now < endTime){
4      ReadyToReadPredicate r2rp("cpuTemperature");
5    EventPredicate ep ("switch");
6    BinaryPredicate bp (r2rp, ep);
7    StopCondition sc = controlerSU.doStep(bp);
8    if (sc.stopReason == READYTOREAD) {
9      simulateBoxAndFanControl(sc.stopTime);
10     double cpuTemperature = c.boxSU.read("cpuTemperature");
11     controlerSU.setVariable("cpuTemperature",
       cpuTemperature);
12   } else { //event occured
13     simulateBoxAndFanControl(sc.stopTime);
14     localIsStopped = !localIsStopped;
15     boxSU.write("isStopped").with(localIsStopped);
16   }
17   }
18 }
19
20 public void simulateBoxAndFanControl(double expectedTime) {
21   double delta = expectedTime - now;
22   //\Delta t == 5 for each connector from boxSU and
       fanControllerSU
23   while (delta + (now % 5) >= 5) {
24     double stepToDo = (5-(now % 5));
25     boxSU.doStep(stepToDo);
26     fanControllerSU.doStep(5);
27     double cpuTemperature = boxSU.read("cpuTemperature");
28     fanControllerSU.write("cpuTemperature")
29                    .with(cpuTemperature);
```

```
30      int fanCommand =
        fanControllerSU.read("fanSpeedCommand");
31      boxSU.write("fanSpeedCommand").with(fanCommand);
32      double boxTemperature = boxSU.read("BoxTemperature");
33      now += stepToDo;
34      delta = expectedTime - now;
35    }
36    if (delta > 0) {
37      boxSU.doStep(delta);
38      now += delta;
39    }
40  }
```

Listing 5.5: Coordination Algorithm dedicated to the example on Figure 5.5 using the proposed interface.

Along with the MCI specifications, the MCL is used to generate the executable Java source code that implements the coordination algorithm. The distributed nature of the coordination is achieved by creating tailored wrappers for each simulation unit. In particular, by providing methods to handle the communication protocol to and from the simulation unit. A possible implementation of one of these methods is represented in Listing 5.6, where we implement the *onTime* method for the *Box* simulation unit. For each port, we have the associated action to perform on the simulation unit and the corresponding step for the coordination algorithm.

```
1  @Override
2  public void onTime(Action currentAction, StopCondition sr)
       {
3      if (currentAction.port.compareTo(CPUfanSpeed) == 0) {
4          model.set(CPUfanSpeed.associatedModelVariableName,
5                      (Double) currentAction.getValue());
6      }
7      ...
8      else if
       (currentAction.port.compareTo(CPUTemperatureForCtrl)
       == 0) {
9          double temperature =
       model.get(CPUTemperatureForCtrl);
10         publish(CPUTemperatureForCtrl, temperature, now);
11     }
12     ...
13     currentAction.setDone(); // Mark the current action as
       executed
14     if (currentAction instanceof PeriodicAction) {
15       todo.add(new PeriodicAction(...);
16     }
17 }
```

Listing 5.6: Implementation in Java of the *onTime* method for the simulation unit illustrated in Figure 5.2. The source code is generated from the corresponding MCI and MCL specification for the simulation unit in Listing 5.1 and the coordination specification in Listing 5.4.

The generated wrapper source code for the Box with CPU and Fan is available in Appendix A.2.

Two possible implementations for the *onEvent* and *onReadyToRead* methods are illustrated in Listing 5.7. In the first method, *onEvent*, an event is retrieved from the simulation unit, and then it is handled by publishing it to the subscribers. In this case, we have only a single event that must be exported and so the implementation takes into account only a single instance at the time. In the second method, the *onReadyToRead* method is called only when the associated event is triggered. The method *retrieve* implements a set of function calls needed to implement the correct communication protocol between the simulation unit that has generated the *readyToRead* event and the target simulation unit that must provide the data. The generated wrapper source code for the Over Heat controller is available in Appendix A.3.

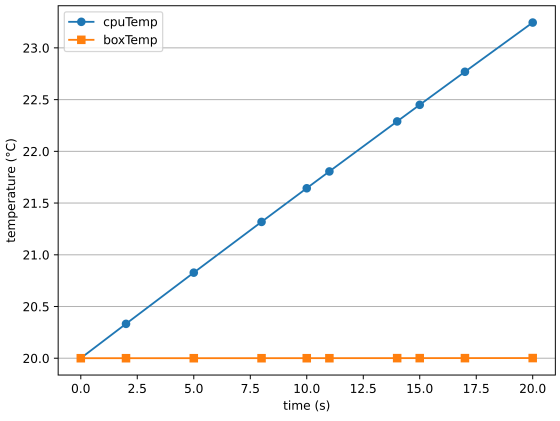
The source code of this use case is available here, along with other test cases and additional technical documentation: https://gitlab.inria.fr/glose/cosim20-java.

```
1       @Override
2   public void onEvent(Action currentAction, StopCondition
      sc) {
3     SwitchCPUState_ctrl2box.setValue(!((boolean)
      SwitchCPUState_ctrl2box.getValue()));
4     boolean value = (boolean)
      SwitchCPUState_ctrl2box.getValue();
5     publish(SwitchCPUState_ctrl2box, value,
      Utils.toBigDecimal(sc.timeValue));
6   }
7
8   @Override
9   public void onReadyToRead(Action currentAction,
      StopCondition sc) {
10    retrieve(portMap.get(sc.objectQualifiedName), now);
11    currentAction.setDone();
12  }
```

Listing 5.7: Implementation in Java of the *onEvent* and *onReadyToRead* methods. The source code is generated from the corresponding MCI and MCL specification for the simulation unit in Listing 5.3 and the coordination specification in Listing 5.4.

5.1.2 Results

The results from the beginning of the co-simulation obtained with this setup are provided in Figure 5.6. The reader should notice that the points are only retrieved as specified in Figure 5.5, i.e., at the exact time, it is needed to have a correct co-simulation. For instance on Figure 5.6, we can see that a first paused was realized by the overheat controller at time 2, i.e., which is the non-deterministic time spent for the state machine to enter in the *normalTemp* state, where the guard of output transition is evaluated and consequently the CPU temperature is read. Then, pauses are realized every 5 seconds and every multiple of 3 (the reading period in the first state of *overHeatController*). This way, we reduce the number of communication points to their strict minimum to have a correct co-simulation and we avoid the delays introduced by the classical sampling strategy.

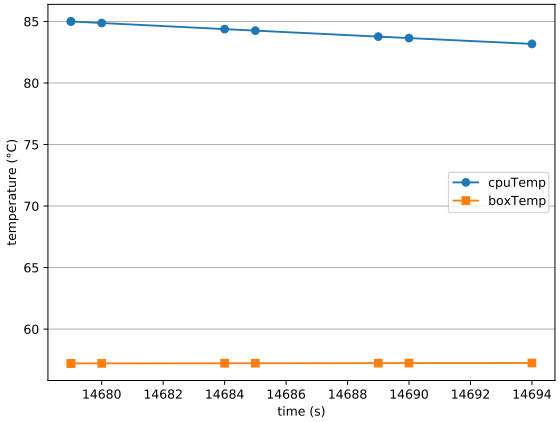


	Temperature (°C)	
Time (s)	CPU	Box
0	20	20
2	20.33212	20.00001
5	20.82698	20.00009
8	21.31794	20.00023
10	21.64328	20.00035
11	21.80536	20.00042
14	22.28902	20.00067
15	22.44940	20.00076
17	22.76890	20.00097
20	23.24500	20.00133

Figure 5.6: Results obtained at the begin of the co-simulation.

In the Figure 5.7, the first point in time is the one when the state machine switch from the *normalTemp* state to the *tooHot* state. It occurred at the time 14679. Consequently, as long as the state machine remains in this state, data are retrieved every 5 seconds as specified in the temporal connectors and in the reading period from the state machine. However, since the state machine entered in the *tooHot* state at time 14679, then the simulation unit was paused after 5 seconds, i.e., at

14684, while the temporal connectors induce a pause every 5 seconds. We can see here that the internal semantics of the simulation is consistently exposed and took into consideration.



	Temperature (°C)	
Time (s)	CPU	Box
14679	85.00110	57.21210
14680	84.87561	57.21432
14684	84.37902	57.22311
14685	84.25629	57.22527
14689	83.77094	57.23384
14690	83.65101	57.23595
14694	83.17671	57.24429

Figure 5.7: Results obtained when the controller enters in the *tooHot* state.

Finally, in Figure 5.8, the simulation is run for 8 hours and 20 minutes (30000 seconds). For this simulation, we obtained 15023 communication steps without sacrificing accuracy over performance. If we were using a time-triggered interface and allowed an error up to 100ms, then we would have 300'000 communication steps and a loss of accuracy.

5.1.3 Discussion

We argued that the proposed interface is extendable, efficient, and intuitive to use. In this section, we discuss some of these points according to our experiment in implementing the API in the GEMOC studio.

Concerning the implementation of the predicates, two main points can be addressed. First, its efficiency strongly relies on how the API is internally implemented. In our case, we modified the code generation to generate a pause when needed. For instance, for the *Updated* predicate, all assignments are instrumented to create a pause. This has only a minor impact on performance. However, if the implementation is done in a wrapper where all micro-steps are checked to see if a variable has been updated, then the execution may suffer from a slowdown. The same phenomenon happens for the Threshold predicate. If one samples the variable to check the crossing, the execution will be slow down and the exact point in time when the crossing occurs may be missed. It is better to inject the actual zero crossings in the model (typically in the equation set) to ensure better performance and accuracy. This is what is expected to be done in collaboration with Safran. Also, the implementation of the predicates must follow the semantics of the simulation unit. For instance, if a simulation unit is executing a model developed in a synchronous language [183], then all the assignments should NOT be caught since according to the synchronous semantics, data are latched at specific points in time. In our implementation, we relied on annotations to provide flexibility on the exposed semantics. Consequently, the tool developer is in charge of providing the expected semantics.

Concerning the extension of the predicate, there are two minors points to take care of. First, it is important to rely on a mechanism to clearly specify which predicate is supported for a specific simulation unit. This may for instance be done in an artifact equivalent of the FMI model description. Second, there is a

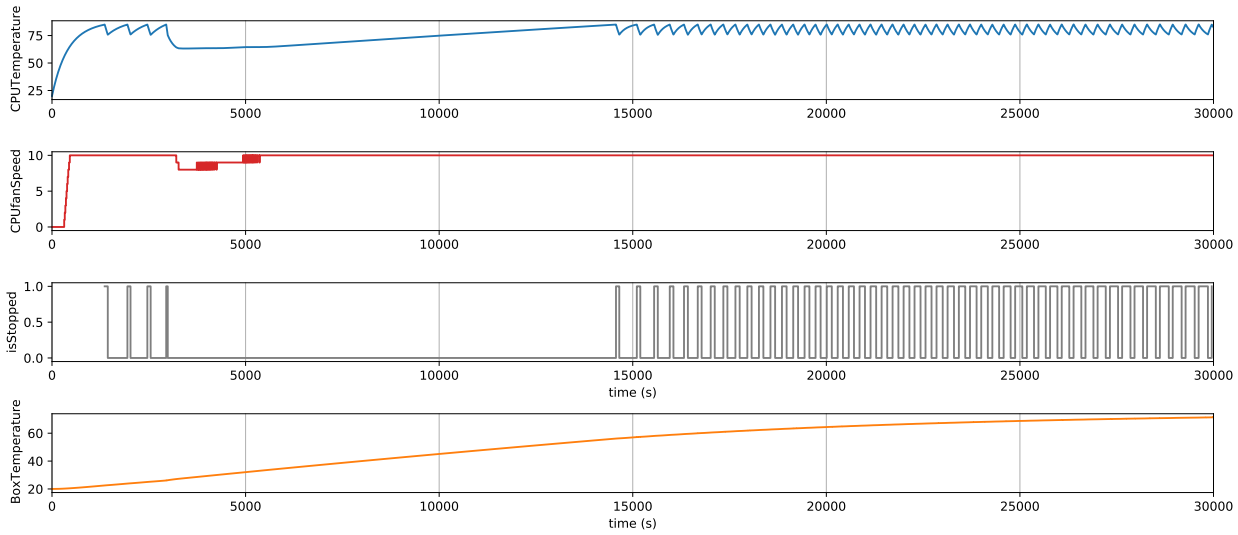


Figure 5.8: Results obtained when running the coordination algorithm.

risk of an uncontrolled evolution of predicates, leading to a predicate tower of Babel. This is a long-term issue and we believe there are few risks it happens. If the road to this situation is taken, it may be interesting to provide an official set of predicate extension repository, where people can look for existing predicate before creating their own and where all predicates are put together.

Concerning the size of the use-case, we choose to use a simple but representative use-case to focus on the interactions between simulation units. It enables us to study the specific behaviors and coordination models without introducing the development complexity of an industrial use case. Due to our interest in the coordination aspect of co-simulation, we propose to validate our approach on a system that experiences all the coordination problems that we identify in subsection 3.5.2.

5.2 Use Case: Fault Injection Simulation

In this use case, we extend the previous systems by adding a simulation unit that injects in the system faulty values to test the behavior in case of a sensor fault (see Figure 5.9). A first we introduce the main concepts of fault injection and its application in the context of Model-Driven Engineering for co-simulation. We then introduce the fault injector simulation unit written in Python that implements an interpreter for a simple DSL to inject values into the co-simulation at a specific point in time.

Fault Injection Fault injection is a testing approach to evaluate dependability. In safety-critical CPS is an important phase of the development due to the high impact in case of failure of the system, such as loss of human being life or environmental damages. The Fault Injection techniques can be classified into three categories based on their implementation: hardware-based, software-based, and simulation-based.

Hardware-based fault injection approaches inject faults at the physical level by changing environment parameters to examine their effects. The emulation of real-life faults is done by disturbing the power supply, stuck-at transistors,

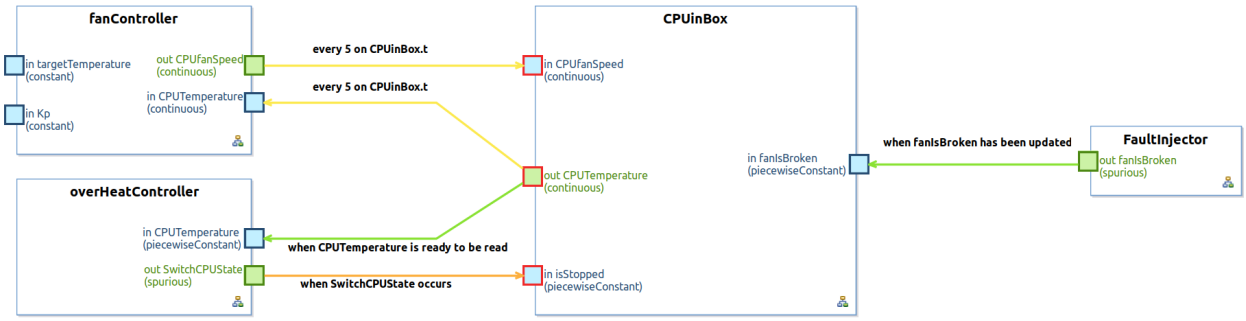


Figure 5.9: Overview of the system with a Fault Injector component.

creating external voltage or current changes, electromagnetic interference, or heavy-ion radiation. For instance, the heavy-ion radiation induces changes in the hardware at the memory level, such as in CPU registers or the main memory, corrupting both program and data memory.

Software-based fault injection approaches inject faults into the software. In contrast with hardware-based techniques, they do not require a physical device, allowing less expensive tests. Moreover, they are used to test programs and operating systems logic by introducing the fault injector in a layer between the hardware and the OS for the former, and by embedding the fault injector into the OS for the latter. Given the flexibility of software, different types of faults can be injected, for instance, disturbances in conditions or flags, loss or duplicated messages, memory faults, or errors in timing.

A fault injection tool can rely on a different trigger mechanism that produces an artificial fault or error and inserts it into the normal co-simulation execution. In particular, we take the classification given by [184], in which the authors classify the fault injection approaches into three types:

- *Execution-driven*: It dynamically occurs at runtime depending on the control flow of the program;
- *Location-based*: It consists of writing values in specific memory locations in order to corrupt it;
- *Time-based*: It occurs at runtime at a specific point in time or predetermined intervals.

In particular, simulation-based fault injection approaches introduce faults into the simulation unit that represents the system-under-test by changing parameters and values. These approaches are used mainly in the context of embedded systems where hardware models are written using Hardware Description Languages (HDLs), such as Verilog [185] or VHDL [186] and hardware faults can be emulated by injecting faults into the model. With the increasing interest in MDE and CPS, this technique is then used to inject faults into models that conform to different formalisms.

In this use-case, we use a model-based approach in which a SU injects faults at specific points in time to test the resilience of the controllers to a faulty sensor as shown in Figure 5.9.

The Fault Injector simulation unit is composed of a faults specification and an interpreter written in Python. The faults specification is written using a simple DSL that defines at which instant the defined variable changes its value (*i.e.* time-based fault injection). The current version implements a *stuck-at* semantics. For instance, Listing 5.8 drives the variable *fanIsBroken* to enable the fan at the

```
@0 fanIsBroken false
@1900 fanIsBroken true
```

Listing 5.8: Faults specification executed by the fault injector SU.

```
load "CPUinBox.mbi"
load "FaultInjector.mbi"

Connector faultInjector
      (from FaultInjector.fanIsBroken to
  CPUinBox.fanIsBroken)
  when value on FaultInjector.fanIsBroken has been
  Updated
  do
    FaultInjector.fanIsBroken -> CPUinBox.fanIsBroken
```

Listing 5.9: MCL specification representing the connector that defines the coordination between the fault injector simulation unit (*FaultInjector*) and the CPU cooling system (*CPUinBox*).

beginning and then disable it when the internal time of the box reaches 1900 seconds.

Based on the previous example, we add a connector from the fault injector to the box: the boolean piecewise-constant port *fanIsBroken* is connected with the *fanIsBroken* input port exposed by the box. In this case, we take advantage of the piecewise-constant data-nature and the interpreter capability to define a connector that exploits the *Updated* event.

To support this scenario, we update the Modelica model that represents the box with the CPU and the fan adding a thermal swith driven by the input variable *fanIsBroken* (see Figure 5.10). It allows to disable or enable the fans according to its boolean value.

In Figure 5.11, we show the results of the co-simulation of the system. In the beginning, both fans are working normally: the temperature is under control and it increases slowly. However, at the instant 1900, we inject a fault to simulate that one of the fans breaks. Once applied, the temperature quickly increases and it reaches the maximum allowed temperature. When the overheat controller senses the maximum temperature, the fan is activated but it cannot lower the temperature as done previously, as shown in Figure 5.8.

We illustrate a simple use-case but still representative: we introduce the support to rapid prototype fault injection scenarios and we integrate a Python executable module into the system and the Cosim20 framework thanks to the language interoperability of the communication protocol.

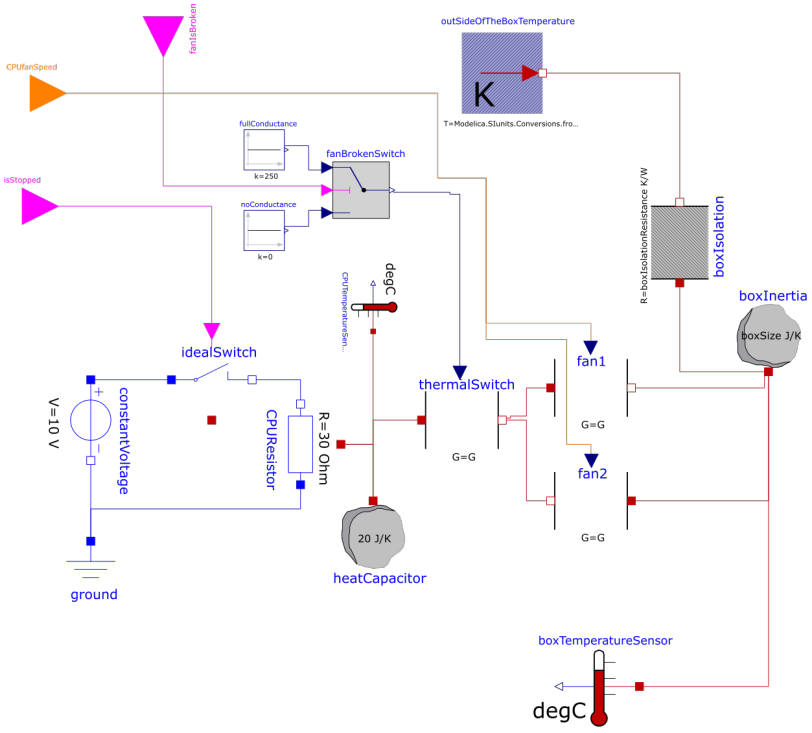


Figure 5.10: Modelica model represeting the box with a CPU and a fan system with a fault injector capability. The *fanIsBroken* input allows to enable or disable the fans according to its boolean value.

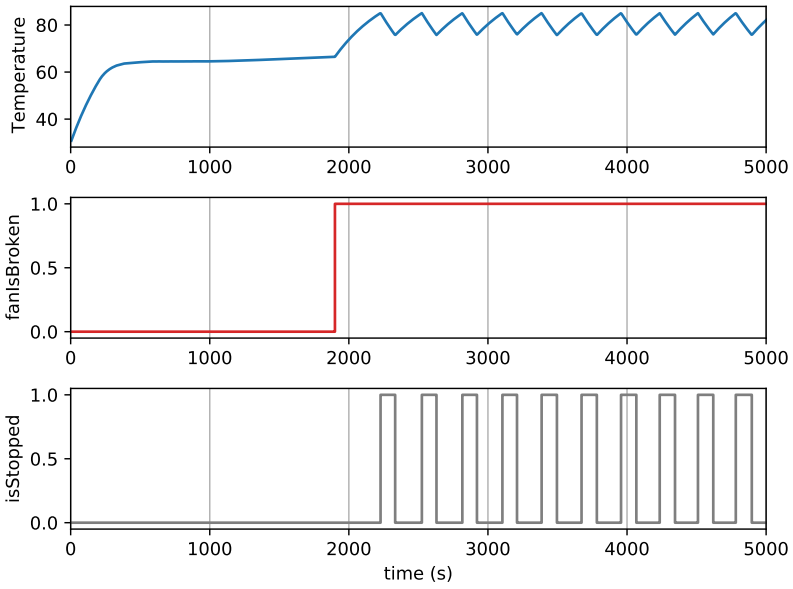


Figure 5.11: Results obtained by applying the fault at instant 1900, as specified in Listing 5.8.

5.3 Conclusion

In this chapter, we have validated our framework by implementing two use-cases: a CPU Cooling System that uses three types of connectors, and a CPU Fault Injection System that enables injecting faults at runtime using a simple DSL that uses a different connector in addition. We have shown that those connectors can express a coordination model between cyber and physical models. In particular, we have defined four main connectors: sample-rate, ready to read, event-based, and updated. The first three connectors are used in the first use-case that defines a heterogeneous coordination model by defining the interaction among two Modelica models and a DSL model based on the Gemoc engine. The last connector is used in the second use case where we add a Python-based simulation unit that injects runtime faults.

For each use case, we have used the MCI and MCL specification to generate the corresponding runtime framework. A simulation unit is handled by its generated wrapper that contains a distributed coordination algorithm in charge to coordinate the overall co-simulation. The communication among the simulation unit is based on a communication protocol implemented using the ∅MQ protocol and the corresponding library.

In the next chapter, we conclude by summarizing the contributions of this thesis and by giving some future perspectives on the thesis that could be a path to continue this work.

Conclusion 6

6.1 Overview

The Cyber-Physical Systems development raises important challenges given by the increasing complexity in systems themselves and their development processes. In this thesis, we focused on the modeling phase of the development with a particular focus on the integration of heterogeneous models. Co-simulation techniques enable to validate and verify the emerging system in the early stage of the development but few limitations may occur: the intrinsic heterogeneity of CPS leads to the use of different specialized tools and languages, and the cooperation among different stakeholders and enterprises rises concern for intellectual property protection of the exchanged models. In the co-simulation context, few solutions were proposed to tackle those problems and to allow seamless collaboration and models exchange among entities (*i.e.* enterprises and stakeholders): the FMI Standard and the HLA standard propose homogeneous interfaces to co-simulation continuous and discrete-event based models, respectively. However, in a Cyber-Physical System co-simulation, its heterogeneity limits the usability of homogeneous co-simulation standards due to their limitation in terms of adaptability and correctness.

We have illustrated the main semantics that composed Cyber-Physical systems and the coordination semantics used to simulate and co-simulation those systems in regards to their *natural* semantics. Both continuous-time based and discrete-event based co-simulation approaches present limitations on the integration of heterogeneous systems due to the semantics gap between the native semantics and the adapted semantics. We have shown that this gap can lead to co-simulation errors due to the semantics mismatch of the models. We have noted that the current approaches rely on homogeneous interfaces and provides an adaptation between the semantics. However, these approaches do not take into account the native semantics of the models to define a coordination model capable to correctly exploit the semantics elements of the underlying models.

In Software and System Engineering, coordination-oriented approaches proposed to explicitly define the integration *glue* for models that conform to different semantics. In particular, we have presented Architecture Description Languages and Coordination Languages that focus on the expression of explicit coordination models between different computational entities. To be able to access the underlying semantics, those languages rely on dedicated interfaces that expose elements of the semantics and syntax of the models. We have noted that those interfaces conform to specific semantics. For instance, the FMI Standard supports only continuous-based models and the proposed interface enforces its semantics to the underlying model semantics. In a heterogeneous context, the interface must be able to express the heterogeneity of the system while ensuring the IP protection of the model.

In this thesis, we proposed a framework dedicated to the modeling of co-simulation. The framework is composed of two Domain-Specific Languages and an Integrated Modeling Environment that embeds them. It provides textual and visual editors with completion, syntax check, and graphical editing to

support the two proposed languages. MCL provides a set of rich connectors enabling the definition of correct coordination between simulation units of Cyber-Physical Systems. Based on these definitions, the framework is then able to automatically generate code for a distributed co-simulation. The generated code is based on point-to-point coordination between simulation units. A case study shows the different benefits of the proposed framework. First, by proposing appropriate connectors it allows the designer to define correct coordination, *i.e.* coordination for which the co-simulation does not introduce unexpected delays. This is important since early V&V should not be biased due to the co-simulation framework. Second, it reduces the communication between the units to their strict minimum to ensure correctness. Less communication means better simulation performance. Finally, the high degree of automation in the framework removes the time-consuming task of writing correct coordination.

6.2 Future works

Co-simulation Runtime Performance Despite not being our primary objective, the performance of the generate runtime architecture should be similar to a custom-made solution, both distribute or not. In some cases where the geographical distribution is not a constraint and the size of the system allows to simulate it on a single machine, it makes sense to provide a local architecture or a monolithic solution (a centralized coordinator).

A possible improvement is to take into consideration these constraints to generate an optimal runtime architecture. It allows choosing which architecture best suits the geographical, computational, and network constraints. Additionally, the network overload should be taken into account to measure a more precise performance gain or loss over the local deployment.

The allocation of resources and computational power should be also considered. In a large system, an analysis on strongly connected simulation units must be performed to allocate a single host or closed hosts to reduce the communication over the network. This should be the case for physical systems where their interactions strongly depend on each other.

Productivity The design and development of a coordination model is a challenging task. Different levels of abstractions, different semantics, different tools, and languages can represent an obstacle to creating a correct coordination model. For this reason, we abstract away any tool or language dependency providing an agnostic DSL tailored for coordination modeling. In particular, our contribution eases the design of heterogeneous and semantics aware coordination models by giving the designer access to semantics-specific elements of each simulation unit. These elements are then used to build semantics-aware connectors that create semantics bridges across the simulation units. Connectors are then used to generate a runnable co-simulation to verify and validate the overall system. The emerging behavior can be analyzed using the tools provided by the Cosim20 framework, such as plotters, or by external tools that exploit the communication layer. Furthermore, compared with other solutions that proposed a library for a specific language, we ease the apprentice by giving an easy-to-learn language specifically designed to support coordination modeling.

Runtime Coordination Algorithm We did not consider the initialization problem. It can be studied as a separate problem, so the out-coming results can be integrated into our proposition without losing their validity. Likewise, it completes our proposition in order to have a functional solution ready to be used in complex co-simulations. Furthermore, we did not exploit all the properties of the simulator exposed in the interface. As the main improvement, the generation should take them into account and generate a better performing runtime algorithm.

Cosim20 Integrated Modeling Environment & Runtime Move to a Pub-Sub network with a proxy: dynamic discovery is needed for large co-simulation. Continuing to connect each subscriber to each publisher, the cost of avoiding dynamic discovery gets higher. A simple solution is to add an intermediary proxy: a static point in the network to which all nodes connect. Usually, this is the role of a message broker, but ∅MQ does not come with a message broker. The proxy opens an XSUB socket, an XPUB socket, and binds each to well-known IP addresses and ports. Then, all other processes connect to the proxy, instead of to each other. It becomes trivial to add more subscribers or publishers. We plan also to add features in the IDE to represent the actual network and help in the deployment of the different simulation units on the different nodes (according to the expected number of communications between nodes).

Coordination pattern between languages The major drawback of our approach is that the instantiation and binding of connector types are done manually by the system designer. With the increasing number and heterogeneity of the components, this task can quickly become difficult and error-prone. For instance, Coordination Frameworks approaches [14, 187–189] identified that the instantiation and binding of connector types can be a systematic activity the system designer repeats many times and can consequently be defined as a pattern. Such a pattern is based on the know-how of the system designer and sometimes on naming or organizational conventions adopted by the models. Thus, they have captured the specification of such a behavioral coordination pattern into a tool to automate the instantiation and binding of connector types. They specify the coordination between heterogeneous languages instead of specifying it between particular models. Such specification is then applied to a set of models to automatically instantiate a set of connector types and bind their instances. We plan to focus on how a particular coordination pattern is captured by specifying the coordination between a set of heterogeneous languages using a coordination framework. Finally, we plan to propose the definition of coordination pattern between language behavioral interface to enable the automatic generation of coordination model based on “good practice” (inspired by [25]).

Appendix

Appendix A

A.1 Detailed MCILang language class-diagram

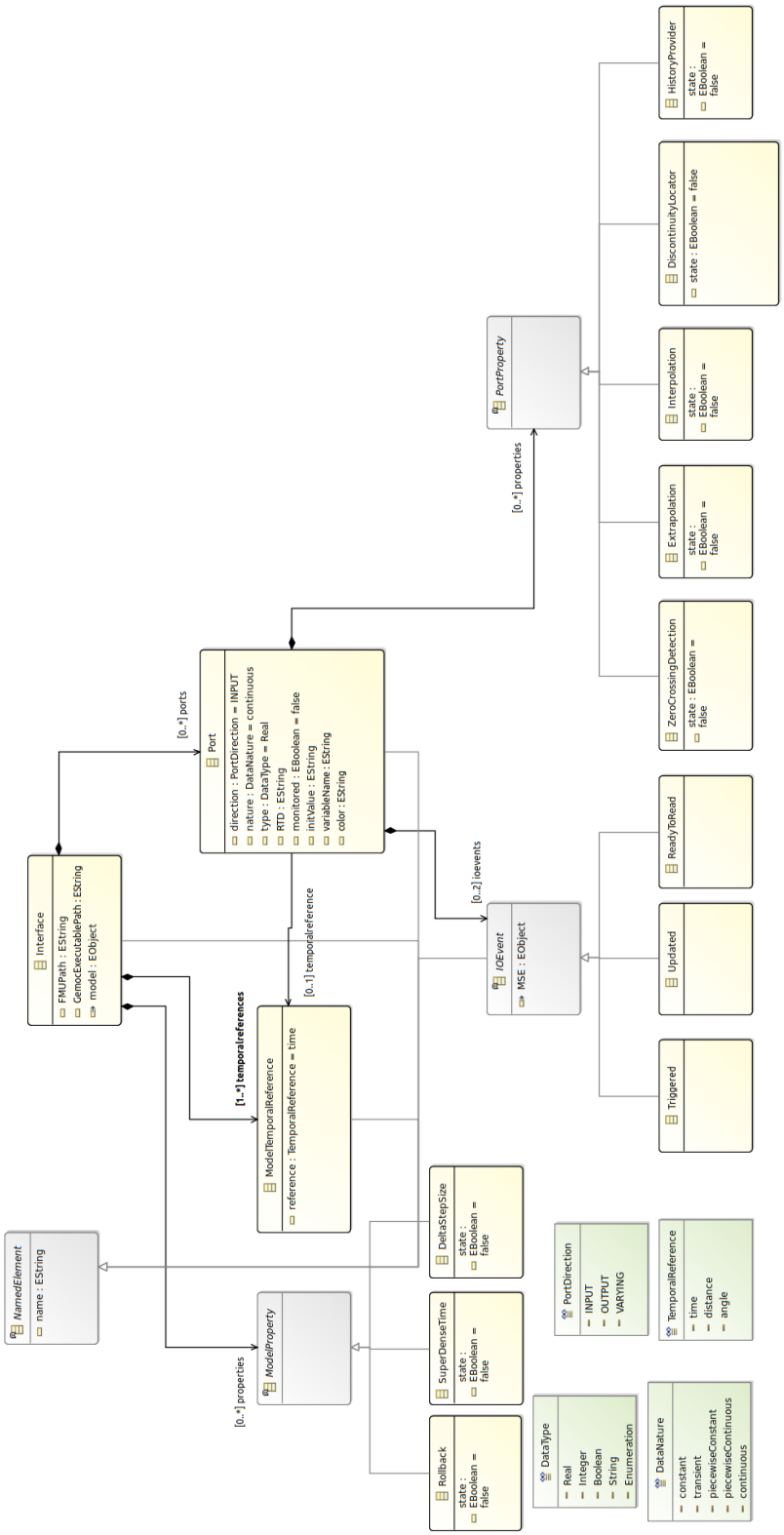


Figure A.1: ECore class diagram for the MCILang language.

A.2 Java class for the wrapper of the Box with CPU and Fan

```
1  package CoSim20GeneratedSystem;
2
3  import fr.inria.glose.cosim20.*;
4  import fr.inria.glose.cosim20.interfaces.*;
5  import org.eclipse.gemoc.execution.commons.commands.*;
6  import org.eclipse.gemoc.execution.commons.predicates.*;
7  import java.io.IOException;
8  import fr.inria.glose.cosim20.CONFIG;
9
10 import java.io.File;
11 import java.io.FileNotFoundException;
12 import java.io.IOException;
13 import java.io.PrintWriter;
14 import java.math.BigDecimal;
15 import java.util.Arrays;
16 import java.util.ArrayList;
17 import java.util.concurrent.TimeUnit;
18
19 public class BoxBI extends CoordinationInterface {
20   // FMI Standard specific variables
21   double startTime = 0.0;
22   double stopTime = CONFIG.EndOfSimulation.doubleValue();
23   String fmuPath =
       "src/test/resources/CPUinBoxWithFanHeatModel.fmu";
24
25   public Port isStopped = new FollowerPort(
26       "CPUprotection::switchCPUState",
27       "isStopped",
28       "CTRL",
29       false);
30   public Port CPUTemperatureForController = new FollowerPort(
31       "CPUTemperatureForController",
32       "CPUTemperature",
33       "CTRL",
34       0.0);
35   public Port CPUTemperatureForPID = new InitiatorPort(
36       "CPUTemperatureForPID",
37       "CPUTemperature",
38       "CTRL",
39       0.0);
40   public Port BoxTemperature = new FollowerPort(
41       "BoxTemperature",
42       "BoxTemperature",
43       "TestBox",
44       25);
45   public FollowerDeterministicPort CPUfanSpeed = new
       FollowerDeterministicPort(
46       "CPUfanSpeed",
47       "CPUfanSpeed",
48       "PID",
49       0,
50       new TemporalPredicate(5));
51
52   public BoxBI() {
53     super("Box", "localhost", 26001, 26101, 2, 2);
54
55     // Setup the FMI model
56     model = new FMIInterface(fmuPath, startTime, stopTime);
57
58     // Connector between Box CPUTemperature -> PIDController
       using dt = 5 seconds
59     initiatorPorts.add(CPUTemperatureForPID);
```

```
60      todo.add(new PeriodicAction(Action.TypeOfAction.PUBLISH,
        CPUTemperatureForPID, "tcp://localhost:26001", now, new
        TemporalPredicate(5)));
61      // Connector between Controller -> Box using Updated
        condition
62      followerPorts.add(isStopped);
63      // Connector between Test -> Box using dt (fixed in this
        example)
64      followerPorts.add(BoxTemperature);
65      // Connector between Box -> Controller using R2R condition
        on Controller
66      followerPorts.add(CPUTemperatureForController);
67      // Connector between PID and Box using dt = 5
68      followerDeterministicPorts.add(CPUfanSpeed);
69
70      portMap.put("CPUprotection::switchCPUState", isStopped);
71      portMap.put("CPUprotection::cpuTemperature",
        CPUTemperatureForController);
72      portMap.put("CPUTemperature", CPUTemperatureForPID);
73      portMap.put("BoxTemperature", BoxTemperature);
74      portMap.put("CPUfanSpeed", CPUfanSpeed);
75
76      // Set the input ports
77      addNewInputPort("CPUprotection::switchCPUState",
        "tcp://localhost:26000", "tcp://localhost:26100");
78      addNewInputPort("CPUprotection::cpuTemperature",
        "tcp://localhost:26000", "tcp://localhost:26100");
79      addNewInputPort("CPUfanSpeed", "tcp://localhost:26003",
        "tcp://localhost:26103");
80
81      model.set(BoxTemperature.associatedModelVariableName, 25);
82      model.set(CPUfanSpeed.associatedModelVariableName, 0);
83      model.set(isStopped.associatedModelVariableName, false);
84    }
85    @Override
86    public void onTime(Action currentAction, StopCondition sr) {
87      if (now.compareTo(currentAction.temporalHorizon) == 0) {
88        // Execute the corresponding action
89        if (currentAction.port.compareTo(CPUfanSpeed) == 0) {
90            model.set(CPUfanSpeed.associatedModelVariableName,
        (Double) currentAction.getValue());
91        } else if (currentAction.port.compareTo(isStopped) == 0) {
92            model.set(isStopped.associatedModelVariableName,
        (Boolean) currentAction.getValue());
93        } else if (currentAction.port.compareTo(BoxTemperature) ==
        0) {
94            model.set(BoxTemperature.associatedModelVariableName,
        (Double) currentAction.getValue());
95        } else if
        (currentAction.port.compareTo(CPUTemperatureForController)
        == 0) {
96            double temperature =
97            (double) model.get(
98
        CPUTemperatureForController.associatedModelVariableName);
99            publish(CPUTemperatureForController, temperature, now);
100       } else if
        (currentAction.port.compareTo(CPUTemperatureForPID) == 0) {
101           double temperature =
102           (double) model.get(
103
        CPUTemperatureForController.associatedModelVariableName);
104           publish(CPUTemperatureForPID, (double) temperature,
        now);
105       }
106
107       currentAction.setDone();
```

```
108       if (currentAction instanceof PeriodicAction) {
109         todo.add(new PeriodicAction(currentAction.typeOfAction,
          currentAction.port, currentAction.hostSource, now,
          ((PeriodicAction) currentAction).getTemporalPredicate()));
110       }
111     }
112   }
113
114   @Override
115   public void onEvent(Action currentAction, StopCondition sr) {}
116
117   @Override
118   public void onReadyToRead(Action currentAction, StopCondition
        sr) {}
119
120   @Override
121   public void onUpdated(Action currentAction, StopCondition sr)
        {}
122
123   @Override
124   public void onSync(Action currentAction) {
125     onTime(currentAction, null);
126   }
127
128   @Override
129   public void onEnd() {}
130
131   @Override
132   public CoordinationPredicate setInitiatorsPredicate() {
133     return null;
134   }
135 }
```

A.3 Java class for the wrapper of the Heat controller

```
1  package CoSim20GeneratedSystem;
2
3  import fr.inria.glose.cosim20.*;
4  import fr.inria.glose.cosim20.interfaces.*;
5  import org.eclipse.gemoc.execution.commons.commands.*;
6  import org.eclipse.gemoc.execution.commons.predicates.*;
7  import java.io.IOException;
8  import fr.inria.glose.cosim20.CONFIG;
9
10 import java.io.File;
11 import java.io.FileNotFoundException;
12 import java.io.IOException;
13 import java.io.PrintWriter;
14 import java.math.BigDecimal;
15 import java.util.Arrays;
16 import java.util.ArrayList;
17 import java.util.concurrent.TimeUnit;
18
19 public class OverHeatController extends BehavioralInterface {
20   public Port SwitchCPUState_ctrl2box = new InitiatorPort(
21     "SwitchCPUState_ctrl2box",
22     "CPUprotection::switchCPUState",
23     "CPUinBox",
24     false);
25   public Port CPUTemperature_box2ctrl = new InitiatorPort(
26       "CPUTemperature_box2ctrl",
27       "CPUprotection::cpuTemperature",
28       "CPUinBox",
```

```
29        25);
30
31    public OverHeatController() {
32      super(
33          "overHeatController",
34          "localhost",
35          38237,
36          36751,
37          0,
38          2);
39      // -------------------------------------------------------
40      // Initialize the model
41      // -------------------------------------------------------
42      this.model = new GemocInterface(
43          "src/test/resources/overHeatControler.jar",
44          "localhost",
45          39635);
46      initiatorPorts = new ArrayList< >();
47      followerPorts = new ArrayList< >();
48      followerDeterministicPorts = new ArrayList< >();
49
50      initiatorPorts.add(SwitchCPUState_ctrl2box);
51      initiatorPorts.add(CPUTemperature_box2ctrl);
52
53      portMap.put("isStopped_ctrl2box", SwitchCPUState_ctrl2box);
54      portMap.put("SwitchCPUState_ctrl2box",
        SwitchCPUState_ctrl2box);
55      portMap.put("CPUprotection::switchCPUState",
        SwitchCPUState_ctrl2box);
56      portMap.put("CPUTemperature_box2ctrl",
        CPUTemperature_box2ctrl);
57      portMap.put("CPUprotection::cpuTemperature",
        CPUTemperature_box2ctrl);
58
59      addNewInputPort(
60          "CPUTemperature_box2ctrl",
61          "tcp://localhost:41789",
62          "tcp://localhost:41989");
63
64      model.set("CPUprotection::cpuTemperature::currentValue", 25);
65    }
66
67    @Override
68    public void onTime(Action currentAction, StopCondition sr) {
69      if (now.compareTo(currentAction.temporalHorizon) == 0) {
70        // Execute the corresponding action
71        if
        (currentAction.port.ID.compareTo(SwitchCPUState_ctrl2box.ID)
        == 0) {
72          boolean value = (boolean)
        SwitchCPUState_ctrl2box.getValue();
73          publish(SwitchCPUState_ctrl2box, value, now);
74        }
75        // Execute the corresponding action
76        if
        (currentAction.port.ID.compareTo(CPUTemperature_box2ctrl.ID)
        == 0) {
77          BigDecimal temp = new
        BigDecimal(currentAction.getValue().toString());
78          int value = temp.intValue();
79          model.set("CPUprotection::cpuTemperature::currentValue",
        value);
80        }
81        // Remove the current action from the to-do list
82        currentAction.setDone();
83        if (currentAction instanceof PeriodicAction) {
84          todo.add(
```

```
85          new PeriodicAction(
86          currentAction.typeOfAction, currentAction.port,
            currentAction.hostSource, now, ((PeriodicAction)
            currentAction).getTemporalPredicate()));
87        }
88      }
89    }
90
91    @Override
92    public void onEvent(Action currentAction, StopCondition sc) {
93      SwitchCPUState_ctrl2box.setValue(!((boolean)
        SwitchCPUState_ctrl2box.getValue()));
94      boolean value = (boolean) SwitchCPUState_ctrl2box.getValue();
95      publish(SwitchCPUState_ctrl2box, value,
        Utils.toBigDecimal(sc.timeValue));
96    }
97
98    @Override
99    public void onReadyToRead(Action currentAction, StopCondition
        sc) {
100     retrieve(portMap.get(sc.objectQualifiedName), now);
101     currentAction.setDone();
102   }
103
104   @Override
105   public void onUpdated(Action currentAction, StopCondition sc)
        {}
106
107   @Override
108   public void onSync(Action currentAction) {
109     onTime(currentAction, null);
110   }
111
112   @Override
113   public CoordinationPredicate setInitiatorsPredicate() {
114     EventPredicate SwitchCPUState_ctrl2box_predicate = new
        EventPredicate("occurs", "CPUprotection::switchCPUState");
115     ReadyToReadPredicate CPUTemperature_box2ctrl_predicate = new
        ReadyToReadPredicate("currentValue",
        "CPUprotection::cpuTemperature");
116     BinaryPredicate binaryPredicate0 = new BinaryPredicate(
117     SwitchCPUState_ctrl2box_predicate,
        CPUTemperature_box2ctrl_predicate,
        BinaryPredicate.BooleanBinaryOperator.OR);
118     return binaryPredicate0;
119   }
120
121   @Override
122   public void onEnd() {}
123 }
```

Bibliography

References in citation order.

[1] Edward A Lee. 'Cyber physical systems: Design challenges'. In: *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*. IEEE. 2008, pp. 363–369 (cited on page 1).

[2] Stefan Klikovits, Rima Al-Ali, Moussa Amrani, Ankica Barisic, Fernando Barros, Dominique Blouin, Etienne Borde, Didier Buchs, Holger Giese, Miguel Goulão, Mauro Iacono, Florin Leon, Eva Navarro, Patrizio Pelliccione, and Ken Vanherpen. *State-of-the-art on Current Formalisms used in Cyber-Physical Systems Development*. Jan. 2019. DOI: 10.5281/zenodo.2533455 (cited on page 1).

[3] Harinder Jagdev and Jim Browne. 'The Extended Enterprise - A Context for Manufacturing'. In: *Production Planning & Control - PRODUCTION PLANNING CONTROL* 9 (Apr. 1998), pp. 216–229. DOI: 10.1080/095372898234190 (cited on pages 2, 11).

[4] Modelisar. *FMI for Model Exchange and Co-Simulation*. July 2014. URL: https://fmi-standard.org/downloads%5C#version2 (cited on pages 2, 3, 16, 20, 22, 24, 33, 36, 40, 42–45, 48, 49, 52, 53, 57, 60, 63, 70, 81).

[5] Judith S Dahmann. 'High level architecture for simulation'. In: *Proceedings First International Workshop on Distributed Interactive Simulation and Real Time Applications*. IEEE. 1997, pp. 9–14 (cited on page 2).

[6] Jens Bastian, Christoph Clauß, Susann Wolf, and Peter Schneider. 'Master for Co-Simulation Using FMI'. In: *8th International Modelica Conference*. 2011 (cited on pages 2, 23, 58, 59).

[7] Tom Schierz, Martin Arnold, and Christoph Clauß. 'Co-simulation with communication step size control in an FMI compatible master algorithm'. In: *Proceedings of the 9th International MODELICA Conference; Munich; Germany*. 076. Linköping University Electronic Press. 2012, pp. 205–214 (cited on pages 2, 23, 58).

[8] David Broman, Lev Greenberg, Edward A. Lee, Michael Masin, Stavros Tripakis, and Michael Wetter. 'Requirements for Hybrid Cosimulation Standards'. In: *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*. HSCC '15. Seattle, Washington: Association for Computing Machinery, 2015, pp. 179–188. DOI: 10.1145/2728606.2728629 (cited on pages 2, 3, 33, 45, 64, 81).

[9] David Broman, Christopher Brooks, Lev Greenberg, Edward A Lee, Michael Masin, Stavros Tripakis, and Michael Wetter. 'Determinate composition of FMUs for co-simulation'. In: *Proceedings of the Eleventh ACM International Conference on Embedded Software*. IEEE Press. 2013, p. 2 (cited on pages 2, 15, 22, 23, 45, 52, 58).

[10] Julien Deantoni, Cédric Brun, Benoît Caillaud, Robert France, Gabor Karsai, Oscar Nierstrasz, and Eugene Syriani. 'Domain Globalization: Using Languages to Support Technical and Social Coordination'. In: *Globalizing Domain-Specific Languages*. Ed. by Combemale, Benoit, Cheng, Betty H.C., France, Robert B., Jézéquel, Jean-Marc, Rumpe, and Bernhard. Vol. 9400. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 70–87. DOI: 10.1007/978-3-319-26172-0_5 (cited on page 2).

[11] Giovanni Liboni, Julien Deantoni, Antonio Portaluri, Davide Quaglia, and Robert De Simone. 'Beyond Time-Triggered Co-simulation of Cyber-Physical Systems for Performance and Accuracy Improvements'. In: *10th Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*. Manchester, United Kingdom, Jan. 2018 (cited on pages 2, 3, 5, 17, 24, 31, 32, 57, 58, 61, 62, 65, 81, 104).

[12] Sadaf Mustafiz, Cláudio Gomes, Hans Vangheluwe, and Bruno Barroca. 'Modular design of hybrid languages by explicit modeling of semantic adaptation'. In: *2016 Symposium on Theory of Modeling and Simulation (TMS-DEVS)*. Apr. 2016, pp. 1–8. DOI: 10.23919/TMS.2016.7918835 (cited on pages 2, 20, 30, 59, 61).

[13] Cláudio Gomes, Casper Thule, David Broman, Peter Gorm Larsen, and Hans Vangheluwe. 'Co-simulation: a Survey'. In: *ACM Computing Surveys* 51.3 (2018), Article 49. DOI: 10.1145/3179993 (cited on page 2).

[14] Johan Eker, Jorn W Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Sonia Sachs, Yuhong Xiong, and Stephen Neuendorffer. 'Taming heterogeneity - the Ptolemy approach'. In: *Proceedings of the IEEE* 91.1 (2003), pp. 127–144 (cited on pages 3, 15, 20, 33, 50, 116).

[15] Bert Van Acker, Joachim Denil, Hans Vangheluwe, and Paul De Meulenaere. 'Generation of an Optimised Master Algorithm for FMI Co-simulation'. In: *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*. DEVS '15. Alexandria, Virginia: Society for Computer Simulation International, 2015 (cited on pages 3, 22, 23, 58).

[16] Stavros Tripakis, David Broman, and Computer Sciences. *Bridging the Semantic Gap Between Heterogeneous Modeling Formalisms and FMI*. Tech. rep. 2014 (cited on pages 3, 81).

[17] Casper Thule, Cláudio Gomes, Julien Deantoni, Peter Gorm Larsen, Jörg Brauer, and Hans Vangheluwe. 'Towards the Verification of Hybrid Co-simulation Algorithms'. In: *Workshop on Formal Co-Simulation of Cyber-Physical Systems (SEFM satellite)*. Toulouse, France, June 2018 (cited on pages 3, 57, 58).

[18] Jean-Philippe Tavella, Mathieu Caujolle, Stephane Vialle, Cherifa Dad, Charles Tan, Gilles Plessis, Mathieu Schumann, Arnaud Cuccuru, and Sebastien Revol. 'Toward an accurate and fast hybrid multi-simulation with the FMI-CS standard'. In: *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2016, pp. 1–5. DOI: 10.1109/ETFA.2016.7733616 (cited on pages 3, 32, 44, 48, 49, 57, 58).

[19] Jean-Philippe Tavella, Mathieu Caujolle, Charles Tan, Gilles Plessis, Mathieu Schumann, Stephane Vialle, Cherifa Dad, Arnaud Cuccuru, and Sebastien Revol. 'Toward an Hybrid Co-simulation with the FMI-CS Standard'. In: 2016 (cited on pages 3, 24, 81).

[20] David Garlan and Mary Shaw. 'Introduction to software architecture'. In: *Advanced Topics in Science and Technology in China* January (1994), pp. 1–33. DOI: 10.1007/978-3-540-74343-9_1 (cited on page 4).

[21] Nenad Medvidovic and Richard N Taylor. 'A framework for classifying and comparing architecture description languages'. In: *ACM SIGSOFT Software Engineering Notes* 22.6 (1997), pp. 60–76 (cited on pages 4, 35, 36, 58).

[22] George A Papadopoulos and Farhad Arbab. 'Coordination models and languages'. In: *Advances in computers* 46 (1998), pp. 329–400 (cited on pages 4, 58, 59).

[23] Edward A Lee and Alberto Sangiovanni-Vincentelli. 'A framework for comparing models of computation'. In: *IEEE Transactions on computer-aided design of integrated circuits and systems* 17.12 (1998), pp. 1217–1229 (cited on pages 4, 60).

[24] Cécile Hardebolle and Frédéric Boulanger. 'Modhel'x: A component-oriented approach to multi-formalism modeling'. In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2007, pp. 247–258 (cited on pages 4, 60).

[25] Matias Ezequiel Vara Larsen, Julien Deantoni, Benoit Combemale, and Frédéric Mallet. 'A Behavioral Coordination Operator Language (BCOoL)'. In: *International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Ed. by Timothy Lethbridge, Jordi Cabot, and Alexander Egyed. 18. to be published in the proceedings of the Models 2015 conference. Ottawa, Canada: ACM, Sept. 2015, p. 462 (cited on pages 4, 35, 50, 60, 116).

[26] Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale. 'Execution framework of the gemoc studio (tool demo)'. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. ACM. 2016, pp. 84–89 (cited on pages 4, 60, 84).

[27] Benoit Combemale, Julien Deantoni, Benoit Baudry, Robert B France, Jean-Marc Jézéquel, and Jeff Gray. 'Globalizing Modeling Languages'. In: *Computer* 47.6 (June 2014), pp. 68–71. DOI: 10.1109/MC.2014.147 (cited on pages 5, 11, 61).

[28] Fabio Cremona, Marten Lohstroh, David Broman, Edward A. Lee, Michael Masin, and Stavros Tripakis. 'Hybrid co-simulation: it's about time'. In: *Software & Systems Modeling* 18.3 (2019), pp. 1655–1679. DOI: 10.1007/s10270-017-0633-6 (cited on pages 6, 24, 33).

[29] International Council on Systems Engineering, ed. *INCOSE Systems Engineering Handbook*. Vol. 2.0. 2000 (cited on page 9).

[30] Joel Moses. 'Flexibility and Its Relation to Complexity and Architecture'. In: *Complex Systems Design & Management*. Ed. by Marc Aiguier, Francis Bretaudeau, and Daniel Krob. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 197–206 (cited on page 9).

[31] Julio M Ottino. 'Engineering complex systems'. In: *Nature* 427.6973 (2004), pp. 399–399. DOI: 10.1038/427399a (cited on page 10).

[32] Nicolai Pedersen., Kenneth Lausdahl., Enrique Vidal Sanchez., Peter Gorm Larsen., and Jan Madsen. 'Distributed Co-Simulation of Embedded Control Software with Exhaust Gas Recirculation Water Handling System using INTO-CPS'. In: *Proceedings of the 7th International Conference on Simulation and Modeling Methodologies, Technologies and Applications - Volume 1: SIMULTECH*, INSTICC. SciTePress, 2017, pp. 73–82. DOI: 10.5220/0006412700730082 (cited on page 10).

[33] Cláudio Gomes, Hans Vangheluwe, and Paul De Meulenaere. 'Property preservation in co-simulation'. PhD thesis. Ph. D. thesis, University of Antwerp, 2019 (cited on pages 10, 15).

[34] Getachew F. Belete, Alexey Voinov, and Gerard F. Laniak. 'An overview of the model integration process: From pre-integration assessment to testing'. In: *Environmental Modelling Software* 87 (2017), pp. 49–63. DOI: https://doi.org/10.1016/j.envsoft.2016.10.013 (cited on page 10).

[35] Jonathan L. Goodall, Bella F. Robinson, and Anthony M. Castronova. 'Modeling water resource systems using a service-oriented computing paradigm'. In: *Environmental Modelling Software* 26.5 (2011), pp. 573–582. DOI: https://doi.org/10.1016/j.envsoft.2010.11.013 (cited on page 10).

[36] Bernd Neumayr, Michael Schrefl, and Bernhard Thalheim. 'Modeling Techniques for Multi-level Abstraction'. In: Jan. 2008, pp. 68–92. DOI: 10.1007/978-3-642-17505-3_4 (cited on page 10).

[37] Heejung Chang and Kangsun Lee. 'Applying Web Services and Design Patterns to Modeling and Simulating Real-World Systems'. In: *Artificial Intelligence and Simulation*. Ed. by Tag Gon Kim. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 351–359 (cited on page 10).

[38] Vadim Lisitsa, Vladimir Cheverda, and V. Volianskaia. 'Numerical Simulation of Geological Faults by Discrete Elements Method'. In: June 2019. DOI: 10.3997/2214-4609.201901680 (cited on page 10).

[39] John A. Stankovic. 'Misconceptions about real-time computing: a serious problem for next-generation systems'. In: *Computer* 21.10 (1988), pp. 10–19. DOI: 10.1109/2.7053 (cited on page 11).

[40] Gabor Karsai, Janos Sztipanovits, Akos Ledeczi, and Ted Bapty. 'Model-integrated development of embedded software'. In: *Proceedings of the IEEE* 91.1 (2003), pp. 145–164 (cited on page 11).

[41] Severin Sadjina, Lars Tandle Kyllingstad, Martin Rindarøy, Stian Skjong, Vilmar Æsøy, and Eilif Pedersen. 'Distributed Co-simulation of Maritime Systems and Operations'. In: *Journal of Offshore Mechanics and Arctic Engineering* 141.1 (Sept. 2018). 011302. DOI: 10.1115/1.4040473 (cited on pages 11, 52).

[42] Robert M Argent. 'An overview of model integration for environmental applications–components, frameworks and semantics'. In: *Environmental Modelling Software* 19.3 (2004). Concepts, Methods and Applications in Environmental Model Integration, pp. 219–234. DOI: https://doi.org/10.1016/S1364-8152(03)00150-6 (cited on page 11).

[43] W Ross Ashby. *An introduction to cybernetics*. Chapman & Hall Ltd, 1961 (cited on page 12).

[44] BOSE Debayan. 'Component Based Development-Application In Software Engineering'. In: *Indian Statistical Institute* (2011) (cited on page 12).

[45] Jean Bézivin and Olivier Gerbé. 'Towards a precise definition of the OMG/MDA framework'. In: Dec. 2001, pp. 273–280. DOI: 10.1109/ASE.2001.989813 (cited on page 12).

[46] Jean Bézivin. 'From Object Composition to Model Transformation with the MDA.' In: Jan. 2001, pp. 350–354. DOI: 10.1109/TOOLS.2001.10021 (cited on page 12).

[47] Jean Bézivin. 'Model Driven Engineering: An Emerging Technical Space'. In: vol. 4143. Jan. 2005, pp. 36–64. DOI: 10.1007/11877028_2 (cited on page 12).

[48] Vicente García Díaz, Edward Núñez Valdez, Jordán Espada, B. Pelayo García-Bustelo, Juan Cueva Lovelle, and Carlos Marín. 'A brief introduction to model-driven engineering'. In: 18 (Apr. 2014), pp. 127–142 (cited on pages 12, 13).

[49] Alberto Rodrigues da Silva. 'Model-driven engineering: A survey supported by the unified conceptual model'. In: *Computer Languages, Systems & Structures* 43 (2015), pp. 139–155. DOI: https://doi.org/10.1016/j.cl.2015.06.001 (cited on page 12).

[50] Object Management Group. *Object Management Group*. http://www.omg.org/. Nov. 2020 (cited on page 12).

[51] Denis Merigoux, Raphaël Monat, and Jonathan Protzenko. *A Modern Compiler for the French Tax Code*. 2020 (cited on page 12).

[52] Douglas C. Schmidt. 'Guest Editor's Introduction: Model-Driven Engineering'. In: *Computer* 39.2 (2006), pp. 25–31. DOI: 10.1109/MC.2006.58 (cited on page 13).

[53] Gordon D Plotkin. 'A structural approach to operational semantics'. In: (1981) (cited on page 13).

[54] Robert W Floyd. 'Assigning meanings to programs'. In: *Program Verification*. Springer, 1993, pp. 65–81 (cited on page 13).

[55] Robert D. Tennent. 'The Denotational Semantics of Programming Languages'. In: *Commun. ACM* 19.8 (Aug. 1976), pp. 437–453. DOI: 10.1145/360303.360308 (cited on page 13).

[56] Frank Budinsky, Raymond Ellersick, David Steinberg, Timothy J Grose, and Ed Merks. *Eclipse modeling framework: a developer's guide*. Addison-Wesley Professional, 2004 (cited on page 13).

[57] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. 'ATL: A model transformation tool'. In: *Science of computer programming* 72.1-2 (2008), pp. 31–39 (cited on page 14).

[58] Krzysztof Czarnecki and Simon Helsen. 'Classification of model transformation approaches'. In: *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*. Vol. 45. 3. USA. 2003, pp. 1–17 (cited on page 14).

[59] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008 (cited on pages 14, 50).

[60] Zoé Drey, Cyril Faucher, Franck Fleurey, Vincent Mahé, and Didier Vojtisek. 'Kermeta language'. In: *Reference Manual* (2009) (cited on page 15).

[61] Jeff C Jensen, Danica H Chang, and Edward A Lee. 'A model-based design methodology for cyber-physical systems'. In: *2011 7th International Wireless Communications and Mobile Computing Conference*. 2011, pp. 1666–1671. DOI: 10.1109/IWCMC.2011.5982785 (cited on page 15).

[62] Gabor Karsai, Andras Lang, and Sandeep Neema. 'Design patterns for open tool integration'. In: *Software & Systems Modeling* 4.2 (2005), pp. 157–170 (cited on page 15).

[63] Andreas Himmler. 'Hardware-in-the-Loop Technology Enabling Flexible Testing Processes'. In: Jan. 2013. DOI: 10.2514/6.2013-816 (cited on page 15).

[64] Ming-chin Wu and Ming-chang Shih. 'Simulated and experimental study of hydraulic anti-lock braking system using sliding-mode PWM control'. In: *Mechatronics* 13.4 (2003), pp. 331–351. DOI: https://doi.org/10.1016/S0957-4158(01)00049-6 (cited on page 15).

[65] Enrico Fraccaroli, Michele Lora, Sara Vinco, Davide Quaglia, and Franco Fummi. 'Integration of mixed-signal components into virtual platforms for holistic simulation of smart systems'. In: *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2016, pp. 1586–1591 (cited on page 15).

[66] ITEA3 project. *Modelisar*. https://itea3.org/project/modelisar.html (cited on page 16).

[67] Torsten Blochwitz, Martin Otter, Martin Arnold, Constanze Bausch, Christoph Clauß, Hilding Elmqvist, Andreas Junghanns, Jakob Mauss, Manuel Monteiro, Thomas Neidhold, Dietmar Neumerkel, Hans Olsson, Jörg-Volker Peetz, and Susann Wolf. 'The Functional Mockup Interface for Tool independent Exchange of Simulation Models'. In: Mar. 2011, pp. 105–114. DOI: 10.3384/ecp11063105 (cited on pages 16, 42).

[68] T. Blochwitz, Martin Otter, Johan Åkesson, Martin Arnold, C. Clauß, Hilding Elmqvist, Markus Friedrich, Andreas Junghanns, Jakob Mauss, Dietmar Neumerkel, Hans Olsson, and Antoine Viel. 'Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models'. In: Sept. 2012. DOI: 10.3384/ecp12076173 (cited on pages 16, 42).

[69] Judith S. Dahmann, Richard M. Fujimoto, and Richard M. Weatherly. 'The Department of Defense High Level Architecture'. In: *Proceedings of the 29th Conference on Winter Simulation*. WSC '97. Atlanta, Georgia, USA: IEEE Computer Society, 1997, pp. 142–149. DOI: 10.1145/268437.268465 (cited on page 16).

[70] Judith S Dahmann, Richard M Fujimoto, and Richard M Weatherly. 'The DoD high level architecture: an update'. In: *1998 Winter Simulation Conference. Proceedings (Cat. No. 98CH36274)*. Vol. 1. IEEE. 1998, pp. 797–804 (cited on page 16).

[71] IEEE Standards Association et al. 'IEEE Standard for Modeling and Simulation (M&S) high level architecture (HLA)–framework and rules'. In: *Institute of Electrical and Electronics Engineers, New York. IEEE Standard* 1516-2010 (2010), pp. 10–1109 (cited on pages 16, 29, 45, 54).

[72] Stefano Centomo, Julien Deantoni, and Robert De Simone. 'Using SystemC Cyber Models in an FMI Co-Simulation Environment'. In: *19th Euromicro Conference on Digital System Design 31 August - 2 September 2016*. Vol. 19. 19th Euromicro Conference on Digital System Design. Limassol, Cyprus, Aug. 2016. DOI: 10.1109/DSD.2016.86 (cited on pages 17, 32, 58).

[73] Virginie Galtier, Michel Ianotto, Mathieu Caujolle, Rémi Corniglion, jean-philippe Tavella, Jose Evora-Gomez, José Juan, Hernández Cabrera, Vincent Reinbold, and Enrique Kremers. 'Experimenting with Matryoshka Co-Simulation: Building Parallel and Hierarchical FMUs'. In: May 2017 (cited on pages 17, 31, 32, 53).

[74] Farhad Arbab, Ivan Herman, and Pål Spilling. 'An overview of Manifold and its implementation'. In: *Concurrency: practice and experience* 5.1 (1993), pp. 23–70 (cited on pages 20, 38).

[75] Sudhir Ahuja, N Curriero, and David Gelernter. 'Linda and Friends'. In: *Computer* 19.8 (1986), pp. 26–34. DOI: 10.1109/MC.1986.1663305 (cited on pages 20, 37, 39).

[76] Ananda Basu, Bensalem Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. 'Rigorous component-based system design using the BIP framework'. In: *IEEE Software* 28.3 (2011). cited By 164, pp. 41–48. DOI: 10.1109/MS.2011.27 (cited on pages 20, 38–40, 59).

[77] Michael Tiller. *Introduction to physical modeling with Modelica*. Springer Science & Business Media, 2001 (cited on page 20).

[78] Edward A. Lee, Stephen Neuendorffer, and Gang Zhou. 'Continuous-Time Models'. In: *System Design, Modeling, and Simulation using Ptolemy II*. Ed. by Claudius Ptolemaeus. Ptolemy.org, 2014 (cited on pages 20, 21).

[79] Sergio Blanes and Fernando Casas. *A concise introduction to geometric numerical integration*. CRC press, 2017 (cited on page 21).

[80] Uri M Ascher, Steven J Ruuth, and Raymond J Spiteri. 'Implicit-explicit Runge-Kutta methods for time-dependent partial differential equations'. In: *Applied Numerical Mathematics* 25.2-3 (1997), pp. 151–167 (cited on page 21).

[81] Jonathan M. Blackledget. 'Chapter 9 - Iterative Methods of Solution'. In: *Digital Signal Processing (Second Edition)*. Ed. by Jonathan M. Blackledget. Second Edition. Woodhead Publishing Series in Electronic and Optical Materials. Woodhead Publishing, 2006, pp. 237–254. DOI: https://doi.org/10.1533/9780857099457.2.237 (cited on page 22).

[82] Carl Kelley. 'Iterative Methods for Solving Linear and Nonlinear Equations'. In: *SERBIULA (sistema Librum 2.0)* 16 (Jan. 1995). DOI: 10.1137/1.9781611970944 (cited on page 22).

[83] Simon Thrane Hansen, Casper Thule, and Cláudio Gomes. 'An FMI-Based Initialization Plugin for INTO-CPS Maestro 2'. In: *Software Engineering and Formal Methods. SEFM 2020 Collocated Workshops*. Ed. by Loek Cleophas and Mieke Massink. Cham: Springer International Publishing, 2021, pp. 295–310 (cited on page 22).

[84] Fabio Cremona, Marten Lohstroh, David Broman, Marco Di Natale, Edward A. Lee, and Stavros Tripakis. 'Step Revision in Hybrid Co-Simulation with FMI'. In: *Proceedings of the 14th ACM-IEEE International Conference on Formal Methods and Models for System Design*. MEMOCODE '16. Kanpur, India: IEEE Press, 2016, pp. 173–183 (cited on pages 22, 32, 44).

[85] Cláudio Gomes, Casper Thule, P. Larsen, J. Denil, and H. Vangheluwe. 'Co-simulation of Continuous Systems: A Tutorial'. In: *ArXiv* abs/1809.08463 (2018) (cited on pages 22–24).

[86] David Broman, Edward A Lee, Stavros Tripakis, and Martin Törngren. 'Viewpoints, formalisms, languages, and tools for cyber-physical systems'. In: *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling*. 2012, pp. 49–54 (cited on page 23).

[87] Cláudio Gomes, Casper Thule, David Broman, Peter Larsen, and Hans Vangheluwe. 'Co-simulation: State of the art'. In: (Feb. 2017) (cited on pages 23, 67).

[88] Christian Andersson. 'Methods and tools for co-simulation of dynamic systems with the functional mock-up interface'. PhD thesis. Lund University, 2016 (cited on page 23).

[89] Christian Andersson, Claus Führer, and Johan Åkesson. 'Efficient predictor for co-simulation with multistep sub-system solvers'. In: *Technical Report in Mathematical Sciences* 2016.1 (2016) (cited on page 23).

[90] Baobing Wang and John S. Baras. 'HybridSim: A Modeling and Co-simulation Toolchain for Cyber-physical Systems'. In: *Proceedings of the 2013 IEEE/ACM 17th International Symposium on Distributed Simulation and Real Time Applications*. DS-RT '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 33–40. DOI: 10.1109/DS-RT.2013.12 (cited on pages 23, 58).

[91] Vitaly Savicks, Michael Butler, and John Colley. 'Co-simulating Event-B and continuous models via FMI'. In: *Proceedings of the 2014 Summer Simulation Multiconference*. Society for Computer Simulation International. 2014, p. 37 (cited on pages 23, 58).

[92] Himanshu Neema, Jesse Gohl, Zsolt Lattmann, Janos Sztipanovits, Gabor Karsai, Sandeep Neema, Ted Bapty, John Batteh, Hubertus Tummescheit, and Chandraseka Sureshkumar. 'Model-based integration platform for FMI co-simulation and heterogeneous simulations of cyber-physical systems'. In: *Proceedings of the 10 th International Modelica Conference; Lund; Sweden*. 096. Linköping University Electronic Press. 2014, pp. 235–245 (cited on pages 23, 46, 58).

[93] Virginie Galtier, Stephane Vialle, Dad Cherifa, jean-philippe Tavella, Jean-Philippe Lam-Yee-Mui, and Gilles Plessis. 'FMI-based distributed multi-simulation with DACCOSIM'. In: *Simulation Series* 47 (Apr. 2015) (cited on page 24).

[94] Pieter J Mosterman. *Hybrid dynamic systems: Modeling and execution*. 2007 (cited on pages 24, 32).

[95] Benjamin Camus, Virginie Galtier, and Mathieu Caujolle. 'Hybrid Co-simulation of FMUs using DEV DESS in MECSYCO'. In: *2016 Symposium on Theory of Modeling and Simulation (TMS-DEVS)*. 2016, pp. 1–8. DOI: 10.23919/TMS.2016.7918814 (cited on page 24).

[96] Richard M. Fujimoto. 'Parallel Discrete Event Simulation'. In: *Commun. ACM* 33.10 (Oct. 1990), pp. 30–53. DOI: 10.1145/84537.84545 (cited on page 26).

[97] Richard M Fujimoto. 'Parallel and distributed simulation systems'. In: *Proceeding of the 2001 Winter Simulation Conference (Cat. No.01CH37304)*. Vol. 1. 2001, 147–157 vol.1. DOI: 10.1109/WSC.2001.977259 (cited on pages 26, 27).

[98] K. Mani Chandy and Jayadev Misra. 'Asynchronous Distributed Simulation via a Sequence of Parallel Computations'. In: *Commun. ACM* 24.4 (Apr. 1981), pp. 198–206. DOI: 10.1145/358598.358613 (cited on page 26).

[99] David R. Jefferson. 'Virtual Time'. In: *ACM Trans. Program. Lang. Syst.* 7.3 (July 1985), pp. 404–425. DOI: 10.1145/3916.3988 (cited on page 27).

[100] Jeffrey S Steinman, Craig A Lee, Linda F Wilson, and David M Nicol. 'Global Virtual Time and Distributed Synchronization'. In: *Proceedings of the Ninth Workshop on Parallel and Distributed Simulation*. PADS '95. Lake Placid, New York, USA: IEEE Computer Society, 1995, pp. 139–148. DOI: 10.1145/214282.214324 (cited on page 27).

[101] Bernard P. Zeigler, Alexandre Muzy, and Ernesto Kofman. 'Chapter 6 - Basic Formalisms: DEVS, DESS, DTSS'. In: *Theory of Modeling and Simulation (Third Edition)*. Ed. by Bernard P. Zeigler, Alexandre Muzy, and Ernesto Kofman. Third Edition. Academic Press, 2019, pp. 153–165. DOI: https://doi.org/10.1016/B978-0-12-813370-5.00014-6 (cited on pages 27, 33).

[102] Bernard Zeigler and Hessam Sarjoughian. 'DEVS Component-Based M&S Framework: An Introduction'. In: (Jan. 2007) (cited on page 27).

[103] Benjamin Camus, Thomas Paris, Julien Vaubourg, Yannick Presse, Christine Bourjot, Laurent Ciarletta, and Vincent Chevrier. *MECSYCO: a Multi-agent DEVS Wrapping Platform for the Co-simulation of Complex Systems*. Research Report. LORIA, UMR 7503, Université de Lorraine, CNRS, Vandoeuvre-lès-Nancy ; Inria Nancy - Grand Est (Villers-lès-Nancy, France), Sept. 2016 (cited on pages 27, 46, 47, 55).

[104] Alex Chung Hen Chow. 'Parallel DEVS: A Parallel, Hierarchical, Modular Modeling Formalism and Its Distributed Simulator'. In: *Trans. Soc. Comput. Simul. Int.* 13.2 (Dec. 1996), pp. 55–67 (cited on pages 28, 29).

[105] Jan Himmelspach and Adelinde M Uhrmacher. 'Sequential processing of PDEVS models'. In: *Proceedings of the 3rd EMSS* (2006), pp. 239–244 (cited on page 29).

[106] Julien Vaubourg, Yannick Presse, Benjamin Camus, Christine Bourjot, Laurent Ciarletta, Vincent Chevrier, Jean-Philippe Tavella, and Hugo Morais. 'Multi-agent Multi-Model Simulation of Smart Grids in the MS4SG Project'. In: *Advances in Practical Applications of Agents, Multi-Agent Systems, and Sustainability: The PAAMS Collection*. Ed. by Yves Demazeau, Keith S. Decker, Javier Bajo Perez, and Fernando de la Prieta. Cham: Springer International Publishing, 2015, pp. 240–251 (cited on pages 29, 46, 52, 55).

[107] Jan Himmelspach, Roland Ewald, Stefan Leye, and Adelinde M. Uhrmacher. 'Parallel and Distributed Simulation of Parallel DEVS Models'. In: *Proceedings of the 2007 Spring Simulation Multiconference - Volume 2*. SpringSim '07. Norfolk, Virginia: Society for Computer Simulation International, 2007, pp. 249–256 (cited on page 29).

[108] Bernard P. Zeigler, Alexandre Muzy, and Ernesto Kofman. *Theory of Modeling and Simulation: Discrete Event & Iterative System Computational Foundations*. 3rd. USA: Academic Press, Inc., 2018 (cited on page 29).

[109] K. Mani Chandy and Jayadev Misra. 'Distributed simulation: A case study in design and verification of distributed programs'. In: *IEEE Transactions on software engineering* 5 (1979), pp. 440–452 (cited on pages 29, 55).

[110] Randal E Bryant. 'Simulation on a distributed system'. In: *Proc. of the 16th Design Automation Conference*. 1979, pp. 544–552 (cited on pages 29, 55).

[111] Andras Varga. 'OMNeT++'. In: *Modeling and tools for network simulation*. Springer, 2010, pp. 35–59 (cited on pages 29, 46).

[112] Luca P Carloni, Roberto Passerone, and Alessandro Pinto. *Languages and tools for hybrid systems design*. now Publishers Inc, 2006 (cited on page 30).

[113] John Lygeros, Claire Tomlin, and Shankar Sastry. 'Controllers for reachability specifications for hybrid systems'. In: *Automatica* 35.3 (1999), pp. 349–370. DOI: https://doi.org/10.1016/S0005-1098(98)00193-9 (cited on page 30).

[114] Timothy Bourke and Marc Pouzet. 'Zelus: A Synchronous Language with ODEs'. In: *Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control* (2013), pp. 113–118. DOI: 10.1145/2461328.2461348 (cited on page 30).

[115] Stefano Centomo, Michele Lora, and Franco Fummi. 'Transaction-level Functional Mockup Units for Cyber-Physical Virtual Platforms'. In: *2018 Forum on Specification Design Languages (FDL)*. 2018, pp. 5–8. DOI: 10.23919/DATE.2018.8342095 (cited on page 30).

[116] Fabio Cremona, Marten Lohstroh, Stavros Tripakis, Christopher Brooks, and Edward A Lee. 'FIDE: An FMI Integrated Development Environment'. In: *31st Annual ACM Symposium on Applied Computing*. Pisa, Italy: ACM New York, NY, USA, 2016, pp. 1759–1766. DOI: 10.1145/2851613.2851677 (cited on pages 31, 32, 44).

[117] Bernard P. Zeigler. 'Embedding DEV and DESS in DEVS'. In: 2005 (cited on page 31).

[118] Herbert Praehofer. 'System theoretic formalisms for combined discrete-continuous system simulation'. In: *International Journal of General System* 19.3 (1991), pp. 226–240 (cited on page 31).

[119] Casper Thule, Kenneth Lausdahl, Cláudio Gomes, Gerd Meisl, and Peter Gorm Larsen. 'Maestro: The INTO-CPS co-simulation framework'. In: *Simulation Modelling Practice and Theory* 92 (2019), pp. 45–61. DOI: https://doi.org/10.1016/j.simpat.2018.12.005 (cited on pages 32, 52).

[120] Jean-Philippe Tavella, Mathieu Caujolle, Charles Tan, Gilles Plessis, Mathieu Schumann, Stéphane Vialle, Cherifa Dad, Arnaud Cuccuru, and Sébastien Revol. 'Toward an Hybrid Co-simulation with the FMI-CS Standard'. In: (Apr. 2016) (cited on pages 32, 58, 63, 64).

[121] Dehui Du, Yao Wang, Yi Ao, and Biao Chen. 'An Optimized Partial Rollback Co-simulation Approach for Heterogeneous FMUs'. In: *2019 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. 2019, pp. 273–280. DOI: 10.1109/TASE.2019.00013 (cited on page 32).

[122] David Goldberg. 'What every computer scientist should know about floating-point arithmetic'. In: *ACM Computing Surveys (CSUR)* 23.1 (1991), pp. 5–48 (cited on page 33).

[123] Michael L Overton. *Numerical computing with IEEE floating point arithmetic*. SIAM, 2001 (cited on page 33).

[124] Edward A Lee and Haiyang Zheng. 'Operational semantics of hybrid systems'. In: *International Workshop on Hybrid Systems: Computation and Control*. Springer. 2005, pp. 25–53 (cited on page 33).

[125] Oded Maler, Zohar Manna, and Amir Pnueli. 'From Timed to Hybrid Systems'. In: *LNCS* 600 (Mar. 1999). Ed. by Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel (cited on page 33).

[126] David Gelernter and Nicholas Carriero. 'Coordination languages and their significance'. In: *Communications of the ACM* 35.2 (1992), p. 96 (cited on pages 35, 38).

[127] 'IEEE Recommended Practice for Architectural Description for Software-Intensive Systems'. In: *IEEE Std 1471-2000* (2000), pp. 1–30. DOI: 10.1109/IEEESTD.2000.91944 (cited on page 35).

[128] Paul C. Clements. 'A Survey of Architecture Description Languages'. In: *Proceedings of the 8th International Workshop on Software Specification and Design*. IWSSD '96. USA: IEEE Computer Society, 1996, p. 16 (cited on page 35).

[129] Prabhat Mishra and Nikil Dutt. 'Chapter 1 - Introduction to Architecture Description Languages'. In: *Processor Description Languages*. Ed. by Prabhat Mishra and Nikil Dutt. Vol. 1. Systems on Silicon. Burlington: Morgan Kaufmann, 2008, pp. 1–12. DOI: https://doi.org/10.1016/B978-012374287-2.50004-5 (cited on page 35).

[130] David Luckham, James Vera, and Sigurd Meldal. 'Three Concepts of System Architecture'. In: (Aug. 1996) (cited on pages 36, 47).

[131] David Luckham. 'Rapide: A language and toolset for causal event modelling of distributed system architectures'. In: Nov. 2006, pp. 88–96. DOI: 10.1007/3-540-64216-1_42 (cited on pages 36, 37, 41, 48, 59).

[132] Peter Feiler, David Gluch, and John Hudak. *The Architecture Analysis & Design Language (AADL): An Introduction*. Tech. rep. CMU/SEI-2006-TN-011. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2006 (cited on page 36).

[133] Emmanuel Durand. 'Description et vérification d'architectures d'application temps réel : CLARA et les réseaux de Petri temporels'. Th. : automatique et informatique appliquées. PhD thesis. Nantes: Nantes, ECN, 1998 (cited on page 36).

[134] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. 'Abstractions for software architecture and tools to support them'. In: *Software Engineering, IEEE Transactions on* 21.4 (Apr. 1995), pp. 314–335. DOI: 10.1109/32.385970 (cited on page 37).

[135] Robert J. Allen. *A Formal Approach to Software Architecture*. Tech. rep. CMU-CS-97-144. Carnegie Mellon University, 1997 (cited on pages 37, 78).

[136] Farhad Arbab. 'Reo: A Channel-Based Coordination Model for Component Composition'. In: *Mathematical. Structures in Comp. Sci.* 14.3 (June 2004), pp. 329–366. DOI: 10.1017/S0960129504004153 (cited on pages 37, 39, 40, 59, 78).

[137] C. A. R. Hoare. *Communicating Sequential Processes*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1985 (cited on page 37).

[138] OMG. 'System Modeling Language, SysML®. Version 1.6'. In: (2020) (cited on page 37).

[139] Nikunj R Mehta, Nenad Medvidovic, and Sandeep Phadke. 'Towards a taxonomy of software connectors'. In: *Proceedings of the 22nd international conference on Software engineering*. ACM. 2000, pp. 178–187 (cited on page 37).

[140] R. K. Pandey. 'Architectural Description Languages (ADLs) vs UML: A Review'. In: *SIGSOFT Softw. Eng. Notes* 35.3 (May 2010), pp. 1–5. DOI: 10.1145/1764810.1764828 (cited on page 37).

[141] Peter Wegner. 'Coordination as constrained interaction (extended abstract)'. English. In: *Coordination Languages and Models*. Ed. by Paolo Ciancarini and Chris Hankin. Vol. 1061. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1996, pp. 28–33 (cited on page 38).

[142] Simon Bliudze and Joseph Sifakis. 'The algebra of connectors–structuring interaction in BIP'. In: *IEEE Transactions on Computers* 57.10 (2008), pp. 1315–1330 (cited on page 38).

[143] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces principles, patterns, and practice*. Addison-Wesley, 1999, p. 344 (cited on page 39).

[144] Mirko Viroli, Jacob Beal, Ferruccio Damiani, Giorgio Audrito, Roberto Casadei, and Danilo Pianini. 'From Field-Based Coordination to Aggregate Computing'. In: Jan. 2018, pp. 252–279. DOI: 10.1007/978-3-319-92408-3_12 (cited on page 39).

[145] Daniel D Corkill. 'Collaborating software: Blackboard and multi-agent systems & the future'. In: 2003 (cited on page 39).

[146] Vitaly Buravlev, Rocco De Nicola, and Claudio Antares Mezzina. 'Tuple spaces implementations and their efficiency'. In: *International Conference on Coordination Languages and Models*. Springer. 2016, pp. 51–66 (cited on page 39).

[147] Gerardo Pardo-Castellote. 'Omg data-distribution service: Architectural overview'. In: *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.* IEEE. 2003, pp. 200–206 (cited on page 39).

[148] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan Rutten. 'Modeling component connectors in Reo by constraint automata'. In: *Science of Computer Programming* 61.2 (July 2006), pp. 75–113. DOI: 10.1016/j.scico.2005.10.008 (cited on page 40).

[149] Kasper Dokter, Sung-Shik Jongmans, Farhad Arbab, and Simon Bliudze. 'Relating BIP and reo'. In: *arXiv preprint arXiv:1508.04848* (2015) (cited on page 40).

[150] Simon Bliudze, Joseph Sifakis, Marius Dorel Bozga, and Mohamad Jaber. 'Architecture internalisation in BIP'. In: *Proceedings of the 17th international ACM Sigsoft symposium on Component-based software engineering - CBSE '14*. 2014, pp. 169–178. DOI: 10.1145/2602458.2602477 (cited on page 40).

[151] Modelica Association. *System Structure and Parameterization*. https://ssp-standard.org/. Nov. 2020 (cited on page 40).

[152] Cláudio Gomes, Bentley Oakes, Mehrdad Moradi, Alejandro Gámiz, Juan Mendo, Stefan Dutre, Joachim Denil, and Hans Vangheluwe. 'HintCO – Hint-based Configuration of Co-simulations'. In: *Proceedings of the 9th International Conference on Simulation and Modeling Methodologies, Technologies and Applications - Volume 1: SIMULTECH,* INSTICC. SciTePress, July 2019, pp. 57–68. DOI: 10.5220/0007830000570068 (cited on page 40).

[153] Moritz Kleine. 'CSP as a Coordination Language. A CSP-based Approach to the Coordination of Concurrent Systems'. In: (2011) (cited on page 41).

[154] Modelica Association. *Functional Mock-up Interface*. https://fmi-standard.org/. Nov. 2020 (cited on page 42).

[155] Youssef Bouanan, Simon Gorecki, Judicael Ribault, Gregory Zacharewicz, and Nicolas Perry. 'Including in HLA Federation Functional Mockup Units for Supporting Interoperability and Reusability in Distributed Simulation'. In: *Proceedings of the 50th Computer Simulation Conference*. SummerSim '18. Bordeaux, France: Society for Computer Simulation International, 2018 (cited on pages 45, 46).

[156] Zhiying Tu, Gregory Zacharewicz, and David Chen. 'A federated approach to develop enterprise interoperability'. In: *Journal of Intelligent Manufacturing* 27.1 (2016), pp. 11–31. DOI: 10.1007/s10845-013-0868-1 (cited on page 46).

[157] Hans Vangheluwe. 'The Discrete EVent System specication (DEVS) formalism'. In: (Jan. 2005) (cited on page 47).

[158] Benoit Combemale, Julien Deantoni, Matias Larsen, Frédéric Mallet, Olivier Barais, Benoit Baudry, and Robert France. 'Reifying Concurrency for Executable Metamodeling'. In: *6th International Conference on Software Language Engineering (SLE 2013)*. Ed. by Richard F. Paige Martin Erwig and Eric van Wyk. Lecture Notes in Computer Science. Indianapolis, Etas-Unis: Springer-Verlag, 2013 (cited on pages 50, 61, 65).

[159] Dorian Leroy, Erwan Bousse, Manuel Wimmer, Tanja Mayerhofer, Benoit Combemale, and Wieland Schwinger. 'Behavioral interfaces for executable DSLs'. In: *Software and Systems Modeling* (Apr. 2020). DOI: 10.1007/s10270-020-00798-2 (cited on page 50).

[160] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007 (cited on page 52).

[161] Lars Ivar Hatledal, Arne Styve, Geir Hovland, and Houxiang Zhang. 'A Language and Platform Independent Co-Simulation Framework Based on the Functional Mock-Up Interface'. In: *IEEE Access* 7 (2019), pp. 109328–109339. DOI: 10.1109/ACCESS.2019.2933275 (cited on pages 52, 53).

[162] Jose Evora-Gomez, José Cabrera, jean-philippe Tavella, Stephane Vialle, Enrique Kremers, and Loïc Frayssinet. 'Daccosim NG: co-simulation made simpler and faster'. In: Feb. 2019, pp. 785–794. DOI: 10.3384/ecp19157785 (cited on pages 52–54).

[163] Paul Baran. 'On Distributed Communications Networks'. In: *IEEE Transactions on Communications Systems* 12.1 (1964), pp. 1–9. DOI: 10.1109/TCOM.1964.1088883 (cited on page 52).

[164] José Juan Hernández-Cabrera, José Évora-Gómez, and Octavio Roncal-Andrés. *JavaFMI*. https://bitbucket.org/siani/javafmi/. 2013 (cited on page 53).

[165] Julien Siebert, Laurent Ciarletta, and Vincent Chevrier. 'Agents and artefacts for multiple models co-evolution. Building complex system simulation as a set of interacting models'. In: 1 (May 2010). DOI: 10.1145/1838206.1838279 (cited on page 55).

[166] Julien Deantoni and Claudio Gomes. 'Towards a Ultimate Formally Verified Master Algorithm'. In: *Short Term Scientific Report COST IC1404* (2018) (cited on page 57).

[167] Ivica Crnkovic and Magnus Peter Henrik Larsson. *Building reliable component-based software systems*. Artech House, 2002 (cited on page 58).

[168] Martin Arnold, Christoph Clauß, and Tom Schierz. 'Error Analysis and Error Estimates for Co-simulation in FMI for Model Exchange and Co-Simulation v2.0'. In: *Progress in Differential-Algebraic Equations*. Ed. by Sebastian Schöps, Andreas Bartel, Michael Günther, E. Jan W. ter Maten, and Peter C Müller. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 107–125 (cited on page 59).

[169] Douglas L Perry. *Vhdl*. McGraw-Hill, Inc., 1993 (cited on page 63).

[170] Peter Fritzson. *Principles of object-oriented modeling and simulation with Modelica 2.1*. John Wiley & Sons, 2004 (cited on page 63).

[171] Albert Benveniste, Benoît Caillaud, Hilding Elmqvist, Khalil Ghorbal, Martin Otter, and Marc Pouzet. 'Multi-Mode DAE Models - Challenges, Theory and Implementation'. In: *Computing and Software Science: State of the Art and Perspectives*. Ed. by Bernhard Steffen and Gerhard Woeginger. Cham: Springer International Publishing, 2019, pp. 283–310. DOI: 10.1007/978-3-319-91908-9_16 (cited on page 64).

[172] Charles Andrè, Frédéric Mallet, and Robert De Simone. 'Modeling Time(s)'. In: *ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS/UML)*. Vol. LNCS 4735. Lecture Notes in Computer Sciences. The original publication is available at www.springerlink.com (http://dx.doi.org/10.1007/978-3-540-75209-7_38). Nashville, TN, United States: Springer, Oct. 2007, pp. 559–573. DOI: 10.1007/978-3-540-75209-7_38 (cited on page 66).

[173] Wlodzimierz Lewandowski, Gerard Petit, and Claudine Thomas. 'Precision and accuracy of GPS time transfer'. In: *IEEE Transactions on Instrumentation and Measurement* 42.2 (1993), pp. 474–479 (cited on page 66).

[174] OMG. 'UML Profile for MARTE (Modeling and Analysis of Real Time Embedded Systems)'. In: *Object Management Group* v1.2 (Dec. 2018) (cited on page 66).

[175] Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann. 'POLYCHRONY for System Design'. In: *Journal of Circuits, Systems and Computers* 12.03 (2003), pp. 261–303 (cited on pages 66, 77).

[176] Antoine Viel. 'Implementing stabilized co-simulation of strongly coupled systems using the Functional Mock-up Interface 2.0'. In: Mar. 2014. DOI: 10.3384/ecp14096213 (cited on page 67).

[177] Julien Deantoni, Diallo Papa Issa, Joël Champeau, Benoit Combemale, and Ciprian Teodorov. *Operational Semantics of the Model of Concurrency and Communication Language*. Research Report RR-8584. Sept. 2014, p. 23 (cited on page 68).

[178] Giovanni Liboni and Julien Deantoni. 'A Semantic-Aware, accurate and efficient API for (co-)simulation of CPS'. In: *4th Workshop on Formal Co-Simulation of Cyber-Physical Systems – conjointly with the 18th edition of the International Conference on Software Engineering and Formal Methods* (2020), pp. 1–16 (cited on pages 74, 85).

[179] Roland Pelayo. *Use LM393 IR Module as Motor Speed Sensor*. https://www.teachmemicro.com/lm393-ir-module-motor-speed-sensor/ (cited on page 76).

[180] Giovanni Liboni and Julien Deantoni. 'WIP on a Coordination Language to Automate the Generation of Co-Simulations'. In: *2019 Forum for Specification and Design Languages (FDL)*. IEEE. 2019, pp. 1–4 (cited on page 81).

[181] Charles André. *Syntax and semantics of the clock constraint specification language*. Tech. rep. 6925. INRIA, 2009 (cited on page 83).

[182] Julien Deantoni, Charles André, and Régis Gascon. *CCSL denotational semantics*. Research Report RR-8628. Inria, Nov. 2014, p. 29 (cited on page 83).

[183] Albert Benveniste, Paul Caspi, Stephen A Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. 'The synchronous languages 12 years later'. In: *Proceedings of the IEEE* 91.1 (2003), pp. 64–83 (cited on page 108).

[184] Cristiano Giuffrida, Anton Kuijsten, and Andrew S Tanenbaum. 'EDFI: A dependable fault injection tool for dependability benchmarking experiments'. In: *2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing*. IEEE. 2013, pp. 31–40 (cited on page 110).

[185] Hamid R Zarandi, Seyed Ghassem Miremadi, and Alireza Ejlali. 'Fault injection into verilog models for dependability evaluation of digital systems'. In: *Second International Symposium on Parallel and Distributed Computing, 2003. Proceedings.* 2003, pp. 281–287. DOI: 10.1109/ISPDC.2003.1267675 (cited on page 110).

[186] Todd A Delong, Barry W Johnson, and Joseph A Profeta. 'A fault injection technique for VHDL behavioral-level models'. In: *IEEE Design Test of Computers* 13.4 (1996), pp. 24–33. DOI: 10.1109/54.544533 (cited on page 110).

[187] Marco Di Natale, Francesco Chirico, Andrea Sindico, and Alberto Sangiovanni-Vincentelli. 'An MDA Approach for the Generation of Communication Adapters Integrating SW and FW Components from Simulink'. English. In: *Model-Driven Engineering Languages and Systems*. Ed. by Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran. Vol. 8767. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 353–369. DOI: 10.1007/EPTCS (cited on page 116).

[188] Per Bjuréus and Axel Jantsch. 'Modeling of mixed control and dataflow systems in MASCOT'. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 9.5 (Oct. 2001) (cited on page 116).

[189] Frédéric Boulanger and Cécile Hardebolle. 'Simulation of Multi-Formalism Models with ModHel'X'. In: *Proceedings of ICST'08*. IEEE Comp. Soc. 2008, pp. 318–327 (cited on page 116).