THÈSE DE DOCTORAT DE
l'UNIVERSITÉ SORBONNE

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Salah Eddine SAIDI

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ SORBONNE

Sujet de la thèse :

Approches de Parallélisation Automatique et d'Ordonnancement pour la Co-simulation de Modèles Numériques sur Processeurs Multi-cœurs

soutenue le

devant le jury composé de :

M. Pierre Siron	Rapporteur
M. Nicolas Navet	Rapporteur
M. Lionel Lacassagne	Examinateur
M. Ramine Nikoukhah	Examinateur
M. Hassen Hadj Amor	Examinateur
M. Nicolas Pernet	Encadrant
M. Yves Sorel	Encadrant
Mme. Liliana Cucu-Grosjean	Directrice de thèse

Abstract

When designing complex cyber-physical systems, engineers have to integrate numerical models from different modeling environments in order to simulate the whole system and estimate its global performances. If some parts of the system are physically available, it is possible to connect these parts to the simulation in a Hardware-in-the-Loop (HiL) approach. In this case, the simulation has to be performed in real-time where models execution consists in periodically reacting to the real (physically available) components and providing periodic output updates. The increase of requirements on the simulation accuracy and its validity domain requires more complex models. Using such models, it becomes hard to ensure fast or real-time execution without using multiprocessor architectures. FMI (Functional Mocked-up Interface), an increasingly common standard for model exchange and co-simulation, offers new opportunities for multi-core execution of numerical models. One goal of this thesis is the extraction of potential parallelism in a set of interconnected multi-rate models. We build on the RCOSIM approach that has been previously developed at IFP Energies nouvelles and which allows the parallelization of FMI models on multi-core processors. It is based on representing the co-simulation by a dependence graph model. In the first part of the thesis, improvements have been proposed to overcome the limitations of RCOSIM. In particular, we propose new algorithms in order to allow handling models that exchange data at different rates and schedule them on multi-core processors. Also, the improvements allow handling specific constraints such as mutual exclusion and real-time constraints. Second, we propose algorithms for the allocation and non preemptive scheduling of the dependence graphs, taking into account their real-time, data dependence and allocation constraints. These algorithms aim at accelerating the execution of the co-simulation or ensuring its real-time execution in a HiL approach. The proposed solutions have been tested on randomly generated dependence graphs and validated against an industrial use case which is an internal combustion engine co-simulation. This thesis is part of a joint action IFP Energies nouvelles - Inria in which Inria brings its real-time systems experience to the numerical simulation challenges of IFP Energies nouvelles.

Contents

List of Figures

1 Introduction

Contents

This thesis deals with the parallelization of co-simulations of numerical models on multi-core architectures. In particular, it focuses on the acceleration and real-time execution of co-simulations through multi-core parallelization. Different research questions related to the aforementioned problems are studied. The work presented in this thesis represents a continuation of two PhD theses that have been previously conducted at IFP Energies nouvelles[1]. See [1, 2]. This first chapter gives an introduction to the research topic of the thesis. First, we explain the general context of the studied research problems. Then, we briefly present the objectives of the thesis. Finally, we give the structure of the thesis.

1.1 Context

The number of computers has grown very fast in recent decades and today they are omnipresent. The most known kind of computers is general purpose computers that are used for human consumption. However, the vast majority of the computers around the world are less visible and are used for different purposes, mainly for controlling physical entities. These computers are called embedded systems.

Systems that combine computational elements and physical processes are referred to as Cyber-Physical Systems (CPS) [3]. The diversity of the involved disciplines makes the process of building CPS challenging, costly and time consuming. Therefore,

[1] www.ifpenergiesnouvelles.fr

applying appropriate methodologies that respond to challenges related to the design, the development and the validation of CPS, is a crucial requirement. In particular, enabling the prediction of the system's behavior before its deployment has the potential to reduce the risks, the cost and the needed effort. Simulation is an efficient way to achieve these requirements as it allows imitating the functioning of the system on a computer and assessing its design. System designers can then identify potential design flaws and correct them before deploying the system.

In order to perform the simulation, the system is first modeled. Traditionally, subparts of the system are modeled separately, and then integrated into one environment to perform simulation at the system level. There exist several modeling formalisms, each of which is adapted to certain kinds of problems. In the modeling formalism considered in this thesis, a model is represented by a set of Ordinary Differential Equations (ODEs) that describe the dynamics of the modeled system. The evolution of the simulation consists in numerically integrating the ODEs while minimizing the error.

The simulation of dynamical systems such as CPS can be accomplished in different ways according to the desired goal. In co-simulation, the different subsystems are described by models of equations and connected together to simulate the whole system on a computer. In this case, synchronized communications have to be ensured between the different models where each model must be able to detect and respond to events raised by the other models. Integrating heterogeneous models usually results in a complex and computationally expensive co-simulation which increases the demand of processing power. Consequently, a principal challenge of co-simulation is the question of how to reduce the execution time. As is well-known, increasing CPU frequency by means of silicon integration has reached its possible limits and semiconductor manufacturers switched in last years to building multi-core processors, i.e. integrating multiple processors into one chip allowing parallel processing on a single computer. Multi-core processors allow reducing the execution time of a program by partitioning it into a set of computational tasks and assigning a subset of tasks to each core to be processed in parallel.

In Hardware-in-the-Loop (HiL) co-simulation, physically available components, e.g. controller hardware, are connected to simulated models on a computer. The controller hardware runs the control algorithm (controller software) and is connected to the simulation computer via electronic interfaces. The goal here is to emulate the behavior of the real physical process so as to run the controller software under realistic conditions. The HiL approach is usually used to test the controller software on its final execution platform. However, the physically available component can instead be a part of the physical process. In HiL, two concepts of time have to be correctly meshed: the simulated time and the real time. Achieving a correct meshing of the simulated time and the real time defines a set of timing constraints imposed on the simulated models. These constraints have to be considered during the execution of the co-simulation. It is not always possible to satisfy these constraints, especially on single-core processors. Performing HiL co-simulations on multi-core processors can enhance the opportunities of satisfying timing constraints which are infeasible on single-core processors.

1.2 Objectives

There are two main research focuses in this thesis: acceleration of co-simulation and HiL co-simulation under real-time constraints on multi-core architectures. We are interested in co-simulations of CPS that are compliant with the FMI standard [4]. FMI facilitates the coupling of diverse models originating from different developers and tools. As already stated, a main problem of co-simulations is their expensive computational cost. Unfortunately, many simulation tools have single-core simulation kernels and do not take advantage of the computation power brought by multi-core architectures. Therefore, enabling parallel execution of computationally expensive co-simulations on multi-core processors is keenly sought by the developers and the users of simulation tools. In this context, we aim at developing appropriate algorithms to efficiently exploit the parallelism provided by multi-core processors in order to accelerate FMI co-simulations and possibly satisfy timing constraints of HiL co-simulations. Different approaches for parallelizing co-simulations are possible and have already been explored. In this thesis, we build on the existing solutions developed at IFP Energies nouvelles and seek to improve them.

Developed at IFP Energies nouvelles, xMOD is a co-simulation and a virtual experimentation platform which allows mixing stand-alone and tool coupling co-simulations and optimizing complex models execution. The Refined CO-SIMulation (RCOSIM) approach [5] is the parallelization approach used in xMOD. It uses the information given by FMI about input-output relationships inside a model that is exported as a Functional Mock-up Unit (FMU). A model's FMU is a package that encapsulates an XML file containing, among other data, the definitions of the model's variables, and a library defining the equations of the model as C functions. Given these features, various execution possibilities can be realized. The parallelization of co-simulation models on a multi-core processor can be seen as the following problem: find an allocation of the functions of the different models to the different cores and define an execution order, i.e. schedule the functions that are allocated to each core. When solving this problem, the utilization of the available cores has to be optimized in order to achieve the best acceleration. Using parallel computing terminology, the problem consists in finding a schedule for all the functions of the co-simulation on a multi-core processor. In this thesis, we continue the work on RCOSIM by addressing some of its limitations, presented below, in order to improve its performance and also to extend its use to different kinds of co-simulations.

In [1] a set of rules is defined for propagating timing constraints from a physically available component to simulated models in a HiL co-simulation. It defines the constraints for each model of the co-simulation. In this thesis, we extend these rules to apply them on FMI compliant models. Furthermore, we propose non preemptive multi-core scheduling algorithms to satisfy the defined timing constraints.

The contributions of this thesis are the following:

- **A dependence graph model for representing FMI co-simulations:** Our contributions consist in extending the dependence graph model of the RCOSIM approach to handle additional requirements and constraints as follows:

1. We extend the mono-rate dependence graph model to handle multi-rate FMI co-simulation. Such co-simulation involves FMUs that are assigned different communication step sizes which define the data exchange rates of the FMUs. Such co-simulations cannot be handeled by RCOSIM because the dependence graph model fails to represent the different communication step sizes. We propose a transformation algorithm that transforms the dependence graph of a multi-rate FMI co-simulation. The result is a new graph that represents well the data exchange rates of the different FMUs.
2. We propose a method for handling mutual exclusion constraints between functions of a same FMU. It is not possible to execute such functions in parallel because they share resources, e.g. variables. The RCOSIM approach handles these constraints in a way that limits the attainable acceleration of the co-simulation. Our proposed solution results in a new graph that defines an order of execution for functions that are mutually exclusive. In order to obtain this new graph, we propose an acyclic orientation ILP formulation and heuristic.
3. We add to the dependence graph model real-time constraints to perform HiL FMI co-simulation when some models are replaced by there real counterparts that are physically available. Since real-time constraints are imposed by the real parts on some inputs and outputs of the simulated models, we propose propagation algorithms that assign, according to the dependence, real-time constraints (release and deadline dates) to the nodes of the dependence graph.

- **Multi-core scheduling of FMI co-simulations:** We improve the scheduling heuristic used in the RCOSIM approach and propose other algorithms as follows:

4. We propose non preemptive multi-core scheduling algorithms for the problem of co-simulation acceleration. We improve the multi-core scheduling heuristic used in RCOSIM by using profiled execution times and accounting for synchronization cost. Also, we propose an Integer Linear Programming (ILP) formulation.
5. We propose an implementation of a runtime non preemptive scheduling solution using the Intel TBB library [6] for the acceleration of FMI co-simulation.
6. We propose non preemptive multi-core scheduling algorithms for the problem of HiL FMI co-simulation under real-time constraints. These algorithms consist in an ILP formulation and a heuristic.

1.3 Thesis Outline

The rest of this thesis is structured in six chapters. In Chapter 2 we give basic concepts and preliminaries related to the different domains that are involved in the thesis. In the first section, we present basic concepts of modeling and simulation. First we specify the type of systems that we are interested in. Then, we briefly present several modeling

formalisms with an emphasis on differential equations and hybrid modeling. Next, we present the principle of numerical simulation before defining the concept of co-simulation. Co-simulation under real-time constraints is defined afterward. Finally, we review some of the most known tools for modeling and simulation, and real-time simulation.

In Chapter 3, we give a detailed description of the research problem of this thesis. We start, in the first section, by giving an overview of the problem. In the second section, we present the RCOSIM approach that we aim at improving and extending in this thesis. Next, we list the limitations of RCOSIM. Finally, we present the open research issues and the objectives of the thesis in detail.

Chapter 4 focuses on the first part of our contributions, i.e. a dependence graph model for representing FMI co-simulations. First, we present in detail the method of construction of the dependence graph of an FMI co-simulation, used in the RCOSIM approach. The next section is dedicated to multi-rate FMI co-simulation where we propose a transformation algorithm for multi-rate dependence graphs and give a small illustrative example. Then, we deal with the problem of handling mutual exclusion constraints. We give a detailed description of the problem and formulate it as an acyclic orientation problem. We propose a heuristic and and ILP formulation that perform the acyclic orientation of the dependence graph. In the last section, we describe the problem of propagating real-time constraints before detailing the proposed propagation algorithms. We give small examples to illustrate the algorithms.

In Chapter 5, we present our proposed scheduling heursitics and ILP formulations for both the acceleration of the co-simulation and the execution of HiL co-simulation under real-time constraints.

We evaluate our proposed solutions in Chapter 6. First, we propose a random generator of synthetic FMI co-simulations. Then we compare the performances of the heuristics and the ILP formulations for the acyclic orientation and the scheduling respectively. Finally, we evaluate our approach by applying it on an industrial use case. We compare its performance with RCOSIM and a runtime scheduling solution based on the Intel TBB library.

Chapter 7 concludes this thesis. First, we give a summary of the objectives and the contributions of the thesis. Then, we present some perspectives for future work.

2 Background

Contents

This chapter describes fundamental concepts and gives literature review about research topics that are involved in this thesis. Topics covered include modeling and simulation, co-simualtion, parallel computing, scheduling in the context of parallel computing, and parallelization approaches related to (co-)simulation.

2.1 Modeling and Simulation

The design of complex systems impose the study of their behavior before building them with the objective of allowing preliminary evaluation, tuning and possibly redesign of

the solution. Simulation has proven successful in responding to this need and became an indisputable step in the design process of complex systems. Simulation is an effective way for cost reduction since it allows correcting design errors before building the system. Simulation is performed by providing models which describe the system and then bringing these models to life by running them in order to imitate, on a computer, the behavior of the simulated system over time.

2.1.1 Systems

Before detailing the concepts and methods of modeling and simulation of systems, it is important to understand what is meant by a system. See Definition 2.1.1.

Definition 2.1.1. A system is defined as a set of interacting parts which form a complex whole and operate towards a common goal.

In order to have clearer understanding, the notion of a system should be conceived in the scope of the context that it is used in. In engineering domains, in addition to the definition given above, a system can be seen as an entity which consumes inputs and produces outputs from and to the environment, and is characterized by an internal state. The state of the system is affected by the inputs that it consumes, and, in turn, affects the produced outputs. Figure 2.1 illustrates this view as usually found in block diagrams where u represents the input of the system, y the output, and x the internal state.

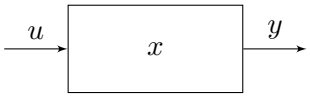


Figure 2.1: A block diagram representation of a system.

A kind of systems that falls within the scope of this thesis is known as Cyber-Physical Systems (CPS) [3]. CPS consist of a combination of tightly or loosely interacting computational elements and physical processes. The computational elements are called embedded systems or controllers and are used to control the physical processes. They are interfaced with the physical processes through sensors and actuators. The aim of the controller is to bring the physical process to a desired state by sending digital control data. If the physical process does not in turn send back data to the controller, the system is referred to as *open loop control*. Alternatively, the controller's behavior is possibly adapted to the change in the state of the physical process which sends feedback data. The term *closed loop control* is used to refer to such interaction between the controller and the physical process. Basically, an error, which is the divergence between the aimed behavior, called reference, and the actual behavior of the physical process is measured and corrected. Figure 2.2 shows a basic example of a CPS with feedback control. It is common that CPS contain multiple feedback loops and involve simple or sophisticated networks that are used for the communication between the different parts of the CPS. A Digital-to-Analog Converter (DAC) is used to convert the data produced by a controller

and consumed by a physical process. Conversely, an Analog-to-Digital Converter (ADC) is used to convert the data produced by a physical process and consumed by a controller.

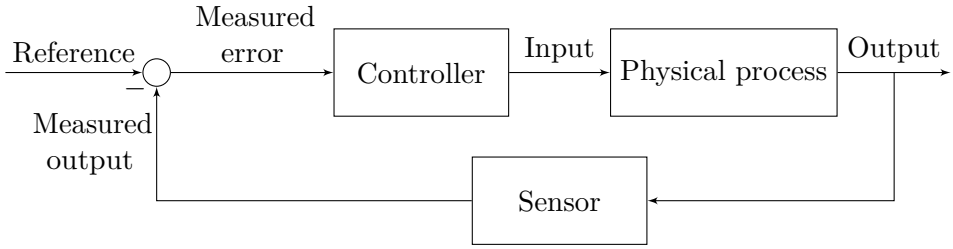


Figure 2.2: Feedback control of a physical process.

Areas where CPS can be found are as important as automotive, aerospace, manufacturing, transportation and many others. The diversity of the involved disciplines makes the process of building CPS challenging, costly and time consuming.

2.1.2 Modeling

Modeling a system consists in creating a mathematical abstraction of its behavior. The first step is to choose a modeling formalism. This choice depends on the properties of the system, the objective of the simulation, and the aimed level of detail in the simulation. For instance, models can be built using continuous time variables to represent continuous dynamics of a physical process. Such mathematical models consist of a set of differential equations which describe the continuously changing physical quantities of a process such as electrical circuits, fluid dynamics, chemical reactions, etc. Other systems feature a behavior which evolves between a finite set of states. These systems can be modeled using formalisms such as DEVS (Discrete Event System Specification) [7], Statecharts [8], and Petri nets [9]. Finally, hybrid modeling allows the modeling of systems with both continuous variables and discrete states. In other words, the system jumps between discrete states and while it is in a certain state, it features a continuous behavior, *i.e.* its quantities change continuously.

In this thesis, we focus on the modeling of dynamical systems using differential equations. A differential equation expresses a variable as a function of its derivatives. In the modeling of dynamical systems, the variables are the physical quantities and the derivatives express their rates of change. A differential equation is called Ordinary Differential Equation, abbreviated ODE, if it involves ordinary derivatives of the variables with respect to an independent variable (usually the time in the modeling of dynamical systems). The ordinary derivative consists in computing the derivative of the function allowing all variables to vary. The term ordinary is used in contrast with the term Partial Differential Equation presented below. Equation 2.1 is an ODE where x is the vector of the variables of interest called the state variables, $\dot{x} = \frac{dx}{dt}$ is the vector of the time derivatives of the state variables, t is the time (the independent parameter), and f is a given function.

$$0 = f(\dot{x}, x, t) \equiv f(\frac{dx}{dt}, x, t) \tag{2.1}$$

A differential equation is called Partial Differential Equation, abbreviated PDE if it involves unknown functions and their partial derivatives. A partial derivative of a function is its derivative with respect to one variable while holding the other variables constant. Equation 2.2 shows such PDE where $x_1, x_2, \ldots, x_n$ are the parameters, $y = y(x_1, x_2, \ldots, x_n)$ is the unknown function, $\frac{\partial y}{\partial x_i} : 1 \leq i \leq n$ are the partial derivatives of y, and f is a given function.

$$0 = f\left(x_1, x_2, \ldots, x_n, y, \frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}, \ldots, \frac{\partial y}{\partial x_n}\right) \tag{2.2}$$

A Differential Algebraic Equation, abbreviated DAE, is a system of equations which involves algebraic equations in addition to differential equations. A DAE can be written in the form shown in equation 2.3 where f represents differential equations involving derivatives of variables and g represents algebraic equations which do not contain derivatives.

$$\begin{aligned} 0 &= f(\dot{x}, x, t) \equiv f(\frac{dx}{dt}, x, t) \\ 0 &= g(x, t) \end{aligned} \tag{2.3}$$

In this thesis, we are particularly interested in modeling dynamical systems using ODEs. Typically, such systems are hybrid dynamical systems modeled using hybrid ODEs. These systems exhibit continuous behavior interspersed with jumps triggered by some events. There are two kinds of events: Events which occur at a particular instant in time are called *time events*. The other kind of events, called *state events* or *zero-crossing*, arise as a result of the value of a state variable crossing a specific threshold. Time events are easier to handle than state events since they occur at known instants in time. A classic example of such hybrid dynamical systems is the bouncing ball model where a ball is dropped from a certain height above the ground. The ball falls with a velocity subject to gravity and bounces when it hits the surface. Equation 2.4 gives a hybrid system of equations which describes the behavior of the bouncing ball where x is the position of the ball (height), v its velocity, g the gravity constant, and γ the coefficient of restitution. The first and second time derivatives of x, $\dot{x} = \frac{dx}{dt}$ and $\ddot{x} = \frac{d^2x}{dt^2}$ represent the velocity and the acceleration of the ball respectively.

$$\begin{aligned} \dot{x} &= v \\ \ddot{x} &= -g \text{ if } x > 0 \\ \dot{x} &:= -\gamma \dot{x} \text{ if } x = 0 \end{aligned} \tag{2.4}$$

The bouncing ball model exhibits a continuous behavior when $x > 0$ and discontinuities occur at bounces, i.e. when $x = 0$, where the velocity of the ball is inverted and scaled down by a factor equal to γ. In other words, the motion of the ball changes from downward to upward. This hybrid dynamical system can be modeled using a hybrid automaton as shown in Figure 2.3.

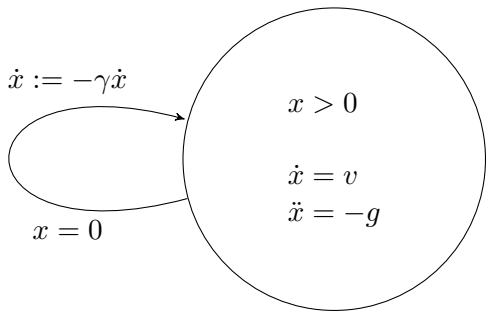


Figure 2.3: Hybrid automaton of the bouncing ball system.

The discrete dynamics found in hybrid dynamical systems should be distinguished from the discrete nature of the controllers. The controllers are digital systems, hence, discrete systems. Control laws, i.e. the algorithms of control can be modeled in the continuous domain. However, since these laws are intended to be implemented on computers, a conversion from the continuous to the discrete domain is needed in order to implement them as controller software. When simulated (see next section) with the physical process, a control law that is modeled in the continuous domain is richer in terms of the information it gives about the controller behavior than its discrete counterpart. Also, continuous control laws allow better optimization of specific criteria that are of interest. The discrete control laws obtained through discretization of the continuous control laws are simulated with the physical processes in order to assess the effects of the discretization and, finally, implement the controller software.

2.1.3 Simulation

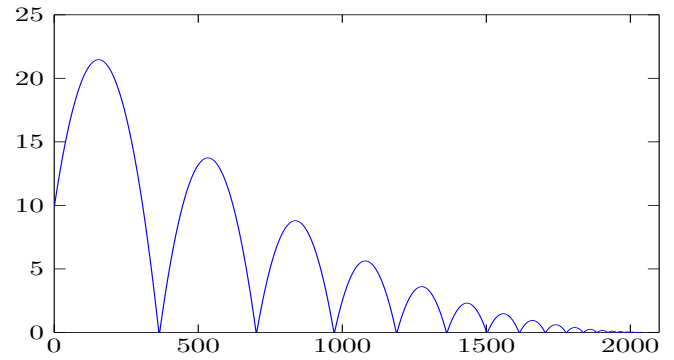
Given a model of the system under study, the simulation consists in running this model in order to produce and plot, on a computer, data consisting in time varying values of the quantities of interest. These data are used in order to assess different aspects about the functioning of the system. Technically speaking, running a model which consists of a set of ODEs means solving these equations. In practice, it is not possible to find the solution of such equations analytically which imposes the use of numerical methods to solve them. Such numerical methods, called solvers, are based on the principle of discretization of the time t. This means that the values of the state variables $x(t)$ of an ODE in the form of equation 2.1 evolve between discrete points of time, called *time steps*, $(t_k, t_{k+1}, \ldots)$ instead of an evolution in continuous time $t \in \mathbb{R}^+$. The distance between two time steps is called the *time step size* or the *integration step size*. The smaller is the integration step size, the more accurate is the numerical resolution of the equations, i.e. the closer it is to the exact solution. However, reducing the integration step size requires more computations and, thus, slows down the execution of the solver. A tradeoff has to be made according to the desired quality and computation speed.

The discretized time can be written as: $t_k = k \times h$ where h is the integration step size and $k \in \mathbb{N}$. The solver computes a sequence: $x_{k+1} = F(x_k, t_k)$ where x_k is the value

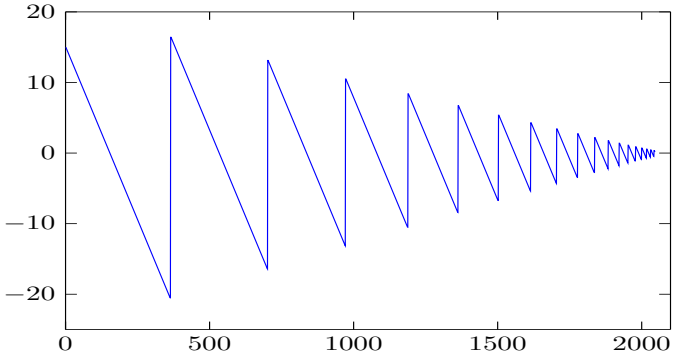
of the variable x at t_k, the k^{th} time step and F is the solver or the integration function. Starting from the initial time t_0 and having the initial condition x_0, which is the value of x at time t_0, the numerical resolution consists in computing approximate values of the quantity x repeatedly with respect to the discretized time. The time step h has to be chosen in such a way that the dynamics of the simulated system are well captured. When the system exhibits dynamics heterogeneity, e.g. fast transients and slow evolutions of the state variables, a fixed time step becomes less efficient. It is therefore more efficient to use a variable time step. A solver with variable step controls the step size using a feedback loop on the error. The step size is adapted according to an estimation of the error.

Solvers are characterized by a number of properties (e.g. order, explicit/implicit, fixed/variable integration step size, convergence, speed, ...) which should be taken into consideration when choosing a solver for a specific problem.

As an example, the simulation of the bouncing ball model represented in Figure 2.3 produces the plots shown in Figure 2.4. The time-varying position and velocity of the ball as it alternates between downward and upward movements are shown in Figure 2.4a and Figure 2.4b respectively.



(a) Position



(b) Velocity

Figure 2.4: Trajectories obtained from the simulation of the bouncing ball model

Quantized State System (QSS) methods [10] offer an alternative to modeling an simulation methods based on time discretization. Their principle is based on discretizing the state and considering the time as continuous. The resolution consists in solving for the time

when the state changes by a quantum. These methods are out of the scope of this thesis.

2.1.4 Co-simulation

Complex systems may involve heterogeneous interacting parts. For instance, In a CPS, the controlled physical process constitutes a multi-physics system and is modeled in the continuous time domain using (hybrid) Ordinary Differential Equations (ODEs). On the other hand, because they are implemented on embedded computers, numerical laws that control the physical process may be modeled in the discrete time domain. For such systems, it becomes necessary to do the modeling at the subsystem level, sometimes without a detailed view about the other subsystems. Models are then coupled together in order to perform simulation at the system level, known as co-simulation. In co-simulation, the different models are simulated in a black-box fashion and an orchestration of their interactions has to be ensured. The interactions between the involved models consist in data exchange.

Co-simulation is an alternative approach to monolithic simulation where a complex system is modeled as a whole using differential equations and simulated by numerically solving these equations. Co-simulation has a number of advantages over the monolithic approach. It allows modeling each part of the system using the most appropriate modeling tool instead of using a single one. Also, it allows a better intervention of the experts of different fields at the subsystem level. Furthermore, co-simulation facilitates the upgrade, the reuse, and the exchange of models.

In co-simulation of models based on ODEs, the equations of each model are integrated using a solver separately. Models exchange data by updating their inputs and outputs at fixed points in time called *communication steps*. The distance between two communication steps is referred to as *communication step size* and denoted H. The communication step size associated with a model is a multiple of the integration step size of the model and defines the rate of communication (data exchange) of this model. It does not make any sense to use a communication step size that is smaller than the integration step size because communication should only be performed at points corresponding to integration steps. Thus, the communication step size should be at least equal to the integration step size. Using a communication step size that is a multiple of the integration step size is interesting when the inputs or the outputs of the model don't need to be updated at every integration step. For instance, if at one out of two integration steps, the model consumes new input values, its communication step size can be set to two times its integration step size. Therefore, the equations of the model are computed at every integration step, and its inputs and outputs are updated at every communication step. Figure 2.5 shows the evolution of time and data exchange in a co-simulation of two models A and B. In this example, the equations of model A are solved using a fixed step size h_A whereas the equations of model B are solved using a variable step size h_B. The communication step size H is considerably larger than both integrations step sizes, allowing fast progress of the integration of the equations by restricting the data exchange to occur at communication steps only. Between communication steps, each model considers that the values of data

produced by the other model are held constant. Another alternative is to estimate these values by employing extrapolation techniques [11, 12].

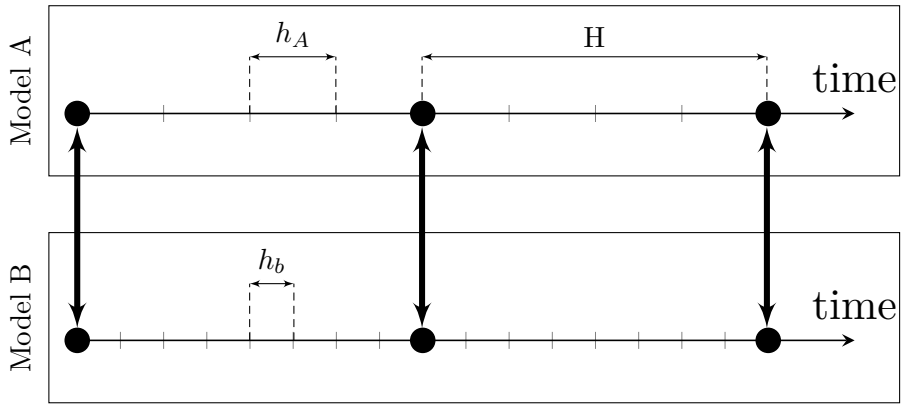


Figure 2.5: Time evolution and data exchange between two models during co-simulation.

For larger co-simulations involving more models, different communication step sizes may be associated with different models. In this case, we talk about *multi-rate* co-simulation.

In this thesis, we are interested, in particular, in co-simulations that are compliant with the Functional Mock-up Interface (FMI) standard [4], presented hereafter. There exist other standards which target the coupling of simulators, e.g. the High-Level Architecture (HLA) [13]. We are interested in the FMI standard because it is adopted by many modeling and simulation tools and is becoming the state of the art standard for co-simulation, thanks to the different possibilities of simulators and models coupling that it offers.

In the context of CPS which contain embedded systems controlling physical processes, different kinds of co-simulation, presented below, can be performed depending on the stage of the controller design.

Model-in-the-Loop

At an early stage of the controller design, Model-in-the-Loop (MiL) co-simulation is performed. In MiL, the model of the controller is included with the model of the controlled physical process in a co-simulation in order to test and validate the functioning scenarios of the controller. By using a system model, MiL aids in the design of control algorithms and also the investigation of design concepts. Once the functions of the control algorithm are specified, the controller software can be implemented.

Software-in-the-Loop

In the next stage, the controller software can be implemented to perform Software-in-the-Loop (SiL) co-simulation. The controller software code can be generated from the controller model. It is then integrated with the simulated models and executed on the computer that runs the simulation. SiL is an inexpensive approach to perform realistic

tests of the controller's performance without the need for using a special hardware. It is common to move back and forth between the MiL and the SiL stages to make necessary rectifications if design flaws are detected.

Hardware-in-the-Loop

After the verification of the controller software, the next stage consists in performing Hardware-in-the-Loop (HiL) co-simulation. In HiL, the controller software is implemented on the controller hardware (e.g. Electronic Control Unit) which is connected to the computer that runs the simulation of the physical process. HiL must run in real-time in order to imitate the real interactions between the controller and the physical process. If any problems are detected, one can go back to SiL or MiL stage to make necessary corrections. Lamberg and Wältermann enumerate the following advantages of HiL [14]:

- The controller algorithm can be tested in an early stage of the development process, allowing early potential corrections and tuning.
- HiL is an efficient alternative for expensive field trials, experiments in borderline zones and hazardous situations.
- Parameters can be tuned in order to perform tests under unusual conditions, e.g. extreme weather conditions.
- Failures that could lead to catastrophic damages in the real system can be tested and corrected systematically.
- If needed, the tests can be reproduced repeatedly and automatically with high precision.

Figure 2.6 gives a view of the different types of co-simulation performed as part of the process of controller design.

Coupling models and performing co-simulation presents many technical challenges. Some attempts have been made to establish methods that allow easy coupling of models and running co-simulations. In the following, we present FMI, a prominent industrial standard that was developed for model exchange and co-simulation.

The Functional Mock-up Interface Standard

The Functional Mock-up Interface (FMI) is a tool-independent and open standard designed in the context of the European ITEA MODELISAR project[1] and is currently developed and maintained by the Modelica Association[2] which promotes the Modelica language (see Section 2.1.6). The FMI standard was developed in order to facilitate the co-simulation of dynamical systems, such as CPS. It provides specifications in order to enable the exchange

[1] itea3.org/project/modelisar.html
[2] www.modelica.org/association/

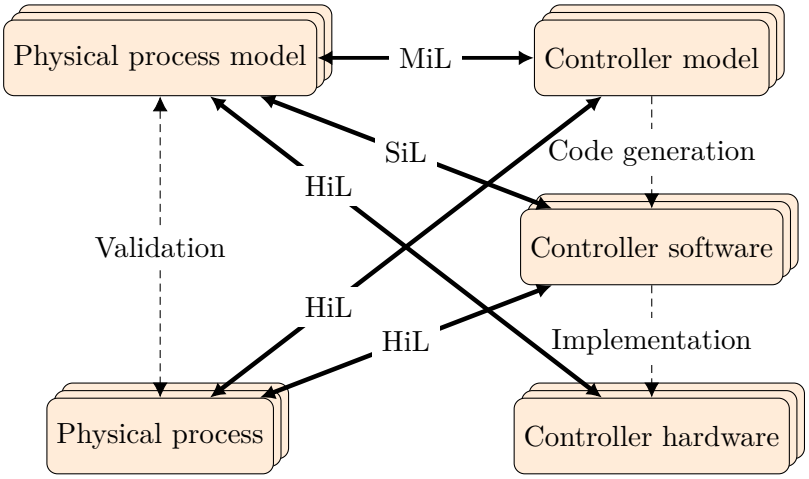


Figure 2.6: Different types of co-simulation involved in the process of controller design.

and the co-simulation of heterogeneous dynamical models that may be developed by different tools. A modeling tool that supports FMI can export a model as a Functional Mock-up Unit (FMU) which can be used in co-simulation environments. FMI defines interfaces for the involved models to allow their co-simulation.

An FMU is a package that encapsulates different files:

- An XML file that contains, among other data, the definition of the different variables of the models and the description of the dataflow between these variables.
- Model functions: Standardized C functions that are used to create instances of the FMU and run them. The functions can be provided as platform dependent binaries (e.g. DLL files) or as C source code.
- Documentation: Optional files that contain documentation about the model.

The FMI standard is organized in two parts:

- FMI for Model Exchange: This specification provides interfaces and defines how model equations should be encapsulated in components. It allows solving each model independently using custom solvers. Accessing and computing the equations is done through standardized function calls. Figure 2.7 illustrates the principle of FMI for Model Exchange.
- FMI for Co-Simulation: This specification defines interfaces between a master algorithm and slave models. It is intended to couple different simulators (models with their solvers) in a co-simulation environment. Figure 2.8 illustrates the principle of FMI for Co-Simulation.

In the context of FMI, we talk about model export and import. Model export means that a model is developed in one tool and then shipped as an FMU. Model import refers

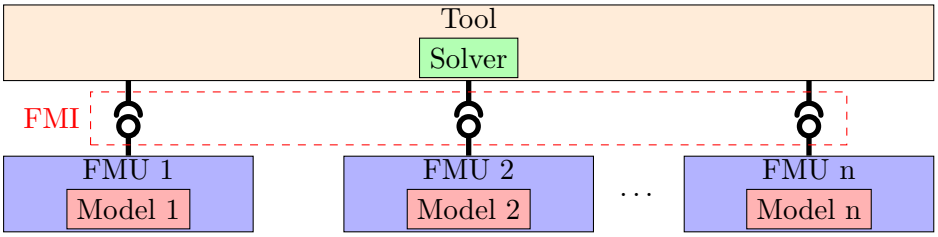


Figure 2.7: FMI for Model Exchange.

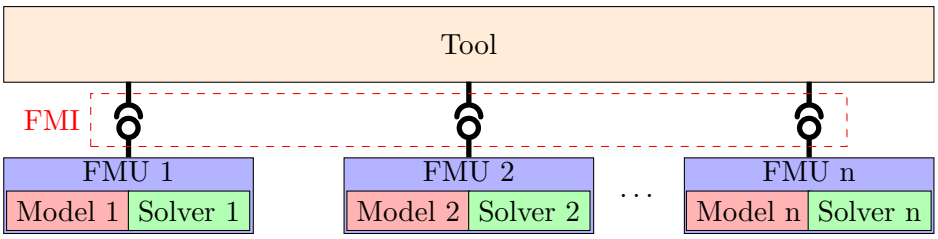


Figure 2.8: FMI for Model Co-Simulation.

to using an FMU in a co-simulation environment different than the tool that was used to develop the FMU. In Figure 2.7 and Figure 2.8, the different FMUs are imported into and executed within the co-simulation environment.

The High-Level Architecture Standard

The High-Level Architecture (HLA) [13] is an IEEE standard developed by the U.S. Modeling and Simulation Coordination Office (M&S CO)[3]. It consists in a specification of a software architecture that allows building a distributed simulation composed of several interacting simulations. Largely used for defense applications, it later gained popularity in the civil domain.

In HLA, each involved simulation is called a *federate* and the distributed simulation composed of interacting federates is called a *federation*. The specification describes how communication is performed within a federation through the *Run Time Infrastructure* (RTI) which is a middleware that provides services for data exchange, synchronization, and coordination between the federates.

An HLA specification includes the following elements:

- An interface specification that describes how federates can be integrated and coordinate by using services provided by the RTI. An Application Programming Interface (API) is provided by the RTI to this end.
- An Object Model Template (OMT) that specifies a framework for communication between the federates. It comprises the Federation Object Model (FOM) which describes the interactions for the whole federation and the Simulation Object Model (SOM) which describes the interactions for one federate.

[3] www.msco.mil

- A set of rules about that have to be respected in the federations and the federates.

Figure 2.9 shows an example of an HLA federation.

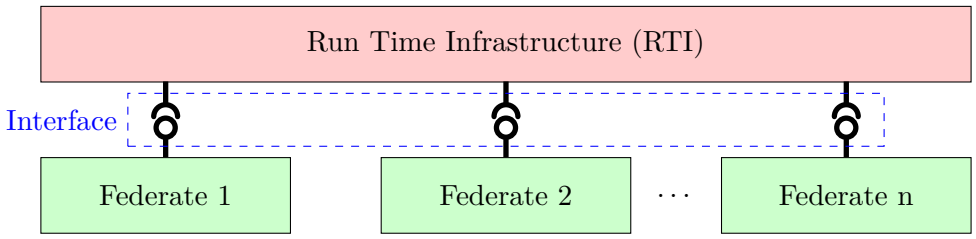


Figure 2.9: HLA federation.

2.1.5 Co-simulation under Real-time Constraints

In the design process of complex systems, it is often necessary to test the behavior of the system or a part of the system as it would be produced by the real system. Therefore, the co-simulation is executed under real-time constraints such that the progress of the simulated time matches the real-time. For example, if temperature takes three minutes to reach 25° in the real system, the simulated temperature has to take three minutes as well to reach the same value. Co-simulation under real-time constraints has to be executed such that the simulated time is advanced at the same speed as real-time.

A typical application of co-simulation under real-time constraints is HiL co-simulation. The key advantage of co-simulation under real-time constraints is that it allows testing the controller under realistic conditions even if the physical process is not available.

Roughly speaking, the difference between co-simulation under real-time constraints and co-simulation without real-time constraints is related to the notion of results validity. For co-simulation without real-time constraints, one seeks to obtain results as soon as possible. The validity of the results depends only on their numerical accuracy. For co-simulation under real-time constraints, the validity of results depends not only on their numerical accuracy but also on their availability time, i.e. they have to be available within specific time *deadlines*. If such deadlines are missed, the results are considered invalid even if their values are correct from a numerical standpoint.

Figure 2.10 illustrates the time evolution of a co-simulation under real-time constraints in comparison to accelerated co-simulation without real-time constraints. For the former, during the resolution of the differential equations, the value of each variable x is computed at every time step. The computation of x_{n+1}, the value of x at time step t_{n+1}, cannot start before the value of x_n is computed as the computation of x_{n+1} depends on the value of x_n. If the time required to compute the value of x_{n+1} exceeds the step size, the real-time constraints are violated which makes the co-simulation invalid. This is known as an *overrun*. In the field of real-time systems, we talk also about *deadline miss*. In the context of co-simulation under real-time constraints, the deadline of a variable computation is the time step by which the value of the variable has to be provided, e.g. time step t_n for the computation of x_n.

A co-simulation under real-time constraints is executed repeatedly. The execution is driven by real-time periods related to data exchange between the simulated part and the real part, e.g. the controller. This period can be different (usually greater) than the integration step sizes used in the co-simulation. In this case, not all computations are required to meet their deadlines. Instead, we seek *rendezvous* points where time steps and real-time periods match. Only the computations whose deadlines correspond to these points must not overrun. Hence, a co-simulation can be guaranteed to satisfy real-time constraints even if the rest of the computations miss their deadlines. Note that this is the main reason we use the term co-simulation under real-time constraints instead of real-time co-simulation. In fact, the latter implies that all computations are subject to real-time constraints.

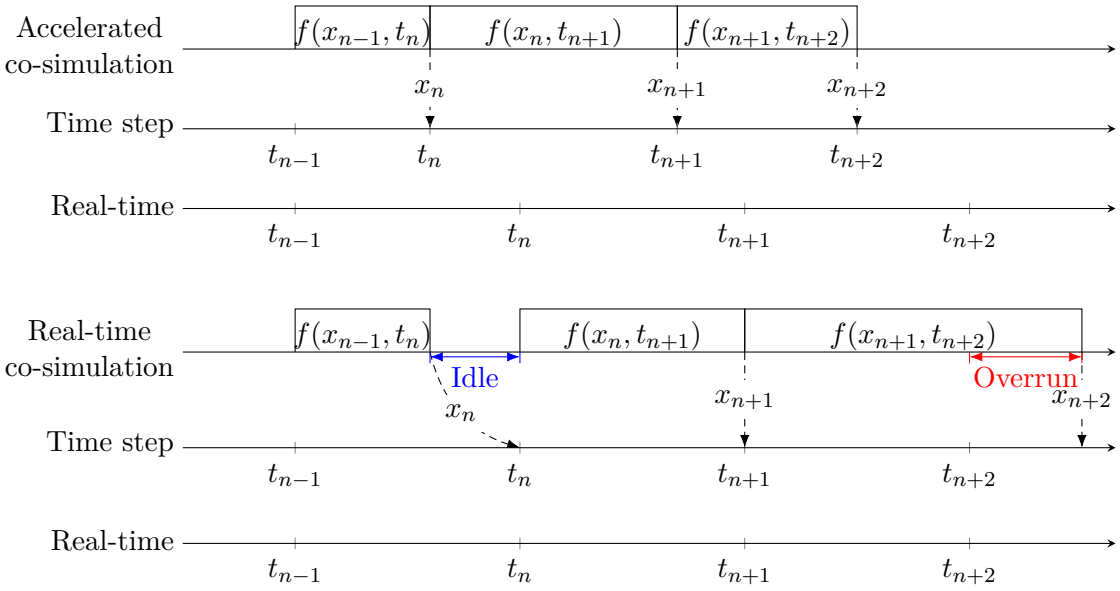


Figure 2.10: Comparison of accelerated co-simulation and co-simulation under real-time constraints.

2.1.6 Languages and Tools for Modeling and Simulation

Building models of systems can be done manually and then transformed into software using general purpose programming languages such as C. However, this approach is not efficient in practice, especially when the modeled system is complex and changes may be required in the model. Many tools and languages for modeling and simulation have been developed in order to facilitate and make the modeling and simulation more efficient. Such tools allow the user to specify an equation-based model in a straightforward manner and come with built-in solvers. Below, we present a non exhaustive list of tools and languages for modeling and simulation.

MATLAB Simulink

Simulink[4], developed by The Mathworks is a graphical modeling environment. Simulink can be used in the process of designing embedded systems. It allows the simulation of embedded systems with the controlled physical processes. Models are built graphically in Simulink using block diagrams. In addition, Simulink is integrated in MATLAB which allows the incorporation of MATLAB functions in Simulink models. Finally, Simulink enables automatic code generation from models.

Modelica

Modelica [15] is an object-oriented equation-based language for the modeling of complex physical systems developed by the Modelica Association. Modelica allows the modeling of physical systems by writing a set of equations. Modelica adopts an acausal approach, i.e. the direction of the signal is not specified in the model. The simulator has to perform symbolic manipulations in order to define inputs and outputs and find an order of execution for these equations. There exist several tools that are based on the Modelica language.

OpenModelica[5] is an open-source modeling and simulation environment based on the Modelica language. It is developed and maintained by the Open Source Modelica Consortium (OSMC). OpenModelica supports the FMI for Model Exchange standard.

Dymola[6], developed by Dassault Systèmes AB, is a modeling and simulation environment based on the Modelica modeling language. Dymola supports the FMI standard and allows interfacing with other tools such as Simulink.

LMS Imagine.Lab Amesim

LMS Imagine.Lab Amesim[7] is a modeling and simulation software developed by Siemens PLM Software. It can be used for the modeling and simulation of mechatronic systems. It is based on the Modelica modeling language. It is oriented towards the modeling of complex physical systems instead of controller design. LMS Imagine.Lab Amesim provides libraries containing collections of components that can be loaded and connected by the user to build models. For simulation, LMS Imagine.Lab Amesim automatically selects a solver that is adapted to the problem. It supports the FMI standard.

xMOD

xMOD[8] is the modeling and co-simulation software developed by IFP Energies nouvelles. It supports the FMI standard and provides an environment for the integration of heterogeneous models built by different parties using different languages and tools. xMOD can execute models with different integration and communication step sizes. Also, it

[4] www.mathworks.com/products/simulink
[5] www.openmodelica.org
[6] www.3ds.com/products-services/catia/products/dymola
[7] www.plm.automation.siemens.com/en_us/products/lms/imagine-lab/amesim
[8] www.xmodsoftware.com

allows the co-simulation of models embedding different solvers or not. In the latter case, xMOD provides a list of different solvers from which the user can choose one for every model. xMOD does not replace original modeling and simulation tools. Instead, it promotes and facilitates their coupling and existence.

DACCOSIM

The Distributed Architecture for Controlled CO-SIMulation (DACCOSIM)[9] is a co-simulation software developed and maintained by EDF Lab Paris-Saclay[10] and CentraleSupélec[11]. It supports the FMI standard and allows distributed co-simulation of FMUs on multi-core architectures or clusters. DACOSSIM is able to execute FMUs with different fixed or variable integration steps.

CosiMate

CosiMate[12] is a co-simulation environment that enables distributed co-simulation. Multiple simulators can be executed on different computers and communicate over a network. CosiMate supports the FMI standard, interfacing with Simulink, and several languages like Modelica, C++, and Java.

Hopsan

Hopsan[13] is a free multi-domain system co-simulation tool developed at the division of Fluid and Mechatronic Systems at Linköping university. Hopsan supports the FMI standard and model export to Simulink.

The additional timing constraints found in real-time simulation require the use of adapted tools. Many real-time simulation tools are developed in such a way to run the simulated part on special dedicated hardware that provides an execution fast enough to ensure real-time constraints. Other solutions tend to enable real-time co-simulation using general purpose computers equipped with Real-Time Operations Systems (RTOS). In [16], the authors give a list of the available real-time simulation tools and detail their characteristics. In the following, we present a non exhaustive list of tools for real-time simulation.

xMOD HiL

xMOD is doted with HiL capabilities. It allows connecting controller hardware to a desktop computer running co-simulation in xMOD using the CAN protocol to perform HiL testing. Also, it features a real-time communication driver, based on the UDP protocol to connect xMOD to different types of HiL platforms such as dSPACE (see below).

[9] sourcesup.renater.fr/daccosim
[10] www.edf.fr
[11] www.centralesupelec.fr
[12] www.cosimate.com
[13] www.iei.liu.se/flumes/system-simulation/hopsan?l=en

Simulink Coder

MATLAB/Simulink offers the Simulink Coder solution for real-time simulation. Simulink Coder generates C/C++ executable code from Simulink models and MATLAB functions. The generated code can be used in a real-time simulation such as HiL testing. The generated code can be deployed with or without a RTOS. Simulink Coder offers three execution modes. In the Single-Tasking Mode, the generated code is executed in a single thread. In the Multi-Tasking Mode, the user specifies sampling periods for parts of the generated code, called rates in Simulink Coder, which are executed and scheduled by a priority-based scheduler. The Asynchronous Mode allows specifying nonperiodic or asynchronous rates. Simulink Coder generates the necessary code to handle such rates.

RT-LAB

OPAL-RT[14], a company specializing in real-time simulation, develops the RT-LAB real-time simulation software. RT-LAB transforms Simulink models into a real-time application by generating and compiling C code. The generated code can be run in parallel on multiple cores. OPAL-RT provides its own hardware targets for the execution of real-time simulation combining COTS parallel computing technologies. Its solution uses a Linux based RTOS.

dSpace

dSpace[15] features the Real-Time Interface (RTI). dSpace relies on Simulink as the modeling tool and uses RTI to extend Simulink Coder for automatic implementation of generated code by Simulink Coder on the real-time hardware of dSpace. An RTI library in Simulink allows adding blocks that implement I/O capabilities to Simulink models. The code generated by Simulink Coder from such models is prepared to be executed on dSpace hardware without manual editing of the code.

RTDS

Developed by RTDS Technologies[16], RTDS is a real-time simulator of power systems. It consists of a custom hardware and a custom software. The hardware is composed of multiple chassis containing each a multi-core processor. The RTDS software is designed for interfacing with the RTDS hardware. It consists of several modules necessary for creating, tuning, loading a simulation into the RTDS hardware, and plotting and visualizing the results.

[14] www.opal-rt.com
[15] www.dspace.com
[16] www.rtds.com

Typhoon HiL

Multiple hardware platforms are proposed by Typhoon HiL[17]. It offers a complete software solution comprising a schematic editor, a module for interfacing with Typhoon hardware, a test suite to run some pre-certification tests, and a power systems toolbox offering a variety of built-in models. Typhoon HiL targets primarily real-time HiL simulation of power systems.

2.2 Parallel Computing

Parallel computing is a very important branch in the computing research and industry. It refers to the discipline that focuses on executing multiple computations simultaneously to solve one problem; thus, accelerating the total time of computation. Its basic idea is to divide a computational task into several sub-tasks that can be performed at the same time. From the beginning of the modern era of computing, computer software has been typically written for sequential execution. In order to solve a problem, an algorithm is designed as a sequence of instructions that are executed one after the other. In order to increase the computation power of computers, the dominant method has for long been frequency scaling. If a processor's frequency is increased, it means that it can execute more instructions per clock cycle and thus can execute a sequential program faster. Moore's law predicted that the number of transistors in a processor would double approximately every two years [17]. This prediction proved correct for many years. However, frequency scaling is facing technological limits and the last decade witnessed a wide shift to multi-core processors among semiconductor manufacturers. The rise of multi-core processors has caused the evolution of many parallel hardware and software technologies.

The main goal of parallel computing is to execute computer programs faster. The speedup obtained from the parallelization can be predicted using Amdhal's law [18]. It states that for a program that is paralellized in order to be executed on multiple processors, the portion of the program that has to be executed sequentially limits limit the attainable speedup. The speedup is, therefore, not linear according to the number of processors and adding more processors does not make the program run faster than the portion of the program that has to be executed sequentially. The following formula gives the theoretical speedup computed using Amdahl's law:

$$S(n) = \frac{1}{(1-P) + \frac{P}{n}} \tag{2.5}$$

$S(n)$ is the theoretical speedup, P is the portion of the program that can be parallelized and n is the number of processors. Figure 2.11 shows the theoretical speedup of a program in function of the number of processors for different values of P. It shows for example that if 50% of a program can be parallelized, the maximum possible speedup is 2, and if 95% of the program can be prallelized, the maximum speedup is around 20.

[17] www.typhoon-hil.com

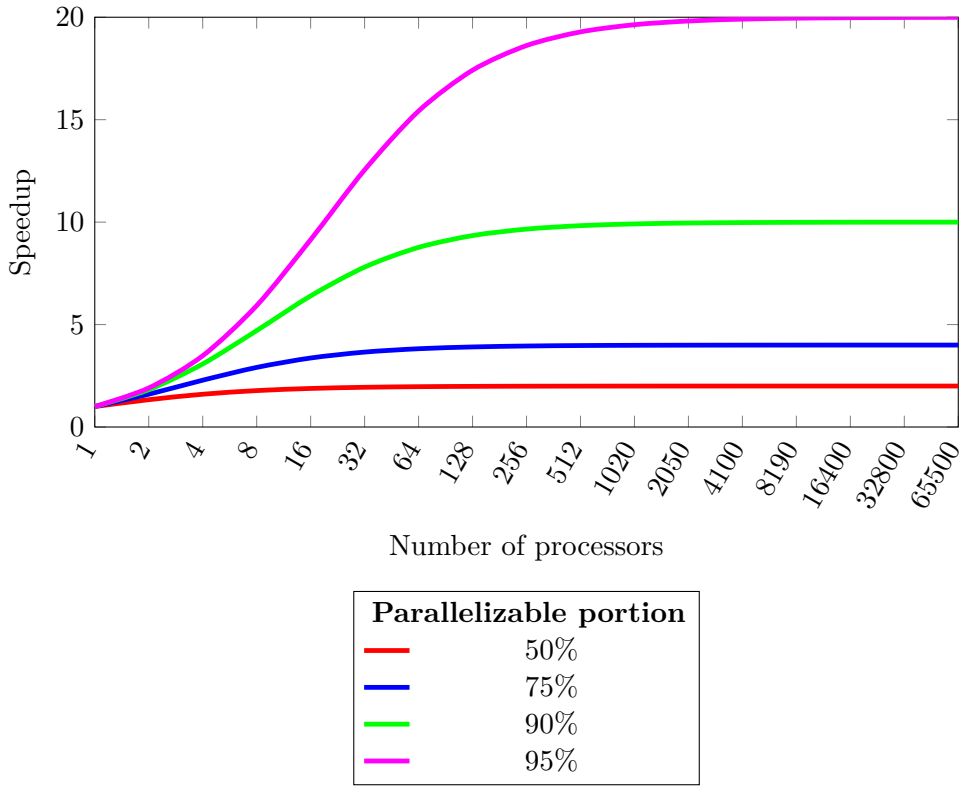


Figure 2.11: Theoretical speedup computed using Amdahl's law for a program in function of the number of processors for different values of P.

2.2.1 Parallelism in Hardware

A computer program consists in a set of instructions. The first computers were only able to execute programs sequentially, i.e. one instruction after another. Later, parallel computing was made possible thanks to the introduction of parallel computers. Parallel computers are of many types, some of which are adapted only to specific kinds of applications. Parallel computers can be classified according to different criteria. Below, we present the common classifications of parallel computers.

Flynn's Taxonomy

The well-known Flynn's taxonomy [19] classifies computers according to instruction and data streams into the following categories:

Single Instruction Stream Single Data Stream (SISD) This is the basic uniprocessor which does not exhibit any parallelism. The execution is sequential where a single instruction stream operates on a single data stream. Examples of such architecture are old desktop computers.

Single Instruction Stream Multiple Data Streams (SIMD) A SIMD computer executes the same instruction stream on multiple data streams in parallel. A Graphics Processing Unit (GPU) is one example of SIMD architectures.

Multiple Instruction Streams Single Data Stream (MISD) Multiple instruction streams are executed on a single data stream. For example, in fault-tolerant computing, the same operation is performed in parallel and the results of all the computations must be the same. Pipeline architectures belong to the MISD class.

Multiple Instruction Streams Multiple Data Streams (MIMD) Different instruction streams are executed on different data streams in parallel. Examples of MIMD computers include multi-core architectures, grid computers, and supercomputers.

Memory Models

Flynn's taxonomy differentiates parallel computers based on their operational behavior. Another important classification of parallel computers is the one based on the organization of the memory.

Shared Memory In this class of parallel architectures, a common memory is shared among multiple processors. All processors access the same global shared memory by operating on a single address space. Communication between the processors is performed through shared memory variables. Shared memory multiprocessors have the advantage of low communication overhead thanks to the proximity of the memory to processors. Scalability is a disadvantage of shared memory multiprocessors as increasing the number of processors creates more traffic between the processors and the memory resulting in memory contention. The latter means conflict between multiple accesses to memory. In practice, shared memory architecture do not scale beyond 16 processors.

There are two kinds of shared memory designs, Uniform Memory Access (UMA) and Non-Uniform Memory Access (NUMA). In the UMA design, the time needed to access the memory is the same for all the processors. This architecture is referred to as Symmetric Multiprocessor also. In the NUMA design, each processor has a local memory, and the shared memory is composed of these local memories. Time to access a specific memory region is not uniform for all processors. Processors access their local memories faster than the local memories of other processors.

Message Passing In the message passing model, also known as distributed memory, each processor has its own memory. Each processor operates on a distinct address space and is only able to access its own memory. As the name suggests, communication between the processors is performed by explicitly passing messages. If a processor requires data from another processor, it explicitly sends a request to this processor and waits for its response. An advantage of the distributed memory architecture is the scalability. If the number of processors is increased, memory is increased also. A disadvantage of the

distributed memory architecture is the time needed to passe messages between processors. This time becomes large in the case of a huge number of processors or long distances between the processors. A typical distributed memory computer is a set of standalone computers interconnected via a network, e.g. Ethernet.

Hybrid Memory It is possible to use both shared and distributed memory in a computer. In this hybrid model, shared memory processors are connected via a network to form a distributed memory architecture. This is the dominant memory architecture in supercomputers today.

2.2.2 Parallelism in Software

Much progress has been made in the design of parallel hardware. That being said, taking advantage of such architectures requires efficient ways for executing software on parallel hardware. A difficult yet integral step in this direction is the process of detecting the parallelism that is inherent in software. This parallelism can be classified into different categories based on the nature of computations that are performed. The main classes of software parallelism are the following:

Data Parallelism

Data parallelism is characterized by performing the same computation on a large set of data. If several processors are available, the data can be distributed across them and the same computation is executed on each processor. For instance a for-loop can be parallelized by distributing the iterations over multiple processors. The same body of the for-loop is executed on all the processors but operates on a different range of iterations on each processor. Data parallelism corresponds to the SIMD parallel hardware.

Task Parallelism

In task parallelism, a program is divided into different computational tasks that are distributed across the processors to be performed in parallel. The challenge here is the question of how to divide the program efficiently so as to obtain the best speedup. In task parallelism, tasks can operate on different data sets. Usually, data dependence exists between the tasks. For instance, the result of one task is needed as input by another task. Such dependence reduces the parallelism. Task parallelism corresponds to the MIMD parallel hardware.

Pipeline Parallelism

Pipeline parallelism combines data and task parallelisms. Multiple tasks operate on streams of data and are executed repeatedly in a sequence. Each task takes its input from the preceding task and produces output to the next task. When a task finishes processing a data element it passes it to the next task and starts processing a new

data element even if the next task has not finished processing. Pipeline processing is common in streaming applications such as video streaming. Pipeline parallelism corresponds to the MISD parallel hardware.

2.2.3 Parallel Programming

In order to efficiently map software parallelism on hardware parallelism, many parallel programming libraries, APIs, and standards have been developed. Basically they differ according to the targeted type of memory. In the following, we define two fundamental concepts found in parallel programming: *processes* and *threads*. Then, present parallel programming models.

Process

A process is an instance of a program. It is characterized by the executable code of the program and its context of execution including a unique process identifier, a memory space, values of the processor's registers, and other system resources. A program is a set of instructions and a process is the actual execution of these instructions. A process contains one or multiple threads (see below). The operating system offers Inter-Process Communication (IPC) mechanisms to handle communication between multiple processes.

Thread

A thread is a unit of execution within a process. Multiple threads can exist within a single process and be executed in parallel. Threads within the same process share the same memory space and the same code. Also, since they share variables, they can communicate directly, in contrast to processes which use IPC. Nevertheless, each thread has its own context including a unique thread identifier, values of the processor's registers, and a call stack.

Let's now present the different parallel programming models with some examples of libraries and standards that follow such models.

Shared Memory Programming

Shared memory programming is based on threads. Multiple threads are created and executed on multiple processors. The programmer does not need to worry about the communication between threads as this is done implicitly via shared variables. Threads may have private variables that are not shared with the other threads. A data consistency problem occurs if two or more threads attempt to write data to the same memory location. Threads must coordinate using synchronization mechanisms in order to avoid data consistency problems. A synchronization mechanism ensures that only one thread can execute a specific segment of code. There are many libraries and APIs for shared memory programming. The following are some examples.

Open Multi-Processing (OpenMP) [20] is an API that has been designed to develop applications that are meant to be executed on shared memory parallel computers such

as multi-core computers. OpenMP is supported by C, C++ and Fortran programming languages. The basic idea of OpenMP is that a master thread is responsible for the creation of slave threads that are allocated to processors to run in parallel. The creation of slave threads is called forking. It is the duty of the developer to specify parts of the code that can run in parallel using preprocessor directives. These directives cause the threads to be created before their execution. When the execution of the slave threads is finished, they join back to the master thread which continues the execution of the program. OpenMP can be used for both data and task parallelism.

Intel Threading Building Blocks (Intel TBB) [6] is a C++ parallel programming library. Using Intel TBB, the developer specifies the parallelism in the form of tasks, not threads. Such tasks are pieces of code that can be executed in parallel but, in contrast to threads, they are not explicitly assigned to hardware resources. For instance, on a multi-core processor, the library creates one thread per core and automatically maps the tasks onto the threads. Therefore, the developer focuses on specifying the parallelism (what can be executed in parallel) instead of handling the parallelism (how to map the parallelism). Intel TBB uses work stealing, i.e. it dynamically tries to balance the computation load among the available processors at runtime.

Shared memory programming can be done using low level multi-threading also. For instance by using POSIX threads (pthreads) or Windows threads. Such low-level approach gives the developer more flexibility and control over the threads, e.g. thread creation and mapping, compared to using libraries such as OpenMP or Intel TBB. Nonetheless, the latter are simpler to use.

Distributed Memory Programming

Distributed memory programming is done using processes that are executed on different processors. The data needs to be partitioned and mapped to the processors with the corresponding tasks. Data is moved between processors if needed. An important challenge is to keep data exchange as low as possible in order to minimize the communication between the processors. Data consistency is not a concern in distributed programming, since each process only writes to the local memory. Nevertheless, the developer needs to implicitly specify the communication between the processors through message passing.

The classical standard for distributed memory programming is the Message Passing Interface (MPI) [21]. MPI is a standard for programming distributed memory parallel computers. It is supported by many programming languages and platforms. It defines a communication protocol for performing the message passing and provides communication and synchronization functionalities for collaborating processes that are allocated to different processors. It supports different kinds of communications such as point-to-point and collective communication. It is also possible to choose the topology of communication to be used.

For more on parallel programming, one can refer to the survey presented in [22]. It gives a very interesting review of the available parallel programming models and tools.

2.2.4 Parallel Scheduling

Parallelization consists in partitioning a sequential program and allocating the different parts in order to be executed on multiple processors. In order to be parallelized, a program needs to be modeled in such a way to express the available parallelism. In general, a model of a program can be made by dividing the program into tasks of computations and defining dependence between them. If the number of the tasks is equal to the number of processors, the parallelization of the program can be achieved by allocating each task to a distinct processor. However, this is not the case in practice, i.e. there are much more tasks than processors. In this case multiple tasks are allocated to one processor and must be executed sequentially. Knowing the time needed to execute each task is also important to model the program. Depending on the application, other properties and constraints can be considered. Having a model of the program, the parallelization consists in defining a schedule for the different tasks, i.e. an allocation to a processor and a time for starting the execution of each task. Parallel computing has received much interest in the scheduling theory community and many algorithms and models have been proposed to solve the problem of application parallelization.

Scheduling in the broad sense refers to the theory, algorithms and systems that deal with problems of sequencing and allocating tasks to resources. Scheduling theory has numerous areas of application like manufacturing, transportation, logistics, sports scheduling, project management, etc.A significant part of the research carried out in the scheduling theory field treats problems related to scheduling computational tasks on parallel computers. This kind of scheduling is known as parallel scheduling. We focus in this section on parallel scheduling from a computing point of view.

In a parallel scheduling problem, the resources are the processors (or cores of a multi-core processor) and the tasks are the computation functions of the application to be executed. Resources are traditionally referred to as computers, or sometimes machines, and tasks can be referred to as jobs. We use the terms processors, to refer to processing elements of a parallel computing system, and tasks to refer to the computational tasks of the application to be executed. A set of n tasks is denoted $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ and a set of m processors is denoted $P = \{p_1, p_2, \ldots, p_m\}$. Scheduling consists in allocating tasks from T to processors from P with respect to predefined criteria, e.g. the minimization of the total execution time of all tasks. Scheduling implies also the definition of an execution order for the tasks that are allocated to the same processor by setting execution start times for the tasks. In general, each task has to be allocated to one and only one processor and a processor can execute at most one task at a time. Additional constraints can be considered depending on the problem.

In scheduling problems, processors can be classified based on their speed of execution [23]:

- *Heterogeneous*: The execution speed of a task depends on both the processor and the task. Not all tasks may be executed on all processors.
- *Homogeneous*: The processors are identical. The execution speed of a given task is

the same on all processors.

- *Uniform*: The execution speed of a task depends only on the speed of the processor. A processor of speed 2 will execute all tasks at exactly twice the speed of a processor of speed 1.

A schedule is called preemptive if the execution of a task can be interrupted by another task of higher priority and resumed later. If a schedule is not preemptive, it is called non preemptive. Furthermore, scheduling algorithms can be classified into online and offline algorithms. Online scheduling algorithms are used when some information about the tasks is not known before the execution. The scheduling algorithm makes scheduling decisions online as the information becomes available. Offline scheduling algorithms can be used when the characteristics of the tasks, such as dependence between them and their execution times, are known before the execution. It is then possible to compute the schedule of the tasks offline.

Scheduling research has been active for over 60 years now and so many methods and algorithms have been proposed to solve different scheduling problems. Different performance measures can be considered such as the makespan objective, the total completion time objective, and the number of late tasks objective [24]. The makepsan is the time needed by a computer to process the whole set of tasks. The general objective of parallel computing is to accelerate the execution of application which corresponds to minimizing the makespan.

Task Dependence Graph

A set of tasks $\mathcal{T}$ which express the parallelism of an application can be represented by a Directed Acyclic Graph (DAG) $G(V, A)$ called the task dependence graph. Each task $\tau_i \in \mathcal{T}$ is represented by a vertex $v_i \in V : 0 \leq i < n$ where n is the number of tasks called, also, the size of graph $G(V, A)$. Dependence between tasks is represented by arcs $(v_i, v_j) \in A : 0 \leq i, j < n$. A vertex may have one or more incoming edges which connect it with its predecessors and one or more outgoing edges which connect it with its successors. A task cannot start its execution unless all its predecessors have finished their execution. Generally, dependence between two tasks is due to data transfer, i.e. one task is executed and produces data that another task needs to consume to start its execution. If a vertex has no predecessor it is called an entry or source vertex. A vertex that has no successor is called an exit or sink vertex. The vertices may be weighted by the execution times of the corresponding tasks. Figure 2.12 shows an example of a task dependence graph. In the remainder of the thesis, for the sake of simplicity we will use the term dependence graph instead of task dependence graph.

Potential and Effective Parallelisms

In industrial practice, we distinguish between the *functional* and *non functional* specifications. Functional specification consists in defining what has to be done. Mainly, the

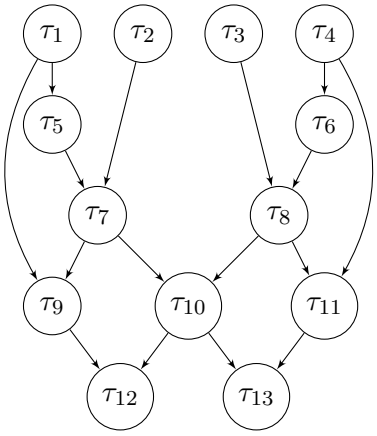


Figure 2.12: Example of a task dependence graph.

different functions of the application and the dependence between them are specified. Non functional specification consists in defining how the functions have to be performed. It provides a description of the hardware architecture, its different components and how they are interconnected. It specifies also allocation constraints if there are any and the timing parameters of the different functions, such as their execution times and periods.

Having both the functional and non functional specifications, the *potential* and the *effective* parallelisms can be deduced. The potential parallelism is related to the functional specification. It is defined by the functions that are not dependent as they can potentially be executed in parallel, e.g. τ_2 and τ_3 in Figure 2.12. The effective parallelism is defined by the hardware architecture, i.e. how many processing elements (processors, cores, ...) are able to execute functions in parallel. If the effective parallelism is less or equal to the potential parallelism, the execution of the application is accelerated. If it is greater, the execution is accelerated also but, no matter how much the effective parallelism is increased, the speedup remains constant. This can be interpreted by Amdahl's law which describes how hardware parallelism limits the exhibition of software parallelism.

List Scheduling

Heuristics are usually used to solve parallel scheduling problems because these problems are NP-complete [25] and using exact algorithms results in exponentially increasing execution times. In particular, list scheduling heuristics have been successfully used in the context of offline scheduling. All list scheduling heuristics are based on the same idea. Tasks that are ready to be scheduled are kept in a list. A task becomes ready to be scheduled once all its predecessors have been scheduled. The heuristic assigns priorities to the tasks in the list and selects the task with the highest priority to schedule it. This process is repeated untill all the tasks have been scheduled. The way the priorities of tasks are computed differs from one list scheduling heuristic to another. In the following, we review list scheduling heuristics that are proposed in the literature for makespan minimization.

A well-known algorithm to minimize the makespan of a dependence graph with no

transitive arcs is Hu's algorithm [26]. It assigns a level to each task in the dependence graph as follows: All tasks that have no immediate successor are at level one. Then, for each of the other tasks, the level is equal to one plus the maximum level of its immediate successors. Hu's algorithm proceeds repeatedly by allocating each time the ready task (whose all immediate predecessors have already been allocated) which has the highest level among all ready tasks to the first available processor.

Coffman-Graham algorithm [27] performs the scheduling in two steps. First a task is labeled with a label which is a function of the labels of its immediate successors (the labeling algorithm is not detailed here). Tasks are then allocated following a highest label first policy.

Papadimitriou and Yannakakis [28] studied the problem of scheduling interval-ordered dependence graphs. In such a graph, two tasks are precedence-related if and only if they can be mapped to non-overlapping intervals on the real number line [29]. A task is assigned a priority based on the number of its successors. A list of the tasks is constructed in a descending order of their priorities and then the tasks are allocated in this order.

In [30], level-based algorithms for scheduling dependence graphs are presented. The proposed Highest Level First with Estimated Times algorithm labels the tasks of the dependence graph with levels where a level corresponds to the length of the longest path from the task to a sink task. It, then, allocates the tasks in a highest-level first fashion. Therefore, the level of a task represents its priority. Highest Level First with No Estimated Times algorithm works similarly but with the assumption that all tasks have unit computation costs. In [31] a similar algorithm is proposed with the improvement of breaking ties by selecting the task with the largest number of successors.

In [32], two algorithms are proposed: First, the Heavy Node First algorithm which is based on a local analysis of the tasks at each level. In this algorithm, a level of a task corresponds to the longest path from a source task to this task. It allocates the heaviest task first. The second algorithm, Weighted Length (WL), considers a global view of the dependence graph by taking into account the relationships among the nodes at different levels.

The authors of [33] proposed the Insertion Scheduling Heuristic (ISH). The main idea of ISH is to fill the *scheduling holes* which are the idle time slots that appear as the schedule is being constructed.

The Modified Critical Path (MCP) algorithm proposed in [34] uses the measure of how late can a task be delayed without increasing the makespan of the schedule. The MCP algorithm assigns priorities to tasks in an ascending order of their latest start dates.

The Earliest Start Time algorithm [35] computes at each step, for each task, the earliest start date and selects the task that has the smallest one to allocate it.

The Dynamic Level Scheduling (DLS) algorithm [36] assigns dynamic levels to tasks. The Dynamic Level (DL) of a task is equal to the difference between the *b-level* (longest path from the corresponding task to a sink task) of the task and its earliest start date. At each step, the algorithm computes the dynamic levels for the ready tasks on all processors. The task-processor pair that gives the largest DL is selected for scheduling.

In [37], Yang and Gerasoulis present the Dominant Sequence Clustering (DSC) algorithm that uses an attribute called the dominant sequence which is the critical path of the dependence graph.

A current trend in multiprocessor scheduling is to use meta-heuristics such as Genetic Algorithms (GA) [38–40].

2.2.5 Parallel Real-time Scheduling

Real-time scheduling concerns the scheduling of tasks in real-time systems. Real-time does not mean fast. Instead, it refers to systems that must be able to respond to external events within specified deadlines [41]. Real-time systems are typically found in the form of embedded systems that control physical processes. They represent the cyber part in a CPS. In general, real-time systems are computing systems that are characterized by timing constraints in addition to the functional requirements. A part of this thesis deals with HiL simulation which can be qualified as a real-time system because the simulated part has to meet predefined deadlines in order to ensure correct results. In order to implement real-time applications, first, *real-time tasks* are defined by characterizing the functions obtained from the functional specification by a number of timing parameters. A real-time task denoted τ_i is characterized by the following parameters (see Figure 2.13):

- Release time r_i^k: Typical real-time applications consist of a set of tasks that are executed repeatedly where each execution is called an occurrence. The time at which an occurrence becomes ready to be executed is called the activation or the release time. r_i^k is the release time of the k^{th} occurrence of the task τ_i;
- First release time: r_i^0, called also offset;
- Start time s_i^k: The time at which the k^{th} instance starts its execution ($s_i^k \geq r_i^k$);
- Execution time C_i: A real-time task has an execution time which cannot be considered to be fixed and may vary from one execution to another. Therefore, a real-time task is characterized by its Worst Case Execution Time (WCET);
- Finishing time f_i^k: The time at which the k^{th} occurrence finishes its execution;
- Response time R_i^k: The duration between the release time and the finishing time of the k^{th} occurrence: $R_i^k = f_i^k - r_i^k$;
- Absolute deadline d_i^k: The time by which the k^{th} occurrence must finish its execution;
- Relative deadline D_i: Starting from the release time, the duration within which the task has to finish its execution;
- Laxity $l_i^k(t)$: Difference between the absolute deadline and the time for which the task has been running: $l_i^k = d_i^k - (t + C_i(t))$.

In addition, real-time tasks are characterized by a parameter related to how consecutive occurrences of a task are activated. Three kinds of tasks can be distinguished:

- Periodic tasks: The occurrences of a given task are activated periodically with a known period. A periodic task is characterized by its period T_i;
- Sporadic tasks: The occurrences of a task are activated such that the minimum time between two successive activations is known. A sporadic task is characterized by T_i, its minimum arrival time;
- Aperiodic tasks: The minimum delay between two activations is not known.

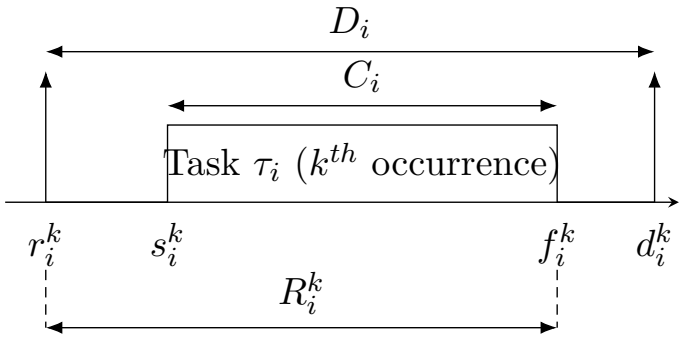


Figure 2.13: Parameters of a real-time task.

Real-time systems can be classified based on the impact of missing deadlines. Hard real-time systems are systems where all deadlines must be met. Violating this constraint leads to the failure of the system and may result in a great loss such as serious injuries, threatening human life, or damaging the surroundings. Soft real-time systems can tolerate some deadlines to be missed but the quality of the result degrades consequently. Firm real-time systems allow only a certain number of deadlines to be missed. We consider that HiL simulation falls within the category of firm real-time systems. In fact, in order to have correct HiL results, deadlines must be met. If a task misses its deadline, it produces invalid results and may cause the failure of the system but the consequences are not as catastrophic and harming as in the case of hard real-time systems.

Many different real-time scheduling algorithms have been proposed in the literature but they are all based on the same idea; tasks are assigned priorities and then scheduled in an order following their priorities. We distinguish between fixed priorities which do not change during the execution and dynamic priorities which are computed by the scheduler during the execution. Also, as in other kinds of scheduling problems, real-time scheduling algorithms can be classified into offline/online and preemptive/non preemptive algorithms.

The main goal of scheduling in real-time systems is to satisfy the different timing constraints of the tasks, i.e. release times, periodic activations, deadlines, etc. Schedulability tests can be used to check whether the tasks can be scheduled using a given scheduling algorithm in such a way to satisfy all the requirements. A schedulability test verifies if the utilization or the density of the processor, defined below, when it executes the set of tasks under test, is within a least upper bound. For a set of n independent periodic tasks, the utilization factor and density, when a preemptive scheduling algorithm is used, are respectively:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{2.6}$$

$$\Delta = \sum_{i=1}^{n} \frac{C_i}{D_i} \tag{2.7}$$

The most known real-time scheduling algorithms are the following:

- Fixed priorities
  - Rate Monotonic (RM) [42]: Tasks are assigned priorities inversely proportional to their periods. A set of tasks $\tau_i \in \mathcal{T} : D_i = T_i$ is schedulable by RM if $U \leq n(2^{\frac{1}{n}} - 1)$.
  - Deadline Monotonic (DM) [43]: Tasks are assigned priorities inversely proportional to their relative deadlines. A set of tasks $\tau_i \in \mathcal{T} : D_i \leq T_i$ is schedulable by DM if $\Delta \leq n(2^{\frac{1}{n}} - 1)$.
- Dynamic priorities
  - Earliest Deadline First (EDF) [42]: Priorities of tasks are inversely proportional to their absolute deadlines. The priority of a task is fixed for one occurrence but may change from one occurrence to another. EDF can schedule a set of tasks $\tau_i \in \mathcal{T} : D_i = T_i$ iff: $U \leq 1$.
  - Least Laxity First (LLF) [44]: Priorities of tasks are inversely proportional to their laxities. The priority may change for the same occurrence and from one occurrence to another. The schedulability test is the same as for EDF.

For multiprocessor real-time scheduling, there exist two principal approaches [45]:

- Global scheduling: Each task can be scheduled on any processor. the scheduler is responsible for migrating the tasks between the processors.
- Partitioned scheduling: The tasks are partitioned into groups, each of which is allocated to one processor. Each processor has a single-processor scheduler.

Global multiprocessor scheduling has significant overhead due to the migration cost. This is the reason why partitioned scheduling is usually used in hard real-time systems. Partitioning and allocating a set of tasks is equivalent to the *Bin Packing* problem which is NP-hard and heuristics are therefore used.

Assuming the tasks are sorted in a list and that processors are organized in a certain order, the most known heuristics that can be used to allocate a set of tasks to multiple processors are:

- Next Fit (NF): A task is tested on the available cores starting from the core the heuristic last allocated a task to. The task is allocated to the first found core that can schedule it. A task is schedulable on a given core if by allocating it to this core the condition ($U \leq 1$) is valid where U is the utilization of the core.

- First Fit (FF): Similar to NF but the search of the core that can schedule the task always starts from the first one.

- Best Fit (BF): Test the task on all cores and allocate it to the one that gives the minimum of U.

- Worst Fit (WF): Test the task on all cores and allocate it to the core that gives the maximum of U.

2.2.6 Parallel Execution

It is important to understand how the previously presented concepts of parallel computing are related. These concepts are involved in parallelization which refers to the process that takes as input a sequential code and achieves a parallel execution of the program. The first step of parallelization consists in detecting the potential parallelism of the program. Depending on the class of parallelism (e.g. task or data), a model, such as a dependence graph, is used in order to represent this potential parallelism. The targeted parallel architecture has to be modeled as well in order to accomplish the parallelism adaptation. Given the models of the potential parallelism and the effective parallelism, a schedule has to be found. In other words, the different parts of the program are allocated to the different components of the parallel architecture and their execution is ordered. Finally, based on the computed schedule, a parallel code is generated to be executed on the parallel architecture.

The AAA Methodology and SynDEx Software

The goal of the Algorithm-Architecture-Adequation (AAA) methodology [46] is to find out the best implementation of an algorithm specifying the functions that the application has to perform onto a multicomponent architecture, while satisfying real-time and embedding constraints. The AAA methodology is based on graph models to exhibit both the potential parallelism of the application algorithm and the available parallelism of the hardware architecture. Adequation means an efficient implementation. The implementation consists in distributing and scheduling the algorithm graph onto the multicomponent graph while satisfying real-time constraints. This is formalized in terms of graph transformations. Heuristics based on distributed real-time scheduling analyses taking into account timing characteristics attached to tasks (period, worst case execution time of computations and of inter-component communications), are used to automatically explore the possible implementations of a given application onto a given multicomponent hardware that satisfy real-time constraints, and to optimize the reaction time as well as resource allocation. The result of graph transformations is an optimized Synchronized Distributed Executive (SynDEx) dedicated to the application, automatically built from a library of architecture dependent executive primitives composing each executive kernel. (Figure 2.14).

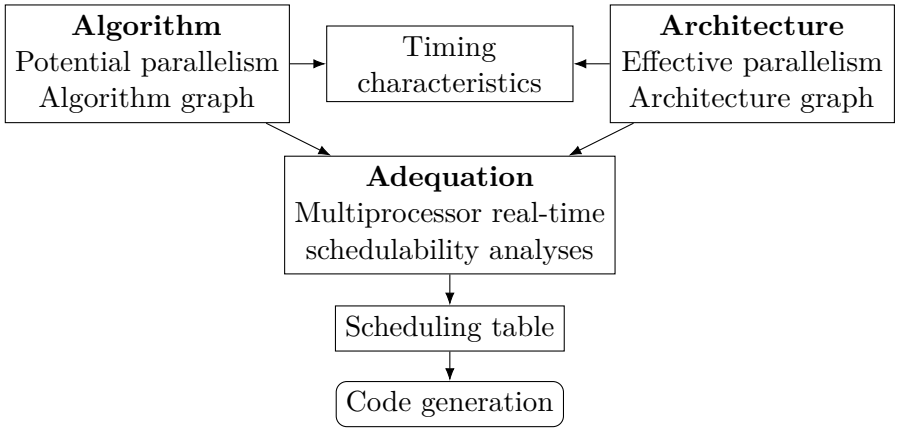


Figure 2.14: The Algorithm-Architecture-Adequation methodology.

2.3 Parallel Execution of Co-simulation

The more accurate is a simulation of a system, the more reliable is the assessment of its behavior. The numerical accuracy can be improved in different ways, for instance, by choosing a small integration step size. However, this means that more computations are performed and thus the computation load becomes large, decreasing the simulation performance. An important challenge faced by the developers and the users of simulation tools is to achieve a good simulation performance while maintaining an acceptable simulation accuracy.

The performance of a simulation can be significantly improved through parallel execution. In this scope, different approaches for the parallelization of simulation have been proposed in the literature. In this section we briefly review some of the approaches for the parallelization of simulation that are found in the literature. We present also some of the available simulation tools that support parallel simulation.

2.3.1 Approaches

In order to achieve simulation acceleration through parallel execution, different approaches are possible. Parallelization approaches can be classified into three categories based on the level at which the parallelization is introduced.

Parallelization across the Method

In this category, we find approaches that seek to parallelize the integration method. For instance, a multi-stage solver requires several computations within one integration step and it is possible to perform such computations in parallel. Such approach is studied in [47] by proposing a theoretical framework for the parallelization of Runge-Kutta methods. Another approach consists in parallelizing operations on vectors for ODE resolution like in the PVODE solver [48] implemented using MPI.

In [49], the authors propose a method for parallelization of modelica programs on CUDA-enabled GPUs. The proposed method relies on marking the functions to be

executed on the GPU by identifying patterns that are GPU suitable such as loops. These functions are then automatically translated into GPU code. In [50], ParModelica, an algorithmic extension of Modelica is proposed. This extension is based on OpenCL and allows stating the parallelism using special declarations in the code. An approach for automatic parallelization of equations on many-core platforms is proposed in [51]. This approach organizes the equations into a set of layers containing, each, a number of sections that can be executed in parallel and computes an offline schedule for their execution. In [52] an approach for the parallelization of multi-body simulation (simulation of systems composed of rigid and flexible bodies) on shared memory multiprocessors is proposed. This approach uses math-kernel libraries and OpenMP to parallelize matrix operations.

Parallelization across the Time

A simulation can be parallelized across the time steps. Examples of such approach are the Parareal algorithm [53], the Parallel Implicit Time-Integrator (PITA) [54], and the Parallel Full Approximation Scheme in Space and Time (PFASST) [55]. These methods divide the time domain into a two-level grid. A solution is evaluated in parallel over a fine time grid to improve a solution obtained sequentially over a coarse time grid.

Parallelization across the System

Finally, a simulation can be parallelized across the system, i.e. the equations used in the simulation are solved in parallel. A well known approach that parallelizes across the system is Waveform Relaxation (WR) [56], initially introduced for the simulation of large scale integrated circuits. The WR method breaks down the system into coupled subsystems of equations and computes the waveform, i.e. the solution, of each subsystem over a given time interval while fixing the waveforms of the other subsystems. The parallelization is made possible by computing the waveforms of several subsystems in parallel. The term waveform is used as the method was originally used to solve differential equations describing electrical circuits where signals are referred to as waveforms.

Transmission Line Modeling (TLM) [57] is a method that allows the decoupling and the parallelization of models by representing them using transmission line graphs such that decoupling points are chosen where variables change slowly because the models are considered to be connected by constants at these points. The approaches presented in [58, 59] are based on the TLM method.

An automatic parallelization approach based on dependence graph scheduling is presented in [60]. For this, the simulation code is analyzed at the expression level to build the dependence graph. A clustering algorithm is also proposed in this work to merge tasks. Finally an offline scheduling heuristic is applied on the dependence graph.

Co-simulation is naturally adapted to parallelization across the system. In fact, as shown in [61], splitting a model into several FMUs, by isolating discontinuities, may reduce the simulation time, even in the case of a sequential execution. In [5], the Refined CO-SIMulation (RCOSIM) approach is presented. It consists in using each FMU

information on input/output causality to build a graph, with an increased granularity and then exploiting the potential parallelism by using a heuristic to build an offline multi-core schedule in order to accelerate the execution.

The parallelization approach of the DACCOSIM tool is presented in [62]. Thanks to a graphical user interface, different FMUs of a co-simulation can be allocated to different multi-core computing nodes manually. If two dependent FMUs are allocated to different nodes, TCP connection is used for communication. Otherwise, interprocess communication is used. Each FMU is executed on its own thread on a distinct core. In addition, DACOSSIM is able to perform input extrapolation. This work has been later extended in [63]. The extension allows encapsulating DACCOSIM and the FMUs it controls in what is called a *Matryoshka* FMU. This allows to import DACCOSIM into other FMI compliant tools.

In [64], the authors propose a solution that combines two paralleization approaches: paralleization across the time and parallelization across the system. In particular, the proposed solution allows performing several stages of the Runge-Kutta solver in parallel within a single step. In addition, paralleization across the system is performed by parallelizing the computations involved in evaluating the equations of the system. A dependence graph of these computations is built and then scheduled on a multi-core processor.

2.3.2 Tools

More and more simulation tools are now endowed with parallel execution capabilities. However, it should be noted that some of these tools adopt parallelization approaches that do not target the numerical part of the simulation. For instance, the Parallel Computing Toolbox in MATLAB allows launching multiple Simulink simulations of the same model in parallel on a desktop multi-core computer or a cluster. These are separate independent simulations of the same model. This feature allows running multiple simulations under different configurations and conditions at the same time. It does not correspond to the focus of this thesis, i.e. the parallelization of the numerical computations of a simulation. Also, Simulink provides an execution mode known as Rapid Accelerator Mode which consists in creating a standalone executable of the model and the solver. Simulink runs in one process and this standalone executable runs in another process on a multi-core processor. Again, although this approach may improve the performance of the simulation, it does not lie within the scope of the thesis.

The Dymola tool enables automatic parallelization of equation resolution. The parallelization approach of Dymola is detailed in [51]. LMS Imagine.Lab Amesim allows launching multiple simulations in parallel, for example to run a model with different parameters. It has also the capability of partitioning models and executing them on multi-core processors. The TLM method, presented above, is integrated in the Hopsan tool. Finally, MBSim parallelizes matrix and vector operations as described in [52]. The co-simulation software xMOD is able to execute FMI co-simulations in parallel on multi-core architectures. It uses the RCOSIM approach [5] presented in Section 2.3.1.

3
Problem Statement

Contents

In the foregoing part of the thesis, we presented the background of the involved research work. We presented preliminary concepts and state of the art review regarding the different topics that are involved in this thesis. In this chapter, we detail the research problem of the thesis. We give an overview of the adopted methods and explain the research problems that this thesis attempts to solve.

3.1 Overview

This thesis addresses the problem of parallel execution of co-simulation. We are interested in co-simulations that are compliant with the FMI standard. Both FMI for Model Exchange and FMI for Co-Simulation are of interest in this thesis. In particular, we focus on closed-source FMI co-simulations, i.e. co-simulations for which functions are provided as executable binaries and the source code is not accessible.

From a hardware standpoint, we target shared-memory architectures, more specifically multi-core architectures such as the ones found in desktop and laptop computers.

The research carried out in this thesis can be divided into two parts. The first part deals with accelerated co-simulation, i.e. the goal of the paralleization of the co-simulation is to accelerate its execution on multi-core architectures. In the second part, we focus on co-simulation under real-time constraints. In particular, we are interested in

HiL (Hardware-in-the-Loop) co-simulation where a part of the co-simulation is replaced by its real counterpart that is physically available. The goal of this part is to satisfy the constraints for a real-time execution of the simulated part through parallelization on multi-core architectures.

3.2 The RCOSIM Approach

The contributions of this thesis constitute improvements to the RCOSIM approach briefly presented in section 2.3.1. The RCOSIM approach is based on offline multi-core scheduling. First, it transforms the co-simulation FMU graph into a dependence graph with finer granularity. This process is detailed in Section 4.1. A multi-core list scheduling heuristic is then used to compute a schedule for this dependence graph, i.e. an allocation of its vertices, which represent functions, to the cores and an order of execution for the vertices that are allocated to each core. Each run of this schedule corresponds to the execution of one simulation step, i.e. update of the inputs, the outputs, and the state. Therefore, the execution of the co-simulation consists in running this schedule repeatedly until the number of desired simulation steps is reached.

In this thesis, we chose to build on the RCOSIM approach in order to achieve parallelization of FMI co-simulations for both accelerated and real-time execution. We did not use known parallel programming libraries for the following specific reasons. It is clear that MPI is not suitable for our goal since we target shared memory architectures whereas MPI is used to program distributed memory architectures. The other option is to use OpenMP or similar libraries which are adapted to shared-memory architectures. However, OpenMP is efficient especially in the case of data parallelism (e.g. loop parallelism) which is not apparent in the co-simulations that we target. In fact, since we do not have access to the source code of the functions, we can not perform parallelization of the functions code, e.g. solver function, by using OpenMP pragmas. We only have information about the co-simulation at the function level, i.e. the functions can only be called but their code cannot be accessed. It should be noted that libraries such as OpenMP and Intel TBB offer task programming features which can be used to execute multiple functions in parallel. Note that the code of each function is not parallelized, but two or more functions can be executed in parallel using this solution. However, they rely on online scheduling which may introduce high overhead and thus decreases the performance. In addition, given that information about dependence between functions is available and the execution times can be measured, we assume that offline scheduling is more efficient to achieve our goal.

3.3 RCOSIM Limitations

Although RCOSIM resulted in interesting co-simulation speed-ups, it has some limitations that have to be considered in the parallelization problem in order to obtain better performances. We identify the following limitations of the RCOSIM approach:

1. So far, the multi-core scheduling heuristic uses empiric execution times of the

different functions. By using realistic execution times, the multi-core execution of the co-simulation should be improved.

2. Only mono-rate co-simulations, i.e. where all the FMUs have the same communication step size, can be handled by RCOSIM. The equations of each FMU are integrated using a specific integration step size whereas its inputs and outputs are updated according to a specific communication step size which is a multiple of the integration step size. This is due to the lack of information about the communication step sizes in the dependence graph constructed by RCOSIM.

3. The FMI standard does not presently require that functions of the same FMU have to be thread-safe, i.e. they cannot be executed simultaneously as they may share a resource (e.g. variable) that might be corrupted if two operations try to use it at the same time. In other words, such functions have to be executed in strictly disjoint time intervals. This is tackled in RCOSIM by modifying the multi-core scheduling heuristic to always allocate the functions of a same FMU to the same core. Consequently, the search space of the scheduling heuristic is reduced, i.e. for a given function, if there is another function of the same FMU that has already been allocated to a specific core, it is allocated to this same core without the need to test it on the other cores. This restriction on the allocation possibilities limits the exploitation of the potential parallelism.

4. The multi-core scheduling heuristic used in RCOSIM is not real-time. In fact, RCOSIM targets only accelerated co-simulation and cannot be used for real-time co-simulation.

5. The dependence graph constructed by RCOSIM does not contain information about real-time constraints. Such information is necessary in order to be able to apply a real-time scheduling algorithm. For instance, it is needed to know if the execution of a given function has to be finished before a specific deadline.

3.4 Open Research Issues and Thesis Objectives

We aim in this thesis at proposing solutions to overcome the limitations of the RCOSIM approach presented in the previous section. Below, we detail the research questions that correspond to these limitations.

Multi-rate FMU Co-simulation

Most industrial applications are multi-rate, i.e. the different FMUs of the co-simulation are executed according to different communication step sizes. In order to parallelize a multi-rate co-simulation, it is needed to incorporate this information in the dependence graph model. Depending on how this information is added, the multi-core scheduling heuristic used in RCOSIM may need to be modified. Therefore, the research question

related to this point is how should the dependence graph model be extended in order to allow RCOSIM to handle multi-rate co-simulation and what is the impact of such extension on the multi-core scheduling problem?

Mutual Exclusion Constraints in FMU Co-simulation

The non thread safe implementation of FMUs implies that at any instant during the execution of the co-simulation, one and only one function of the same FMU can be executed. Consequently, if the scheduling heuristic allocates two or more functions belonging to the same FMU to different cores, a mechanism that ensures these operations are executed in strictly different time intervals must be set up. We address the following questions: How can such mutual exclusion constraints be satisfied while optimizing the exploitation of the potential parallelism? Should the dependence graph model be extended or should these constraints be handled by the multi-core scheduling heuristic?

Specification of Real-time Constraints for FMU Co-simulation

In a co-simulation under real-time constraints, and more specifically HiL co-simulation, the physically available part periodically exchanges data with the simulated part, according to specific periods. The simulated part has to deliver data to the physically available components following these periods. Therefore, their execution has to be done within specific deadlines. Otherwise, the co-simulation fails. Given the periods of data exchange, we seek to define what are the constraints that have to be respected by the different FMUs in order to satisfy the real-time constraints. In other words, how should the executions of the different simulated FMUs be bounded so as to be able to deliver data in time to the real components at every period?

Multi-core real-time scheduling for FMU Co-simulation

Based on the defined real-time constraints, a multi-core scheduling algorithm is needed in order to allocate the functions of the simulated FMUs on the multi-core architecture in such a way to respect the real-time constraints. We are interested also in defining how the schedulability of a given co-simulation under real-time constraints on a given multi-core architecture can be tested?

4

Dependence Graph Model for FMU Co-simulation

Contents

This chapter describes a dependence graph model that we propose for representing an FMU co-simulation. The different phases for building such model are explained including the initial construction of the dependence graph, transformations that it undergoes in order to represent multi-rate co-simulation and mutual exclusion constraints, and finally rules for characterizing the graph with real-time parameters.

4.1 Dependence Graph of an FMU Co-simulation

Automatic parallelization of computer programs embodies the adaptation of the potential parallelism inherent in the program to the effective parallelism that is provided by the hardware. Because computer programs are usually complex (multiple functions, nested function calls, control flow jumps, etc.), this process of adaptation requires the use of a model for abstracting the program to be parallelized. The aim of using such model is to identify which parts of the program can be executed in parallel by expressing some features of the program such as data dependence between different parts. Dependence graphs are commonly used for this purpose (see Section 2.2.4). A dependence graph, denoted $G(V, A)$, where V is the set of vertices and A is the set if arcs, defines the partial order to be respected when executing a set of tasks. This partial order describes the potential parallelism of the program, i.e. vertices that are not in precedence relation A which is asymmetric and transitive.

The co-simulation of FMUs lends itself to the dependence graph representation as shown hereafter. According to the FMI standard, the code of an FMU can be exported in the form of source code or as precompiled binaries. However, most FMU providers tend to adopt the latter option for proprietary reasons. We are, thus, interested in this case. The method for automatic parallelization of FMU co-simulation that we propose in this thesis is based on representing the co-simulation by a dependence graph. We present in the rest of this section how this graph is constructed and a set of attributes that characterize it. The graph construction and characterization method is part of the RCOSIM approach as presented in [5].

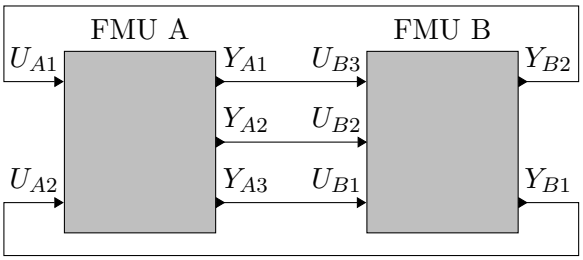
4.1.1 Construction of the Dependence Graph of an FMU Co-Simulation

The entry point for the construction of a dependence graph of an FMU co-simulation is a user-specified set of interconnected FMUs as depicted in Figure 4.1a. At this stage, we consider only co-simulations where all FMUs are assigned identical communication step sizes (this restriction is relaxed in Section 4.2). We refer to the graph which represents such co-simulation as *mono-rate* graph. The execution of each FMU is seen as computing a set of inputs, a set of outputs, and the state of the FMU. A computation of an input, output, or the state is performed by FMU C function calls. An input (resp. output) is computed by calling the *fmiSet* (resp. *fmiGet*) function and the state is computed by calling *SetTime, GetDerivatives, SetContinuousStates, etc.*, functions in the case of FMI for Model Exchange or the *DoStep* function in the case of FMI for Co-Simulation. Thanks to FMI, it is additionally possible to access information about the internal structure of a model encapsulated in an FMU. In particular, as shown in Figure 4.1b, FMI allows the identification of Direct Feedthrough (e.g. Y_{B1}) and Non Direct Feedthrough (e.g. Y_{A1}) outputs of an FMU and other information depending on the version of the standard:

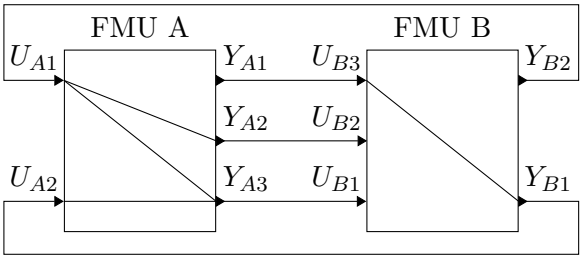
- FMI 1.0: Dependence between inputs and outputs is given. The computation of the state at a given time step t_k is considered necessary for the computation of

every output at the same time step t_k. It is considered that the computation of the state at a simulation step t_{k+1} requires the computation of each of the inputs at the simulation step t_k.

- FMI 2.0: In addition to the information provided in FMI 1.0, more information is given about data dependence. It is specified which output at a given time step depends on the state computation at the same step. Also, it is specified which input at time step t_k needs to be computed before the computation of the state at the step t_{k+1}.



(a) Inter-FMU dependence specified by the user



(b) Intra-FMU dependence provided by FMI

Figure 4.1: An example of inter and intra-FMU dependence of two FMUs connected by the user

The information provided by FMI on input-output dependence allows transforming the FMU graph into a graph with an increased granularity. For each FMU, the inputs, outputs, and state are transformed into operations. An input, output, or state operation is defined as the set of FMU function calls that are used to compute the corresponding input, output, or state respectively. The co-simulation is described by a dependence graph $G(V, A)$, called the operation graph, where each vertex $o_i \in V : 0 \leq i < n$ represents one operation, each arc $(o_i, o_j) \in A : 0 \leq i, j < n$ represents a precedence relation between operations o_i and o_j, and $n = |V|$ is the size of the operation graph. The operation graph is built by exploring the relations between the FMUs and between the operations of the same FMU. A vertex is created for each operation and arcs are then added between vertices if a precedence dependence exists between the corresponding operations. If FMI 1.0, which does not give information about the dependence between the state computation and the input and output variables computations, is used, we must add arcs between all input

operations and the state operation of the same FMU. Furthermore, arcs connect all output operations and the state operation of the same FMU because the computation at the time step t_k of an output must be performed with the same value of the state (computed at simulation step t_k) as for all the outputs belonging to the same FMU. An execution of the obtained graph corresponds to one simulation step. Therefore, running the co-simulation consists in repeatedly executing the graph until the desired number of steps is reached. A new execution of the graph cannot be started unless the previous one was totally finished. The operation graph corresponding to the FMUs of Figure 4.1 is shown in Figure 4.2.

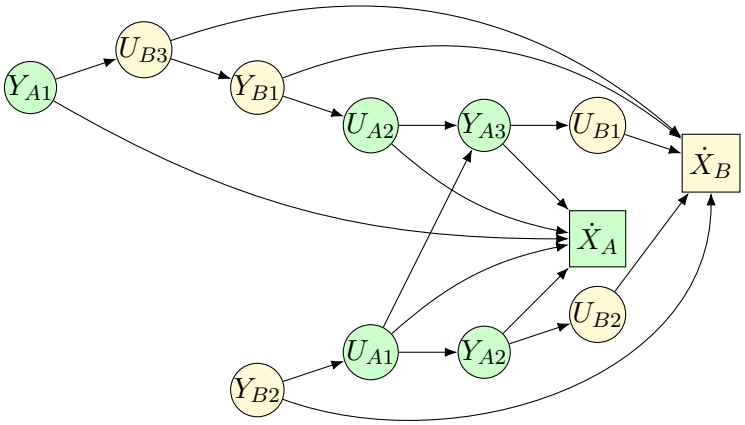


Figure 4.2: Operation graph obtained from the FMUs of Figure 4.1

4.1.2 Dependence Graph Attributes

The operation graph is used as input to a scheduling algorithm. In addition to the partial order defined by the graph, the scheduling algorithm uses a number of attributes to compute an efficient schedule of the operation graph. Many list scheduling algorithms use attributes that are computed by the Critical Path Method [65]. Below, we define a set of attributes and notations to characterize the operation graph.

The notation $tpe(o_i)$ is used to refer the type of the operation o_i, i.e. $tpe(o_i) \in \{update_{input}, update_{output}, update_{state}\}$, and $f_m(o_i)$ denotes the FMU to which the operation o_i belongs. Operation o_j is a predecessor of operation o_i if there is an arc from o_j to o_i, i.e. $(o_j, o_i) \in A$. We denote the set of predecessors of o_i by $pred(o_i)$. Operation o_j is an ancestor of operation o_i if there is a path in G from o_j to o_i. The set of ancestors of o_i is denoted by $ance(o_i)$. Operation o_j is a successor of operation o_i if there is an arc from o_i to o_j, i.e. $(o_i, o_j) \in A$. We denote the set of successors of o_i by $succ(o_i)$. Operation o_j is a descendant of operation o_i if there is a path in G from o_i to o_j. The set of descendants of o_i is denoted by $desc(o_i)$. A profiling phase allows measuring the execution time of each operation $o_i \in V$, denoted $C(o_i)$. For each operation, the average execution time of multiple co-simulation runs is used. When co-simulation under real-time constraints is aimed, Worst Case Execution Times (WCET) are used instead.

An operation o_i is characterized by its communication step $H(o_i)$ which is equal to the communication step assigned to the FMU $f_m(o_i)$. The earliest start time from start denoted $S(o_i)$ and the earliest end time form start denoted $E(o_i)$ are defined by equations 4.1 and 4.2 respectively. $S(o_i)$ is the earliest time at which the operation o_i can start its execution. $S(o_i)$ is subject to constraints imposed by precedence relations. The earliest time the operation o_i can finish its execution is $E(o_i)$.

$$S(o_i) = \begin{cases} 0, & \text{if } pred(o_i) = \emptyset. \\ \max_{o_j \in pred(o_i)}(E(o_j)), & \text{otherwise.} \end{cases} \tag{4.1}$$

$$E(o_i) = S(o_i) + C(o_i) \tag{4.2}$$

The latest end time from end denoted $\overline{E}(o_i)$ and the latest start time from end denoted $\overline{S}(o_i)$ are defined by equations 4.3 and 4.4 respectively.

$$\overline{E}(o_i) = \begin{cases} 0, & \text{if } succ(o_i) = \emptyset. \\ \max_{o_j \in succ(o_i)}(\overline{S}(o_j)), & \text{otherwise.} \end{cases} \tag{4.3}$$

$$\overline{S}(o_i) = \overline{E}(o_i) + C(o_i) \tag{4.4}$$

The critical path of the graph is the longest path in the graph. The length of a path is computed by accumulating the execution times of the operations that belong to it. The length of the critical path of the operation graph denoted by CP is defined by equation 4.5. The critical path is a very important characteristic of the operation graph. It defines a lower bound on the execution time of the graph, i.e. in the best case, the time needed to execute the whole graph is equal to the length of the critical path.

$$CP = \max_{o_i \in V}(E(o_i)) \tag{4.5}$$

The flexibility $F(o_i)$ is defined by equation 4.6. It expresses the length of a time interval within which operation o_i can be executed without increasing the total execution time of the graph.

$$F(o_i) = CP - S(o_i) - C(o_i) - \overline{E}(o_i) \tag{4.6}$$

4.2 Dependence Graph of a Multi-rate FMU Co-simulation

The operation graph model presented in the previous section allows modeling only mono-rate FMU co-simulations. For some applications this model is sufficient to be used for multi-core scheduling. However, many industrial co-simulation applications feature behaviors that cannot be captured by this model. In particular, many industrial applications involve FMUs that are executed according to different communication step sizes. This is especially true when different FMUs of a co-simulation are provided by different parties. It is very common that an FMU provider designs the FMU in such a

way that its proper functioning depends on using specific communication step sizes. It is, therefore, highly unrecommended, and in some cases even impossible, to change the communication step size of the FMU. In other cases, even if it is possible and acceptable to change the communication step size of a given FMU, better performance and/or accuracy could be obtained when using a specific communication step size. As a consequence, our operation graph model has to be extended in order to accommodate multi-rate data exchange between operations. To fulfill this, we propose in this section, a method to transform the initial operation graph $G(V, A)$ into a new operation graph.

4.2.1 Repeatable Pattern of a Multi-rate Dependence Graph

Consider an operation graph that is constructed as described in the previous section from a multi-rate co-simulation, i.e. a co-simulation where some FMUs are assigned different communication step sizes. Such graph is referred to as a *multi-rate* operation graph. In the case of mono-rate co-simulation, the intial operation graph constructed from the co-simulation is sufficient because it describes a repeatable pattern for the execution of the co-simulation. As such, a schedule of this graph can be run repeatedly in order to run the co-simulation. In such schedule, each operation appears exactly once. In the case of multi-rate co-simulation, the intial operation graph does not describe a pattern that can be repeatedly executed. One way for making such operation graph suitable for multi-core scheduling is to transform it into a mono-rate graph. In a schedule of the resulting mono-rate graph , each operation appears exactly once. The aim of this transformation is to obtain a repeatable pattern of the operation graph while ensuring that each operation is executed according to the communication step size assigned to its respective FMU, and also maintaining a correct data exchange between the different FMUs, whether they are assigned different or identical communication step sizes. Similar algorithms have been used in the real-time scheduling literature to deal with multi-rate scheduling problems [66, 67].

In the case of mono-rate co-simulation, the length of the repeatable pattern is equal to the unique communication step size. In order to perform a transformation of a multi-rate operation graph, we need, first, to specify the length of the repeatable pattern. We define the notion of *hyperstep* (HS) in Definition 4.2.1.

Definition 4.2.1. In the context of multi-rate FMU co-simulation, the hyperstep is the least common multiple (lcm) of the communication step sizes of all the operations: $HS = lcm(H(o_1), H(o_2), \ldots, H(o_n))$.

The hyperstep is the smallest interval of time steps for describing an infinitely repeatable pattern of all the operations. The transformation consists, first of all, in repeating each operation o_i, $r(o_i)$ times where $r(o_i)$ is called the repetition factor of o_i and $r(o_i) = \frac{HS}{H(o_i)}$. Each repetition of the operation o_i is called an occurrence of o_i and corresponds to the execution of o_i at a certain time step. We use a superscript to denote the number of each occurrence, e.g. o_i^p denotes the p^{th} occurrence of o_i. Operations belonging to the same FMU have the same repetition factor since they are all executed according to the communication step size assigned to the FMU that they belong to.

Therefore, we define the repetition factor of an FMU to be equal to the repetition factor of its operations. Figure 4.3 shows an example of the execution of two operations o_i and o_j such that $H(o_i) = 2$ and $H(o_j) = 3$. In this example, the hyperstep is $HS = lcm(2, 3) = 6$ and the repetitions factors are $r(o_i) = \frac{6}{2} = 3$ and $r(o_j) = \frac{6}{3} = 2$. It can be seen that the execution pattern over an interval of length 6 is repeated.

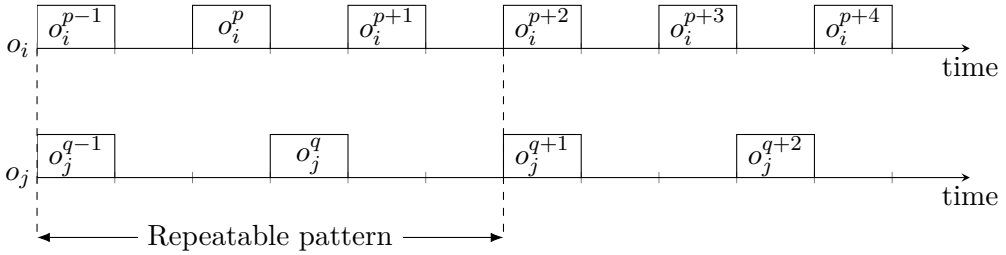


Figure 4.3: A basic example of a repeatable pattern of a multi-rate co-simulation

4.2.2 Multi-rate Transformation Rules

In order to complete the multi-rate transformation, we need to specify dependence between the occurrences that are added by creating arcs. Arcs are added between operations following the rules presented hereafter. Consider two operations $o_i, o_j \in V$ connected by an arc $(o_i, o_j) \in A$ in the initial operation graph. Adding an arc (o_i^p, o_j^q) to A, depends on the time steps at which o_i^p and o_j^q are executed. Let t_{kp} and t_{kq} be the time steps at which o_i^p and o_j^q are executed respectively. An arc is added between occurrences o_i^p and o_j^q if the following conditions are satisfied:

1. o_i^p is executed at a time step that precedes or matches the time step at which o_i^p is executed, i.e. $t_{kp} \leq t_{kq}$.

2. o_i^p is the latest occurrence of o_i that satisfies the first condition, i.e. $p = \max\{p' : 0 \leq p' < r(o_i)$ *and* $t_{kp'} \leq t_{kq}\}$ where $t_{kp'}$ denotes the times step at which occurrence $o_i^{p'}$ is executed.

In the case where $H(o_i) = H(o_j)$, and therefore $r(o_i) = r(o_j)$, occurrences o_i^p and o_j^q which correspond to the same number, i.e. $p = q$, are connected by an arc. On the other hand, if $H(o_i) \neq H(o_j)$, we distinguish between two types of dependence: we call the arc $(o_i, o_j) \in A$ a *slow to fast* (resp. *fast to slow*) dependence if $H(o_i) > H(o_j)$ (resp. $H(o_i) < H(o_j)$). For a slow to fast dependence $(o_i, o_j) \in A$, one occurrence of o_i is executed while several occurrences of o_j are executed. In this case, arcs are added between each occurrence $o_i^p : p \in \{0, 1, \ldots, r(o_i) - 1\}$, and the occurrence o_j^q such that:

$$q = \left\lceil p \times \frac{H(o_i)}{H(o_j)} \right\rceil \tag{4.7}$$

Let operations o_i and o_j shown in Figure 4.3 be connected by an arc (o_j, o_i) in the initial operation graph. For $q = 1$, the added arcs between the occurrences of o_i and o_j can be represented on the time chart of the execution as shown in Figure 4.4.

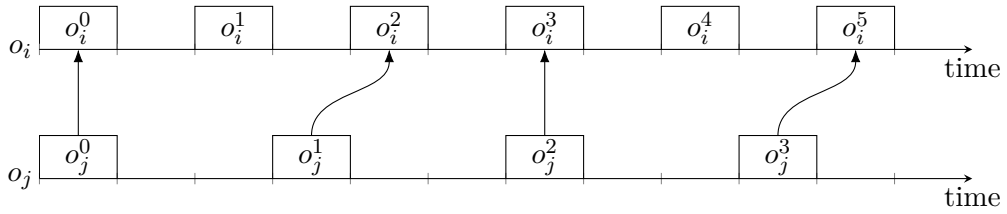


Figure 4.4: Slow to fast dependence

We recall that for a slow to fast dependence, the master algorithm can preform extrapolation of the inputs of the receiving FMU.

For a fast to slow dependence $(o_i, o_j) \in A$, arcs are added between each occurrence o_i^p, and the occurrence $o_j^q : q \in \{0, 1, \ldots, r(o_j) - 1\}$ such that:

$$p = \left\lfloor q \times \frac{H(o_j)}{H(o_i)} \right\rfloor \tag{4.8}$$

Let operations o_i and o_j shown in Figure 4.3 be connected by an arc (o_i, o_j) in the initial operation graph. For $p = 1$, the added arcs between the occurrences of o_i and o_j can be represented on the time chart of the execution as shown in Figure 4.5.

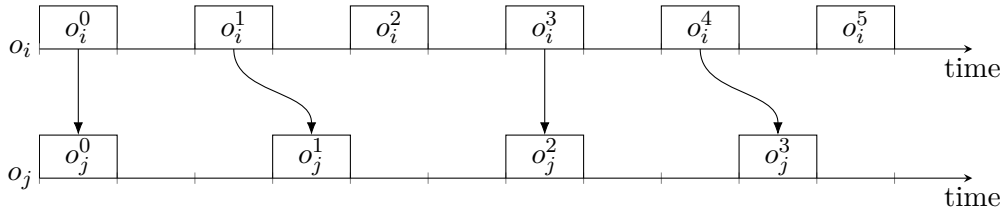


Figure 4.5: Fast to slow dependence

Arcs are added also between the occurrences of the same operation, i.e. $(o_i^p, o_i^{p'})$ where $p \in \{0, 1, \ldots, r(o_i) - 2\}$ and $p' = p + 1$. Finally, for each FMU, arcs are added between the p^{th} occurrence of the state operation, where $p \in \{0, 1, \ldots, r(o_i) - 2\}$, and the $(p + 1)^{th}$ occurrences of the input and output operations.

4.2.3 Multi-rate Transformation Algorithm

The multi-rate graph transformation is detailed in Algorithm 1. The algorithm traverses all the graph by applying the aforementioned rules in order to transform the graph and finally stops when all the nodes and the edges have been visited.

Figure 4.6 shows the graph obtained by applying the multi-rate transformation algorithm on the graph of Figure 4.2. In this example $H_B = 2 \times H_A$, where H_A and H_B are the communication steps of FMUs A and B respectively.

Algorithm 1: Multi-rate graph transformation algorithm

Input : Initial operation graph $G(V, A)$;
Output : Transformed operation graph $G(V, A)$;
foreach $o_i \in V$ **do**
  Compute the repetition factor of o_i: $r(o_i) \leftarrow \frac{HS}{H(o_i)}$;
  Repeat the operation o_i: $V \leftarrow V \cup \{o_i^p\}, p \in \{1, \ldots, r(o_i) - 1\}$;
foreach $(o_i, o_j) \in A$ **do**
  if $H(o_i) > H(o_j)$ **then**
    for $p \leftarrow 0$ **to** $r(o_i) - 1$ **do**
      Compute $q = \left\lceil p \times \frac{H(o_i)}{H(o_j)} \right\rceil$;
      Add the arc (o_i^p, o_j^q) to the graph: $A \leftarrow A \cup \{(o_i^p, o_j^q)\}$;
  else if $H(o_i) < H(o_j)$ **then**
    for $q \leftarrow 0$ **to** $r(o_j) - 1$ **do**
      Compute $p = \left\lfloor q \times \frac{H(o_i)}{H(o_j)} \right\rfloor$;
      Add the arc (o_i^p, o_j^q) to the graph: $A \leftarrow A \cup \{(o_i^p, o_j^q)\}$;
  else
    for $p \leftarrow 0$ **to** $r(o_i) - 1$ **do**
      Add the arc (o_i^p, o_j^p) to the graph: $A \leftarrow A \cup \{(o_i^p, o_j^p)\}$;
foreach $o_i \in V$ **do**
  for $p \leftarrow 0$ **to** $r(o_i) - 2$ **do**
    Add an arc between successive occurrences of o_i: $A \leftarrow A \cup \{(o_i^p, o_i^{p+1})\}$;
foreach $o_i \in V : tpe(o_i) = state$ **do**
  for $p \leftarrow 0$ **to** $r(o_i) - 2$ **do**
    foreach $o_j \in V : f_m(o_j) = f_m(o_i)$ *and* $tpe(o_i) \in \{input, output\}$ **do**
      Add the arc (o_i^p, o_j^{p+1}) to the graph: $A \leftarrow A \cup \{(o_i^p, o_j^{p+1})\}$;

Without any loss of generality, the superscript which denotes the number of the occurrence of an operation is not used in the remainder of the thesis for the sake of simplicity, unless needed to specify the occurrence. Each occurrence of an operation o_i^p in the graph $G(V, A)$ becomes an operation that is referred to using the notation o_j.

4.3 Dependence Graph with Mutual Exclusion Constraints

The FMI standard states that *"FMI functions of one instance don't need to be thread safe"*. Therefore, an FMU does not implement any service to support concurrent access to its functions from multiple threads, and it is up to the executing environment to ensure the calling sequences of the FMU functions are respected as specified in the FMI standard. These restrictions introduce mutual exclusion constraints on the operations of the same FMU. We propose in this section an offline method for handling these

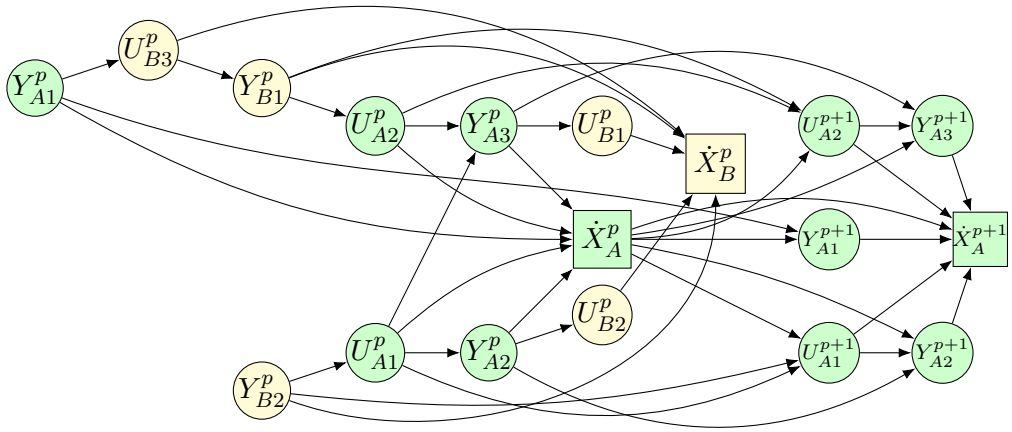


Figure 4.6: Graph obtained by applying the multi-rate transformation algorithm on the graph of Figure 4.2

constraints with low synchronization overhead.

4.3.1 Motivation

In order to study the impact of mutual exclusion constraints, we have evaluated the performance obtained using two mutual exclusion strategies. In the first one, a dedicated lock (system object that guarantees mutual exclusion) is used for each FMU. Every time an FMU function call is made at runtime, the associated lock has to be acquired before the execution of the function code can be started. This mechanism allows the synchronization of threads that execute different functions of the same FMU sharing same resources. Thanks to using the locks, each operation can be allocated to any core. We refer to such allocation as *unconstrained allocation*. The second solution is explained in [5] and consists in allocating the operations of a same FMU to the same core. We refer to such allocation as *constrained allocation*. The scheduling heuristic that was used in these tests is presented in Chapter 5. The theoretical speed-up was estimated by computing the makespan of the operation graph. The makepsan is the total time needed to compute the whole graph. Results are given in Figure 4.7. It shows that the expected speed-up using constrained allocation is less than the one using unconstrained allocation, when the number of cores is less than five, but similar when five cores or more are available. When using less than five cores, the large number of output operations can be efficiently allocated only if the unconstrained allocation is used: the speed-up difference between the constrained and the unconstrained allocation cases is due to this restriction on the allocation. Five is the minimal number of cores for enabling the execution of each state operation on a different core. Due to the predominant execution times of the state operations, their impact on the speed-up overrides the optimization of the allocation of the other operations. This explains why the speed-up difference between the unconstrained and the constrained allocation cases becomes very small using five cores or more.

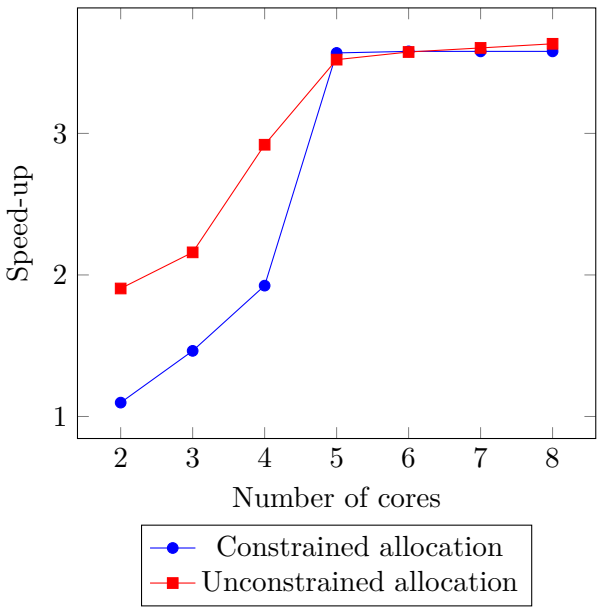


Figure 4.7: Theoretical speed-up.

We implemented and tested both mutual exclusion strategies in order to compare their runtime performance. Tests were performed on the industrial use case described in Chapter 6. Execution times measurements were performed by getting the system time stamp at the beginning of the execution and after 30 seconds of the simulated time. As previously mentioned, we compared the speed-up by dividing the single-core co-simulation execution time by the co-simulation execution time on a fixed number of cores. Figure 4.8 sums up the results. It shows the impact of mutex synchronization overhead on the speed-up. Whatever the number of the available cores, the speed-up remains close to 1.3. On the contrary, the implementation of the constrained allocation results in a runtime speed-up that is similar to the theoretical speedup in terms of speed-up improvement when the number of cores is increased until reaching five. Nevertheless, the maximum measured speed-up (2.4) remains smaller than the theoretical one (3.5). In fact, the theoretical speed-up computation considers the makespan ratio without any estimation of the runtime overhead which certainly has an important impact on the speed-up.

The restrictions introduced by employing the tested mutual exclusion techniques makes it highly desirable to find an alternative solution that could satisfy the mutual exclusion constraints while: i) leaving as much flexibility as possible for allocating the operations to the cores and; ii) introducing lower synchronization overhead. In the rest of this section, we suggest a method for offline handling of mutual exclusion constraints. The proposed method is based on modeling the mutual exclusion constraints in the operation graph of the co-simulation.

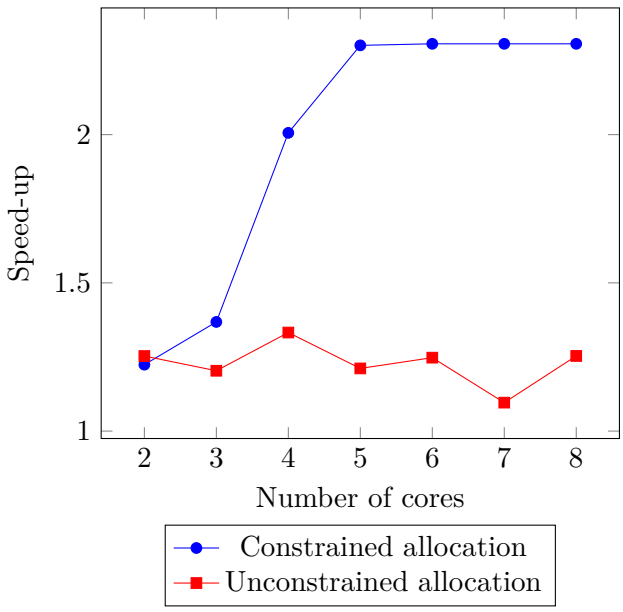


Figure 4.8: Runtime speed-up.

4.3.2 Acyclic Orientation of Mixed Graphs

The operation graph model can be extended in order to represent scheduling problems that involve precedence constraints and also mutual exclusion constraints. This is commonly done using *mixed graphs*. A mixed graph $G(V, A, D)$ is a graph which contains a set A of directed arcs denoted $(o_i, o_j) : 0 \leq i, j < n$ and a set D of undirected edges denoted $[o_i, o_j] : 0 \leq i, j < n$. In the scheduling literature, these graphs are known also as *disjunctive graphs* [68]. In addition to the precedence constraints represented by arcs as described in Section 4.1, mutual exclusion relations are represented by edges in a mixed graph such that:

- *Precedence constraints:* $\forall (o_i, o_j) \in A, o_i$ must finish its execution before o_j can start its execution.
- *Mutual exclusion constraints:* $\forall [o_i, o_j] \in D, o_i$ and o_j must be executed in strictly disjoint time intervals.

Operations belonging to the same FMU can be executed in either order but not in parallel. Undirected edges can be added between these operations in order to represent such mutual exclusion constraints. This transforms the operation graph into a mixed graph. In order to compute a schedule for such mixed graph, an execution order has to be defined for each pair of operations connected by an undirected edge which is interpreted by assigning a direction to this edge. Cycles must not be introduced in the graph while assigning directions to edges, otherwise, the scheduling problem becomes infeasible. Once all edges have been assigned directions, the result is a new operation graph which is a DAG. Since the final goal is to accelerate the execution of the co-simulation which comes down to minimizing the makespan of the operation graph, the acyclic orientation of the

mixed graph has to minimize the length of the critical path of the resulting DAG. We recall that the length of the critical path of the operation graph represents a lower bound on the makespan of the graph. This problem is known as *acyclic orientation* [69]. We denote the acyclic orientation as a function $\phi : \ [o_i, o_j] \in D \rightarrow \{(o_i, o_j), (o_j, o_i)\}$

The acyclic orientation problem is closely related to vertex coloring of a graph [70]. In its general form, i.e. when all edges of the graph are undirected, vertex coloring is a function $\alpha : V \rightarrow \{1, 2, \ldots, k\}$ which labels the vertices of the graph with integers, called colors, such that the inequality 4.9 holds.

$$\forall \ [o_i, o_j] \in D, \ \alpha(o_i) \neq \alpha(o_j) \tag{4.9}$$

The acyclic orientation of the graph can then be obtained by assigning a direction to every edge such that the color of the corresponding tail vertex is smaller than the color of the corresponding head vertex. A graph coloring with k colors is referred to as *k-coloring*. In its general form, vertex coloring aims at finding a *minimum vertex coloring*, i.e. minimizing k the number of the used colors. The minimum number of colors required to color an undirected graph G is called the chromatic number and is denoted $\chi(G)$. The Gallai–Hasse–Roy–Vitaver theorem [71–74] links the length of the longest path of the graph, obtained by the orientation which minimizes this length, to vertex coloring of the graph. It states that the length of the longest path of a directed graph is at least $\chi(G)$. Thus, a minimum vertex coloring leads to an acyclic orientation that minimizes the length of the critical path of the resulting graph. Computing the chromatic number of a graph is NP-complete [75].

The acyclic orientation of a mixed graph can be obtained via vertex coloring also. However, vertex coloring of a mixed graph has to take into account both arcs and edges of the graph. More precisely, a vertex coloring of a mixed graph is a function $\alpha : V \rightarrow \{1, 2, \ldots, k\}$ such that inequalities 4.9 and 4.10 hold.

$$\forall \ (o_i, o_j) \in A, \ \alpha(o_i) < \alpha(o_j) \tag{4.10}$$

A coloring of a mixed graph $G(V, A, D)$ exists only if it is cycle-free [76], i.e. the directed graph $G(V, A, \emptyset)$ does not contain any cycle. The problem of acyclic orientation of mixed graphs has been studied in the literature in [77–79]. Efficient algorithms have been proposed for the orientation of special types of mixed graphs. It has been shown that, in the general case, the problem is NP-Hard.

4.3.3 Problem Formulation

Let $G(V, A)$ be an operation graph of an FMU co-simulation constructed as described in Section 4.1. In order to represent mutual exclusion constraints between FMU operations, the initial operation graph $G(V, A)$ is transformed into a mixed graph by connecting each pair of mutually exclusive operations o_i, o_j by and edge $[o_i, o_j]$. The resulting mixed graph is denoted $G(V, A, D)$, where V is the set of operations, A is the set of arcs, and D is the set of edges. Once the mixed graph is constructed, directions have to

be assigned to its edges in order to define an order of execution for mutually exclusive operations. The precedence and mutual exclusion relations represented by the mixed graph $G(V, A, D)$ are given by expressions 4.11 and 4.12 respectively. If operations o_i and o_j are connected by an arc (o_i, o_j), the time interval $(S(o_i), E(o_i)]$ must precede the time interval $(S(o_j), E(o_j)]$. Otherwise, if operations o_i and o_j are connected by an edge $[o_i, o_j]$, time intervals $(S(o_i), E(o_i)]$ and $(S(o_j), E(o_j)]$ must be strictly disjoint.

$$\forall\ (o_i, o_j) \in A,\ E(o_i) \leq S(o_j) \tag{4.11}$$

$$\forall\ [o_i, o_j] \in D,\ (S(o_i), E(o_i)] \cap (S(o_j), E(o_j)] = \emptyset \tag{4.12}$$

The timing attributes of the operations in the mixed graph $G(V, A, D)$ are the same as in the initial graph $G(V, A)$ because the added set of edges $[o_i, o_j] \in D$ does not impact the computation of these attributes. The attributes of an operation o_i, connected by an edge with another operation, may change only when this edge is assigned a direction following the order of the execution intervals of o_i and o_j.

An edge $[o_i, o_j]$ is called a conflict edge if the intervals $(S(o_i), E(o_i)]$ and $(S(o_j), E(o_j)]$ in the graph $G(V, A)$ overlap. This can be written in the form of expression 4.13. If for a given edge $[o_i, o_j]$ either $E(o_i) \leq S(o_j)$ or $E(o_j) \leq S(o_i)$, there is no conflict and the edge can be assigned a direction.

$$E(o_i) > S(o_j) \text{ and } E(o_j) > S(o_i) \tag{4.13}$$

It should be noted that, for a given edge $[o_i, o_j]$, choosing either of the execution orders does not impact the numerical results of the co-simulation since these operations do not have data dependence. An order have to be defined only because we have to ensure mutual exclusion between them due to the non-thread-safe implementation of FMI. Following the definition given in the previous section, the corresponding vertex coloring is a function $\alpha : V \to \{1, 2, \ldots, k\}$ which is equivalent to mapping the operations $o_i \in V$ to the time intervals $[S(o_1), E(o_1)], [S(o_2), E(o_2)], \ldots, [S(o_n), E(o_n)]$.

The problem of acyclic orientation of the mixed graph $G(V, A, D)$ can be stated as an optimization problem as follows:

Input	Mixed graph $G(V, A, D)$
Output	DAG $G(V, A)$
Find	Coloring $\alpha : V \to \{1, 2, \ldots, k\}$
Minimize	Number of colors k
Subject to	Precedence constraints: $\forall\ (o_i, o_j) \in A,\ \alpha(o_i) < \alpha(o_j)$ Mutual exclusion constraints: $\forall\ [o_i, o_j] \in D,\ \alpha(o_i) \neq \alpha(o_j)$

4.3.4 Resolution using Linear Programming

Let $G(V, A, D)$ be a mixed graph constructed form the operation graph $G(V, A)$ as described in the previous sections to represent precedence and mutual exclusion constraints between operations of an FMU co-simulation. In the following, we present an Integer Linear Programming formulation for the problem of acyclic orientation of $G(V, A, D)$. The proposed formulation is based on the scheduling notation which gives a more compact set of constraints compared to a formulation that uses the vertex coloring notation.

Variables and Constants

Tables 4.1 and 4.2 summarize the variables and the constants that are used in the ILP formulation respectively.

Table 4.1: Variables used in the ILP formulation of the acyclic orientation problem

Variable	Type	Description
$S(o_i)$	Integer	Start time of operation o_i
$E(o_i)$	Integer	End time of operation o_i
b_{ij}	Binary	Orientation decision variable associated with edge $[o_i, o_j] \in D$
CP	Integer	Length of the critical path of the graph

Table 4.2: Constants used in the ILP formulation of acyclic orientation problem

Constant	Type	Decription
$C(o_i)$	Integer	Execution time of operation o_i
M	Integer	Large positive number

Constraints

The following set of constraints is used in the ILP formulation of the acyclic orientation problem:

- *Precedence constraints:* The start time of each operation is equal to the maximum of the end times of all its predecessors. Expression 4.14 captures this constraint. Note that expression 4.14 indicates that the start time of operation o_j is greater or equal to the end time of each predecessor o_i. This is sufficient to express $S(o_j) = \max_{o_i \in pred(o_j)}(E(o_i))$ since the formulated problem is a minimization problem.

$$\forall (o_i, o_j) \in A, S(o_j) \geq E(o_i) \tag{4.14}$$

- *Mutual exclusion constraints:* We define the binary variable b_{ij} which is associated with the direction that is assigned to edge $[o_i, o_j]$. b_{ij} is set to 1 if the edge $[o_i, o_j]$ is

assigned a direction from o_i to o_j, i.e. $\phi([o_i, o_j]) = (o_i, o_j)$ and to 0 otherwise. Note that b_{ij} is the complement of b_{ji}. For every pair of operations that are connected by and edge, we have to ensure that their time intervals are strictly disjoint, i.e. $\forall\ [o_i, o_j] \in D,\ (S(o_i), E(o_i)] \cap (S(o_j), E(o_j)] = \emptyset$. Expressions 4.15 and 4.16 capture this constraint where M is a large positive integer.

$$\forall [o_i, o_j] \in E, S(o_i) \geq E(o_j) - M \times (1 - b_{ij}) \tag{4.15}$$

$$\forall [o_i, o_j] \in E, S(o_j) \geq E(o_i) - M \times b_{ij} \tag{4.16}$$

- *Time intervals:* Expression 4.17 is used to compute the end time of each operation.

$$\forall o_i \in V, E(o_i) = S(o_i) + C(o_i) \tag{4.17}$$

- *Length of the critical path:* The critical path CP is equal to the maximum of the end times of all the operations as stated by expression 4.18.

$$\forall o_i \in V, P \geq E(o_i) \tag{4.18}$$

Objective

The objective of this linear program is to minimize the length of the critical path of the operation graph (expression 4.19).

$$min(CP) \tag{4.19}$$

While exact algorithms such as ILP give optimal results, they suffer form very long execution times that are not acceptable for the users. For many real world applications, ILP fails to produce the results within acceptable times. Heuristics are usually good alternatives. While the optimality of the solution cannot be guaranteed when using heuristics, they, in most cases, provide results of good quality, not too far form the optimal solution within acceptable execution times.

4.3.5 Acyclic Orientation Heuristic

We propose in this section a heuristic for the acyclic orientation of the mixed graph $G(V, A, D)$. A straightforward acyclic orientation can be obtained by sorting the operations in a non decreasing order of their start times $S(o_i)$ and assigning directions to edges following this order, i.e. $\forall [o_i, o_j] \in D, S(o_i) \leq S(o_j), \phi([o_i, o_j]) = (o_i, o_j)$. This is a fast greedy acyclic orientation, however it can be improved as we show hereafter.

Let s be the sum of the repetition factors of all the FMUs. The set of operations V can be represented as a union of mutually disjoint non empty subsets such that

every subset contains all operations that belong to the same FMU and that correspond to the same occurrence:

$$V = \bigcup_{k=1}^{s} V_k : \ \forall\ o_i^p, o_j^q \in V_k,\ k \in \{0, 1, \ldots, s\},\ f_m(o_i^p) = f_m(o_j^q) \text{ and } p = q \tag{4.20}$$

We know that edges in the set D exist only between operations that belong to the same FMU. Furthermore, for every edge $[o_i^p, o_j^q] \in D$, operations o_i^p and o_j^q correspond to the same occurrence, i.e. $p = q$. Although operations which belong to the same FMU and correspond to different occurrences are mutually exclusive, it is not needed to connect them by an edge because an execution order is already ensured for these operations by the way the operation graph is constructed. In other words, all the operations of an FMU, and which correspond to the same occurrence p have to finish their execution before the next occurrence $p + 1$ of any operation can start its execution. Similarly to the operation set, the edge set D can be subdivided into mutually disjoint non empty subsets:

$$D = \bigcup_{k=1}^{s} D_k,\ \forall\ [o_i^p, o_j^q] \in D_k,\ k \in \{0, 1, \ldots, s\},\ f_m(o_i^p) = f_m(o_j^q) \text{ and } p = q \tag{4.21}$$

In view of the above, we define the set of subgraphs which constitute the graph $G(V, \emptyset, D) = \bigcup_{k=1}^{s} G(V_k, D_k)$. Theorem 4.3.1 states the relationship between the acyclic orientations of the subgraphs $G(V_k, D_k)$ and the acyclic orientation of the mixed graph $G(V, A, D)$.

Theorem 4.3.1. *An acyclic orientation of the mixed graph $G(V, A, D)$ can be obtained by finding an acyclic orientation for every subgraph $G_k(V_k, D_k)$ following the non decreasing order of the start times of the operations as described previously.*

Proof. In order to prove this, we have to show that every edge in D is assigned a direction and that the resulting orientation does not lead to the creation of a cycle. We use a proof by contradiction to prove this statement. Since every edge $[o_i, o_j]$ belongs to one subset of edges D_k, finding acyclic orientations for all the subgraphs $G_k(V_k)$ leads to assigning a direction to every edge in D. The existence of a cycle in the resulting graph means that there exists at least an edge $[o_i, o_j]$, such that $S(o_i) > S(o_j)$, that has been transformed into the arc (o_i, o_j). However, this is not possible because the greedy acyclic orientation assigns directions to edges following a non-decreasing order of the start times of the operations which contradicts the previous assertion and thus proves Theorem 4.3.1. □

Consider now that the acyclic orientation of each subgraph $G_k(V_k, D_k)$ is obtained by finding a vertex coloring for this subgraph. This vertex coloring can be seen as a sequence of assignments $\alpha_1, \alpha_2, \ldots, \alpha_{|D_k|}$, such that every assignment α_l assigns a color to one operation $o_i \in V_k$ and leads to assigning directions to edges that connect o_i with other already colored operations $o_j \in V_k$. The number of assignments needed to perform the acyclic orientation of $G_k(V_k, D_k)$ is at most equal to the number of edges $|D_k|$. Following

the coloring of an operation and the engendered assignment of directions, the attributes of some operations may change. Two situations have to be distinguished:

- Coloring α_l of operation o_i does not lead to assigning a direction to any conflict edge. In this case, no changes of the timing attributes occur.
- Coloring α_l of operation o_i leads to assigning a direction to at least one conflict edge $[o_i, o_j] \in D_k$. Without any loss of generality, suppose that the edge $[o_i, o_j]$ is transformed into the arc (o_i, o_j). The start time $S(o_j)$ is changed as follows: $S(o_j) \leftarrow E(o_i)$. This leads to changing the end time $E(o_j)$ also and possibly causes a domino effect for the start times and end times of all the descendants $o_{j'} \in desc(o_j)$ (see Algorithm 2). Moreover, if $\overline{S}(o_j) > \overline{E}(o_i)$, the end time from end $\overline{E}(o_i)$ is changed as follows: $\overline{E}(o_i) \leftarrow \overline{S}(o_j)$. Similarly, this leads to changing the start time from end $\overline{S}(o_j)$ and possibly causes a domino effect for the start times and end times of all the ancestors $o_{i'} \in ance(o_i)$ (see Algorithm 3).

Algorithm 2: Update of the start and end times following an assignment α_l

Input : Attributes of the mixed graph $G(V, A, D)$, partially colored subgraph $G_k(V_k, A_k, D_k)$;
Output : Update of the start and end times of a subset of operations $\{o_i\} \subset V$;
Set α_l the last assignment of color made to an operation $o_i \in V_k$;
Set $A_{k,l} = \{(o_t, o_h)\}$ the set of all arcs created from the orientations engendered by α_l;
foreach $(o_t, o_h) \in A_{k,l}$ **do**
  if $S(o_h) < E(o_t)$ *and* $S(o_t) < E(o_h)$ **then**
    $S(o_h) \leftarrow E(o_t)$;
    $E(o_h) \leftarrow S(o_h) + C(o_h)$;
    `update(`o_h`)`;
Procedure `update(`o_h`)`
  if $succ(o_h) \neq \emptyset$ **then**
    foreach $o_{h'} \in succ(o_h)$ **do**
      if $S(o_{h'}) < E(o_h)$ **then**
        $S(o_{h'}) \leftarrow E(o_h)$;
        $E(o_{h'}) \leftarrow S(o_{h'}) + C(o_{h'})$;
        `update(`$o_{h'}$`)`;
  return;

We now describe our proposed acyclic orientation heuristic. The heuristic takes as input a mixed graph $G(V, A, D)$ and the attributes of the operations $o_i \in V$ as computed for the digraph $G(V, A, \emptyset)$, and assigns directions to all the edges $[o_i, o_j] \in D$. By applying Theorem 4.3.1, the heuristic consists in finding vertex colorings of the subgraphs which constitute the graph $G(V, A, D)$. In the first step, the graph $G(V, \emptyset, D)$ obtained

Algorithm 3: Update of the start and end times from end following an assignment α_l

Input : Input Attributes of the mixed graph $G(V, A, D)$, partially colored subgraph $G_k(V_k, A_k, D_k)$;

Output : Output Update of the start and end times from end of a subset of operations $\{o_i\} \subset V$;

Set α_l the last assignment of color made to an operation $o_i \in V_k$;

Set $A_{k,l} = \{(o_t, o_h)\}$ the set of all arcs created from the orientations engendered by α_l;

foreach $(o_t, o_h) \in A_{k,l}$ **do**

  if $S(o_h) < E(o_t)$ *and* $S(o_t) < E(o_h)$ **then**

    if $\overline{E}(o_t) < \overline{S}(o_h)$ **then**

      $\overline{E}(o_t) \leftarrow \overline{S}(o_h)$;

      $\overline{S}(o_t) \leftarrow \overline{E}(o_t) + C(o_t)$;

      `update(`o_t`)`;

Procedure `update(`o_t`)`

  if $pred(o_t) \neq \emptyset$ **then**

    foreach $o_{t'} \in pred(o_t)$ **do**

      if $\overline{E}(o_t^*) < \overline{S}(o_t)$ **then**

        $\overline{E}(o_{t'}) \leftarrow \overline{S}(o_t)$;

        $\overline{S}(o_{t'}) \leftarrow \overline{E}(o_{t'}) + C(o_{t'})$;

        `update(`$o_{t'}$`)`;

  return;

by removing all the arcs $(o_i, o_j) \in A$ from the mixed graph $G(V, A, D)$ is partitioned into s subgraphs where s is the sum of the repetition factors of all FMUs such that each subgraph contains all the operations of one FMU which correspond to the same occurrence and all the edges that connect them: $G(V, \emptyset, D) = \bigcup_{k=1}^{s} G_k(V_k, \emptyset, D_k)$. Then, the set of operations $o_i \in V$ is sorted in a non decreasing order of the start times $S(o_i)$. Next, the heuristic iteratively assigns colors to operations. It keeps a list of of already colored operations L_k for each subgraph $G(V_k, \emptyset, D_k)$. The operations of every list $o_i \in L_k$ are sorted in increasing order of their assigned colors. At each iteration, the heuristic selects among the operations not yet colored $o_i \in V$, the operation which has the earliest start time $S(o_i)$ to be assigned a color. Ties are broken by selecting the operation with the least flexibility. We call the operation to be colored at a given iteration, the *pending operation*. The heuristic checks in the order of L_k if the edges which connect the pending operation $o_i \in V_k$ with the operations $o_j \in L_k$ are conflict edges. If a conflict edge $[o_i, o_j] \in D_k : o_j \in L_k$ is detected, the pending operation is assigned the color $\alpha(o_j)$ and the colors assigned to all the already colored operations $o_{i'} \in L_k : \alpha(o_{i'}) \geq \alpha(o_i)$, are increased $\alpha(o_{i'}) \leftarrow \alpha(o_{i'}) + 1$. The corresponding edges are then accordingly assigned directions. Afterward, the timing attributes of the operations

are updated using Algorithms 2 and 3. At this point, the increase in CP, the critical path of the graph, is evaluated. Next, the operations $o_{i'} \in L_k$: $\alpha(o_{i'}) > \alpha(o_i)$ are reassigned their previous colors $\alpha(o_{i'}) = \alpha(o_{i'}) - 1$, and the pending operation is assigned a new color $\alpha(o_i) \leftarrow \alpha(o_i) + 1$. The increase in the critical path is evaluated again similarly. After repeating this process for all the edges $[o_i, o_{i'}] \in D_k : \ o_{i'} \in L_k$, the pending operation is finally assigned the color which leads to the least increase in the critical path, and edges $[o_i, o_{i'}] \in D_k : o'_i \in L_k$ are assigned directions accordingly. The heuristic begins another iteration by selecting a new operation to be colored. The heuristic assigns a color to one operation at each iteration. Every operation is assigned a color higher than the colors of all its predecessors which guarantees that no cycle is generated. The heuristic finally stops when all the operations have been assigned colors.

Complexity

The outermost loop (while loop) of the acyclic orientation heuristic is repeated n times, such that at each iteration, one operation is assigned a color. Recall that n is the number of operations in the operation graph $G(V, A)$. The selection of the operation with latest start time is done in $\mathcal{O}(\log n)$. The first inner loop iterates over all the edges connecting the selected operation. It is repeated at most e times, where e is the maximum number of edges connecting one operation. The inner most loop is executed twice in all cases. This results in an execution of the nested inner loops in $O(e)$. In addition Algorithms 2 and 3 that are called in the heuristic have each a complexity of $O(n)$ since they are based on a recursion whose depth is at most n. Therefore, the complexity of the acyclic orientation heuristic is evaluated to $\mathcal{O}(n^2 e)$.

4.4 Dependence Graph with Real-time Constraints

Real-time (co-)simulation, a widely used term in the literature, refers to co-simulation that requires that the amount of time needed to compute all equations of a model must be less than the integration step size of the model [80]. In this thesis, we talk instead about *co-simulation under real-time constraints*. The difference between the two notions will be clarified through this section. In particular, we are interested in such co-simulation within the context of HiL testing. In this section, we focus on defining these real-time constraints which are not given as it is usually the case in classical real-time systems. First, we explain what these constraints are and where they originate from. Then, we describe how to define real-time constraints for a co-simulation represented by an operation graph. The work presented in this section is based on the method of propagating real-time constraints described in [1]. This method was proposed for co-simulations where only partial information about intra-model dependence is available. Our work is an adaptation of this method to FMU co-simulation represented by a dependence graph which provides information about input/output/state dependence.

Algorithm 4: Acyclic orientation heuristic

Input : Mixed graph $G(V, A, D)$
Output : DAG $G(V, A)$;
Set s the number of all occurrences of all FMUs;
Partition the graph $G(V, \emptyset, D)$ into s subgraphs: $G(V, \emptyset, D) = \bigcup_{k=1}^{s} G_k(V_k, \emptyset, D_k)$;
Initialize lists $L_k \leftarrow \emptyset : 0 \leq k < s$;
Set Ω the set of all the operations not already colored;
while $\Omega \neq \emptyset$ **do**
  Select the operation $o_i \in V_k : \ S(o_i) = \max_{o_j \in \Omega}(S(o_j)), 0 \leq k < s$ (break ties by selecting the operation with the least flexibility);
  if $L_k = \emptyset$ **then**
    $\alpha(o_i) \leftarrow 1; \ L_k \leftarrow L_k \cup \{o_i\}$;
  else
    Set $\sigma \leftarrow \infty$; `// Initialize the increase in the critical path`
    foreach $o_j \in L_k$ **do**
      if $S(o_i) < E(o_j)$ *and* $S(o_j) < E(o_i)$ **then**
        foreach $c \in \{\alpha(o_j), \alpha(o_j) + 1\}$ **do**
          $\alpha(o_i) \leftarrow c$;
          `evaluate`(o_i, L_k);
          foreach $o_{i'} \in L_k$*:* $\alpha(o_{i'}) > \alpha(o_i)$ **do**
            Reassign $o_{i'}$ its previous color: $\alpha(o_{i'}) \leftarrow \alpha(o_{i'}) - 1$;
      if $S(o_i) \geq E(o_j)$ **then**
        $\alpha(o_i) \leftarrow \alpha(o_j) + 1$;
        `evaluate`(o_i, L_k);
      else
        $\alpha(o_i) \leftarrow \alpha(o_j)$;
        `evaluate`(o_i, L_k);
      $\alpha(o_i) = color$;
      foreach $o_{i'} \in L_k$ **do**
        if $\alpha(o_{i'}) > \alpha(o_i)$ **then**
          Assign a direction to the edge $[o_i, o_{i'}] \in D_k : \phi([o_i, o_{i'}]) \leftarrow (o_i, o_{i'})$;
        else
          Assign a direction to the edge $[o_i, o_{i'}] \in D_k : \phi([o_i, o_{i'}]) \leftarrow (o_{i'}, o_i)$;
      Update the timing attributes using Algorithms 2 and 3;
      Remove o_i from Ω; $L_k \leftarrow L_k \cup \{o_i\}$;

Procedure `evaluate`(o_i, L_k)
  foreach $o_{i'} \in L_k : \ \alpha(o_{i'}) \geq \alpha(o_i)$ **do**
    $\alpha(o_{i'}) \leftarrow \alpha(o_{i'}) + 1$;
    Update the timing attributes using Algorithms 2 and 3;
  Compute the new critical path and set σ' the increase in the critical path;
  if $\sigma' < \sigma$ **then**
    $color \leftarrow \alpha(o_i); \ \sigma \leftarrow \sigma'$;
  return;

4.4.1 Preliminaries

A HiL setup is composed of a simulated component and a physically available component that is interfaced with the simulated component via inputs and outputs. It should be noted that multiple parts may be physically available and involved in HiL, e.g. multiple controllers interacting with multiple parts of physical processes. We refer to all these parts as the real component. Figure 4.9 shows a basic example of a HiL co-simulation. The simulated component is represent by an operation graph. It consists of two FMUs A and B whose operations are colored in green and yellow respectively. The real component has one input and one output connected to an output operation and and an input operation of the simulated component respectively.

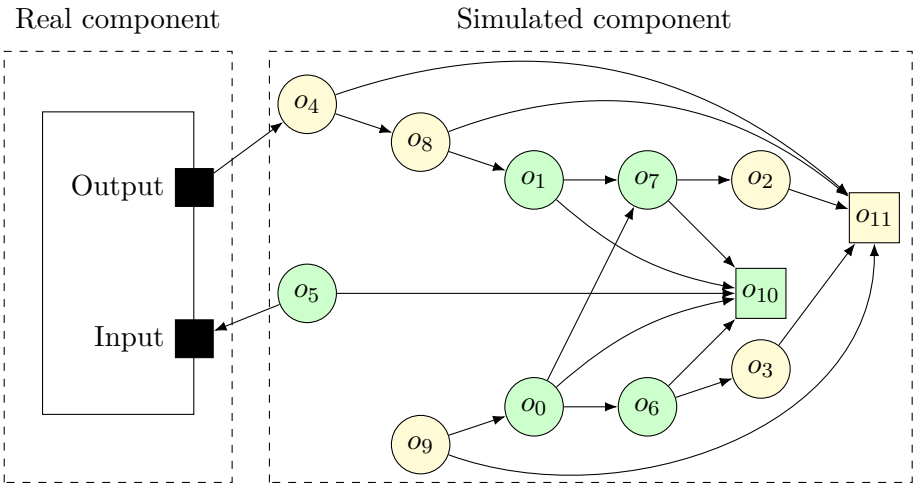


Figure 4.9: Example of co-simulation under real-time constraints.

The goal of the HiL testing phase in the model based design process is mainly to run realistic tests. In other words, it aims at estimating the performance of the real component by providing a realistic environment through the simulated component. The simulated component has to interact with the real component at the same rate as its real counterpart. As such, the inputs and outputs of the real component, which are periodically sampled, define real-time constraints which involve that the simulated time has to match the real time. These constraints are initially defined on the outputs and inputs of the simulated component that are connected with the inputs and outputs of the real component respectively. Since these inputs and outputs of the simulated component depend on other operations of the co-simulation, the real-time constraints are propagated towards the other operations.

In [1], it has been shown that the real-time constraints have to be propagated in different ways depending on the type of the operation (input, output, state) and also the type of intra-model connections (direct feedthrough, non direct feedthrough). In this work, the author dealt with co-simulation at the model level, i.e. a co-simulation is represented by a graph of models. Moreover, the author distinguishes between direct

feedthrough and non direct feedthrough model. A direct feedthrough model contains at least one output which directly depends on an input while a non direct feedthrough models contains none. For further details, one should consult the original work. Because we adopted a different approach which takes advantage of the FMI standard to represent a co-simulation with finer granularity, we cannot apply the method proposed in [1]. Instead, we propose a new method that we present in this section. First, we give in the following some preliminaries about the propagation of real-time constraints on a dependence graph representing an FMU co-simulation.

Different hardware and software components such as communication buses and software acquisition modules are used to connect the real component with the simulated component (co-simulation). In our work, we abstract the details about all these communication components away by representing the connection as a data dependence between an input (resp. output) of the real component and an output (resp. input) of the co-simulation (see Figure 4.9). The co-simulation periodically reads and writes data from and to the real component. Therefore, the real-time constrainsts are initially defined on the inputs and outputs that are directly connected with the real component. For instance, the real component sends data via its output to update an input of the co-simulation every $20ms$.

We consider that the simulated component consists in an FMU co-simulation represented as an operation graph constructed and upon which the different transformations (multi-rate, acyclic orientation) have been applied as described in previous sections. All operations of the co-simulation including the inputs (resp. outputs) of the co-simulation that are connected with the real component may have successors or not. This is an important difference from the method proposed in [1] where outputs of the co-simulation do not have successors.

We assume that the sampling period of a given input/output of the real component is a multiple of the communication step size of the operation of the simulated component that is connected with it. In contrast to [1], we do not consider the case where cycles exist in the operation graph since, as we showed previously, the operation graphs we deal with are cycle-free by construction.

The propagation of real-time constraints loosens the constraints imposed on the co-simulation compared to classical real-time co-simulation [80]. The latter, indeed, requires setting a periodic deadline for the computations of each model's equations that is equal to its integration step size. This approach, in the case of HiL co-simulation where only data exchange between the real and the simulated component have to be performed in real-time, usually over-constrains the co-simulation. We chose to use the term co-simulation under real-time constraints over the term real-time co-simulation because, in our approach, not every operation is constrained to be executed in real-time, i.e. with a periodic deadline that is equal to its integration step size. In the subsequent sections, we present how the real-time constraints are computed and propagated.

4.4.2 Definition of Real-time Constraints

Real-time systems are based on real-time tasks which represent the elementary units of execution. These tasks are characterized by a number of parameters such as periods, Worst Case Execution Times (WCET), deadlines, and release dates. Such parameters constitute an abstract model of the tasks that is used for the design and the analysis of real-time systems. We consider co-simulation under real-time constraints to be a real-time system where the operations of the co-simulation represent the real-time tasks. Therefore, the operation graph model presented in 4.1 has to be completed with real-time parameters that will allow the design and the analysis of real-time multi-core scheduling algorithms for co-simulation under real-time constraints. Hereinafter, we define real-time constraints that are assigned to operations, necessary for performing HiL co-simulation.

Let the inputs and the outputs of the real component be sampled with sampling periods T_x and T_y respectively where x and y denote the numbers of the corresponding input and output respectively. In other words, an input (resp. output) of the real component is periodically activated every T_x (resp. T_y) units of time. The sampling periods of the different inputs and outputs of the real component can be identical or different.We refer to operations of the simulated component that are directly connected to the real component as *gate operations*, e.g. operations o_4 and o_5 in Figure 4.9.

The periodic activation of an output of the real component leads to producing data that are consumed by an input gate operation o_i of the simulated component. This input gate operation is periodically updated by the values produced by the output of the real component following a sampling period T_y. Therefore, at the z^{th} sample $z \times T_y$, the input gate operation o_i is updated to its value corresponding to simulated time $z \times T_y$. This defines a periodic release for this input gate operation, i.e. time points at which its value is periodically updated following consumption of data produced by the real component.

Definition 4.4.1. A *release* is a real-time constraint applied on an input gate operation o_i of the simulated component in a HiL co-simulation. Such constraint is defined by its period $R(o_i) = T_y$ where T_y is the sampling period of the output of the real component that is connected with o_i. The occurrences of a release constraint are written $z \times T_y, z \in \mathbb{N}$. This means that the value of the gate operation o_i for simulated time $z \times T_y$ is available at real time $z \times T_y$.

Similarly, the periodic activation of an input of the real component requires data produced by an output gate operation o_j of the simulated component to be available. This output gate operation periodically produces data that is consumed by the input of the real component following a sampling period T_x. Therefore, before the w^{th} sample $w \times T_x$, the output gate operation o_j has to produce its value corresponding to simulated time $w \times T_x$. This defines a periodic deadline for this output gate operation, i.e. it has to periodically produce its updated value to be consumed by the real component before a specific periodic dates.

Definition 4.4.2. A *deadline* is a real-time constraint applied on an output gate o_j of the simulated component in a HiL co-simulation. Such constraint is defined by its period

$D(o_j) = T_x$ where T_x is the sampling period of the input of the real component that is connected with o_j. The occurrences of a deadline constraint are written $w \times T_x, w \in \mathbb{N}$. This means that the value of o_j for simulated time $w \times T_x$ has to be available at the latest by real time $w \times T_x$.

The aforementioned definitions specify real-time constraints for the operations that are directly connected with the real component. These operations being dependent on other operations and vice versa, it becomes necessary to define the impact of the real-time constraints on the rest of the operations.

4.4.3 Propagation of a Single Real-time Constraint

In this section, we assume that all the gate operations are subject to a single real-time constraint, i.e. all the real-time periods of the inputs and outputs of the real component are equal. In classical real-time co-simulation, real-time constraints are defined by setting a real-time period for each model that is equal to its integration step size. Such approach, if used in a HiL co-simulation, may lead to pessimistic constraints. In other words, it may overconstrain the co-simulation by setting stringent requirements that are not needed in order to ensure the real-time exchange between the simulated and the real components. In our approach, we only impose the necessary constraints on the gate operations. Then, these constraints are *propagated* to the remaining operations of the graph. These operations become subject to constraints that are induced from the constraints imposed on the gate operations. As stated previously, the main advantage of our approach is potentially loosening the real-time constraints imposed on the co-simulation.

Property 4.4.1. An operation graph $G(V, A)$ is said to satisfy property 4.4.1 if every operation $o_i \in V$ is assigned a release and a deadline.

The final goal is to apply real-time multi-core scheduling algorithm on the operation graph. For this, the operation graph model need to be completed with release and deadline constraints defined previously. We consider that any real-time multi-core scheduling algorithm will use these parameters. Therefore, we only consider operation graphs that satisfy Property 4.4.1. In the following, we present the process of propagating real-time constraints in an operation graph. We, also, derive a necessary condition that ensures that the operation graph resulting from the propagation of the real-time constraints is conformant with property 4.4.1.

Propagation of Release Constraints

Let a release constraint of period T_y be applied on an input gate operation $o_i \in V$. A subset of the occurrences of o_i, $o_i^s, 0 \leq s < r(o_i)$ that appear in the operation graph are subject to occurrences of the release constraint. The z^{th} occurrence of the release constraint $z \times T_y = t_k$ is applied on the occurrence o_i^{s,t_k}, i.e. the occurrence p of operation o_i executed at time step t_k, where $0 \leq p < r(o_i)$ and $t_k \in \mathbb{R}^+$. Since we consider the release constraint period to be a multiple of the communication step size of o_i, we can determine which

occurrences of o_i are subject to occurrences of the release constraint. More specifically, the occurrence $z \times T_y$ of the release constraint is applied on occurrence o_i^p : $p = z \times \frac{T_y}{H(o_i)}$. Therefore o_i^p is assigned a release $R(o_i^p) = z \times T_y$. Once o_i^p is released and executed, the operations that depend on o_i^p can be released. Therefore, the release constraint $R(o_i^p)$ is propagated towards all the successors of o_i^p. This propagation is given by expression 4.22.

$$\forall o_{i'} \in succ(o_i) : R(o_{i'}) = R(o_i) \tag{4.22}$$

Let's consider, for example, the HiL co-simulation shown in Figure 4.9. Let $H_A = 2$ and $H_B = 4$ be the communication step sizes of FMUs A and B respectively and let the sampling period of the output of the real component be $T_{out} = 4$. Figure 4.10 shows the propagation of the release constraint in the operation graph. Note that the multi-rate transformation algorithm is applied before the propagation. The operations that are assigned a release are colored in blue. The release assigned to each operation is shown below or above the corresponding operation. The dashed blue arrow indicates the direction of the propagation which is from a predecessor to a successor. The propagation of the release constraint is performed iteratively. Starting from the input gate operation o_4^p, for each operation that is assigned a release constraint, this constraint is propagated towards all its successors as described above. The values of release that are shown correspond to the first occurrence of the release constraint $z \times T_{out} = 0 \times 4 = 0$. In order to find the occurrence of the input gate operation o_4 that is subject to this occurrence of the release constraint, we use the formula given above: $p = z \times \frac{T_{out}}{H(o_4)} = 0 \times \frac{4}{4} = 0$. Therefore, in Figure 4.10, $p = 0$.

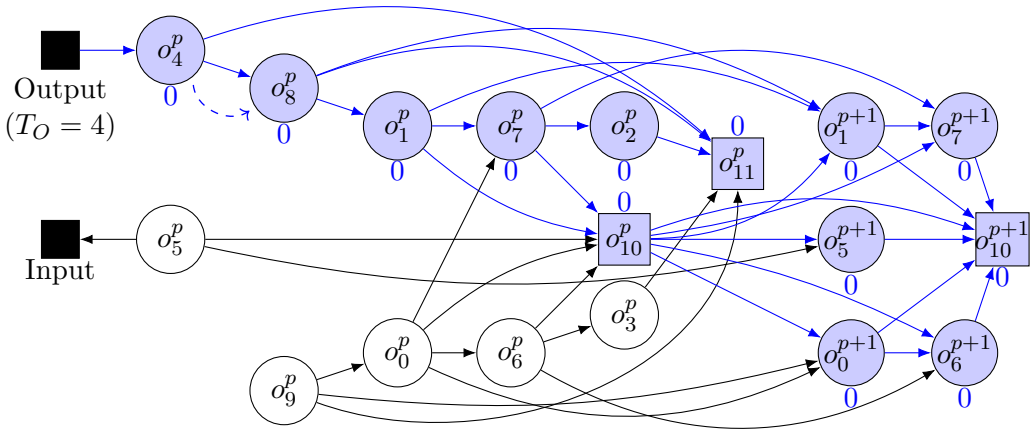


Figure 4.10: Example of release propagation.

The process of propagating a release constraint can be summarized as follows:

1. Let $G(V, A)$ be the operation graph representing the simulated component and let $o_i \in V$ be an input gate operation connected to an output of the real component whose sampling period and its occurrences are denoted T_y and $z \times T_y$ respectively.

2. Assign to the occurrences $o_i^p,\ 0 \leq p < r(o_i)$, the release constraints $R(o_i^p) = z \times T_y : p = z \times \frac{T_y}{H(o_i)}$.

3. For every operation o_i assigned a release constraint in the previous step: If $succ(o_i) = \emptyset$, stop. Otherwise, propagate $R(o_i)$ towards all the successors $o_{i'} \in succ(o_i)$.

4. Repeat the previous step for ever operation that is newly assigned a release constraint.

Propagation of Deadline Constraints

Now let a deadline constraint of period T_x be applied on an output gate operation o_j of the operation graph. Occurrences of this deadline constraint are applied on a subset of the occurrences of operation o_j, $o_j^q : 0 \leq q < r(o_j)$ that appear in the operation graph. The w^{th} occurrence of the deadline constraint $w \times T_x = t_k$ is applied on the occurrence o_j^{q,t_k}, i.e. the occurrence q of operation o_j executed at time step t_k, where $0 \leq q < r(o_j)$ and $t_k \in \mathbb{R}^+$. The deadline constraint is a multiple of the communication step size of o_j, hence, we can compute which occurrences of o_j are subject to occurrences of the deadline constraint. The occurrence $w \times T_x = t_k$ of the deadline constraint is applied on occurrence $o_j^q :\ q = w \times \frac{T_x}{H(o_j)}$. Each occurrence of the operation o_j that is subject to an occurrence of the deadline constraint $w \times \frac{T_x}{H(o_j)}$ is assigned a deadline $D(o_j^q) = w \times \frac{T_x}{H(o_j)}$. This constraint is, then, propagated towards all the predecessors of o_j^q. The propagation of a deadline constraint is given by Expression 4.23.

$$\forall o_{j'} \in pred(o_j) : D(o_{j'}) = D(o_j) \tag{4.23}$$

Figure 4.11 illustrates the propagation of the deadline constraint in the operation graph. The sampling period of the input of the real component is $T_{in} = 4$. We recall that the execution of the co-simulation consists in running the operation graph repeatedly. Therefore, each run corresponds to a pattern that involves specific occurrences of the operations. While in the pattern of the operation graph that is shown previously, operation o_5^p does not have a predecessor, the state operation o_{10}^{p-1} belonging to the preceding pattern is a predecessor of o_5^p. The operations that are assigned a deadline are colored in red. The deadline assigned to each operation is shown above the corresponding operation. The dashed red arrow indicates the direction of the propagation which is from a successor to a predecessor. The propagation of the deadline constraint is performed iteratively. Starting from the output gate operation o_5^q, for each operation that is assigned a deadline constraint, this constraint is propagated towards all its predecessors. The values of deadline that are shown correspond to the second occurrence of the release constraint $w \times T_{in} = 1 \times 4 = 4$. In order to find the occurrence of the output gate operation o_5 that is subject to this occurrence of the release constraint, we use the formula given previously: $q = w \times \frac{T_{in}}{H(o_5)} = 1 \times \frac{4}{2} = 2$. Therefore, in Figure 4.11, $q = 2$.

The process of propagating a deadline constraint can be summarized in the following steps:

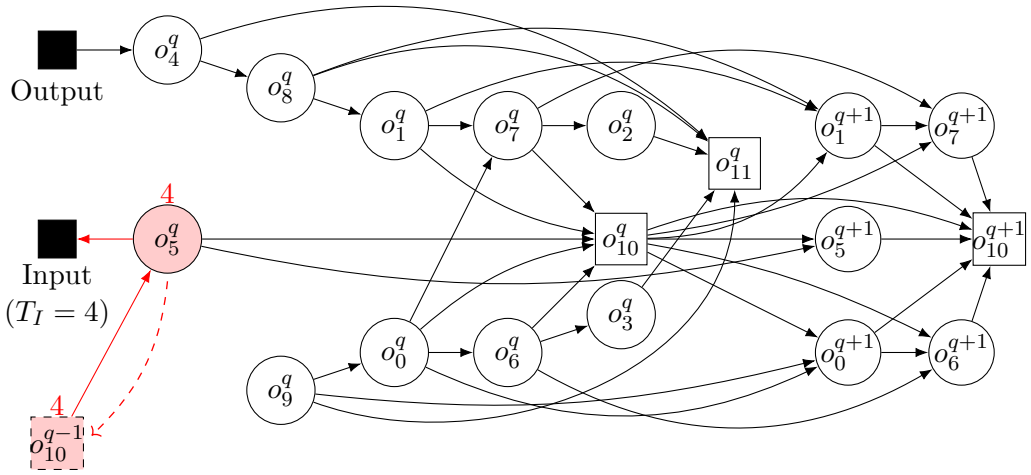


Figure 4.11: Example of deadline propagation.

1. Let $G(V, A)$ be the operation graph representing the simulated component and let $o_j \in V$ be an output gate operation connected to an input of the real component whose sampling period and its occurrences are denoted T_x and $w \times T_x$ respectively.

2. Assign to the occurrences o_j^q, $0 \leq q < r(o_j)$, the deadline constraints $D(o_j^q) = w \times T_x : \ q = w \times \frac{T_x}{H(o_j)}$.

3. For every operation o_j assigned a deadline constraint in the previous step: If $pred(o_j^q) = \emptyset$, stop. Otherwise, propagate $D(o_j)$ towards all the predecessors $o_{j'} \in pred(o_j)$.

4. Repeat the previous step for ever operation that is newly assigned a deadline constraint.

Propagation of Real-time Constraints through Looping

It can be seen that the propagation of the real-time constraints is based on graph traversal. The Breadth First Search (BFS) or the Depth First Search (DFS) graph traversal algorithms can be used to perform this propagation. On the one hand, the release constraints are propagated following the topological ordering of the graph since the constraint is always propagated from a predecessor to a successor. We refer to such propagation as *forward propagation*. On the other hand, the deadline constraint is propagated in reverse topological ordering of the graph because the constraint is always propagated from a successor to a predecessor. We refer to such propagation as *backward propagation*. This means that the release (resp. deadline) constraint cannot propagate towards the operations that come before (resp. after) the input (resp. output) gate operation that is subject to this constraint.

For the operation graph to be scheduled by a real-time multi-core scheduling algorithm, it has to satisfy Property 4.4.1. Below, we propose a method to assign release and deadline

constraints to operations that are not traversed in the forward and backward propagation phases respectively. This method is based on *looping* the propagation. In other words, a release (resp. deadline) constraint is propagated in a reverse order of the forward (resp. backward) propagation phase. Such looping is possible because we can define a pattern of the operation graph that is repeated periodically in runtime.

Definition 4.4.3. In the context of co-simulation under real-time constraints, the *hyperperiod* is the least common multiple of the real-time sampling periods of all inputs and outputs of the real component and the communication step sizes of all operations of the simulated component: $HP = lcm(H(o_1), H(o_2), \ldots, H(o_n), T_1, T_2, \ldots, T_m)$ where n is the number of operations in the operation graph and m is the number of inputs and outputs of the real component.

The hyperperiod notion resembles the notions of hyperperiod found in the real-time literature and of hyperstep defined in Section 4.2. More specifically, it specifies a time interval for describing a periodic pattern of the real-time constraints assigned to operations. It only differs from both notions in that it combines real-time periods and communication step sizes. In the context of co-simulation under real-time constraints, we apply the multi-rate transformation algorithm over the hyperperiod instead of the hyperstep. Therefore, the repetition factor of each operation $o_i \in V$ (see Section 4.2) becomes $r(o_i) = \frac{HP}{H(o_i)}$.

Given the periodic pattern of the operation graph, a release constraint can be written as follows:

$$\forall o_i^p, o_i^{p'} : p' = p - \frac{HP}{H(o_i)},\ R(o_i^{p'}) = R(o_i^p) - HP \tag{4.24}$$

Similarly, a deadline constraint can be written as follows:

$$\forall o_j^q, o_j^{q'} : q' = q + \frac{HP}{H(o_j)},\ D(o_j^{q'}) = R(o_j^u) + HP \tag{4.25}$$

Let's consider that a release constraint is propagated in the operation graph $G(V, A)$ starting from the gate operation o_i. Also, consider that the multi-rate transformation algorithm has been applied on the operation graph over the hyperperiod HP. Then, $o_{i'}^p$, the last occurrence of the state operation $o_{i'} : f_m(o_{i'}^p) = f_m(o_i)$ that is assigned a release constraint during the forward propagation phase corresponds to the time step that is equal to the hyperperiod, i.e. it can be written $o_{i',HP}^p$. In order to propagate the release constraint to the operations that come before o_i in the graph, we loop back to the occurrence $o_{i'}^{p'}$ of the state operation $o_{i'}$ by performing a negative shift of the release constraint whose length is equal to the hyperperiod. The occurrence $o_{i'}^{p'}$ is a predecessor of the first occurrence of the gate operation o_i that appear in the periodic pattern of the operation graph. This is done for every FMU, for which the state operation is assigned a release. An example is shown in Figure 4.12.

Afterward, starting from the operations that are newly assigned release constraints, a new forward propagation phase is applied. In Figure 4.13, this phases starts from

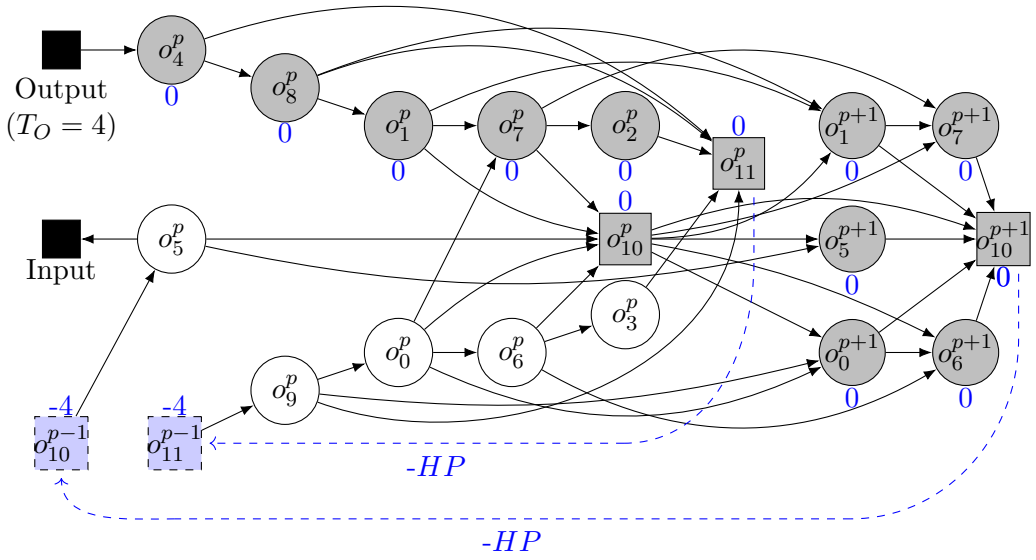


Figure 4.12: Release back loop propagation phase.

operations o_{10}^{p-1} and o_{11}^{p-1}. The propagation ends after this phase as every operation is assigned a release date.

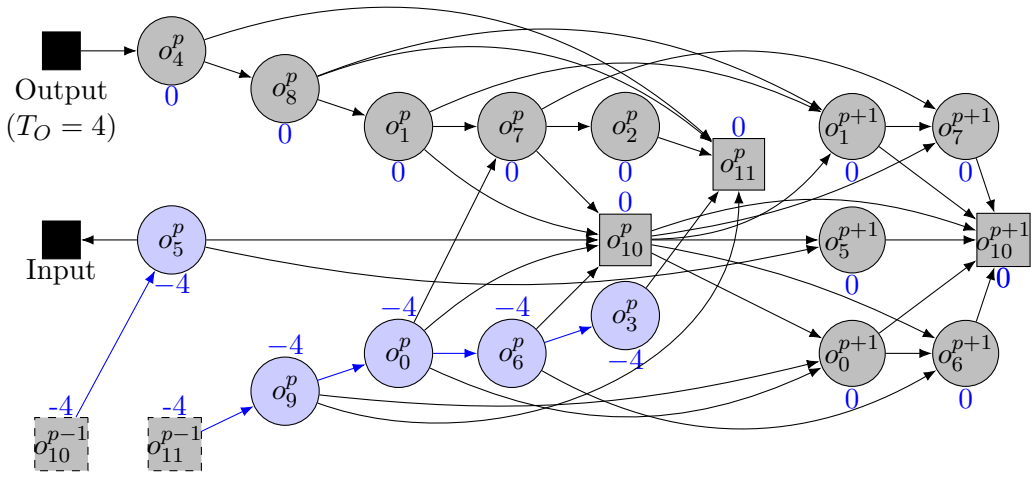


Figure 4.13: Second release forward propagation phase.

Now consider that a deadline constraint is propagated in the graph $G(V, A)$ starting from the gate operation o_j. Let $o_{j'}^{q'}$ be the occurrence of the state operation $o_{j':f_m(o_{j'}^p)=f_m(o_j)}$ that is a predecessor of the first occurrence of the gate operation o_j that appear in the periodic pattern of the operation graph. Also, let $o_{j'}^{q}$ be the last occurrence of the state operation $o_{j'}$ that appear in the periodic pattern of the operation graph. Like for the release constraint, the operations that come after o_j in the operation graph can be assigned deadline constraints by looping forward the deadline constraint that is

assigned to $o_{j'}^{q'}$ towards $o_{j'}^{q}$. In other words, a positive shift equal to the length of the hyperperiod is applied by looping forward to the last occurrence of the state operation $o_{j'}^{q}$. This is performed for every FMU whose state operation was assigned a deadline date. Figure 4.14 shows an example of such looping.

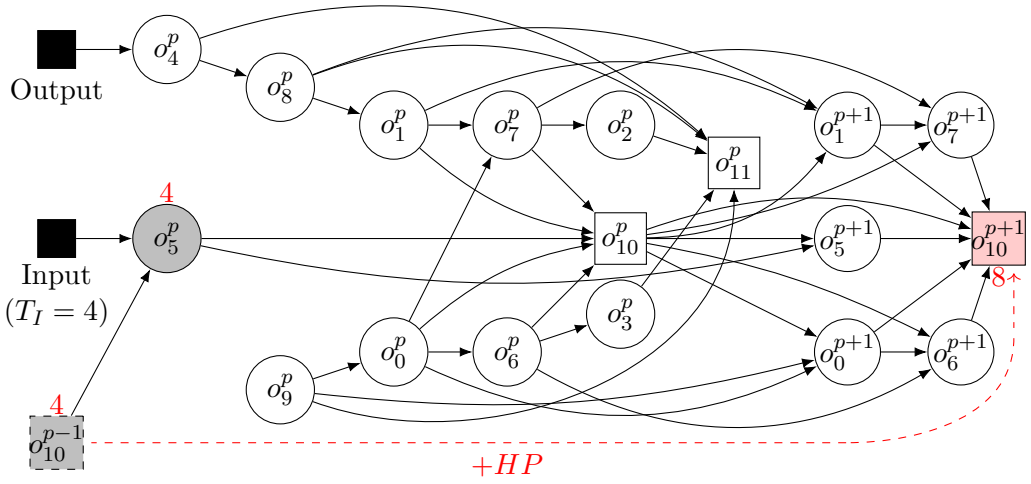


Figure 4.14: Deadline forward loop propagation phase.

A backward propagation phase, starting from the operations that are newly assigned deadline constraints, is then applied as shown if Figure 4.15.

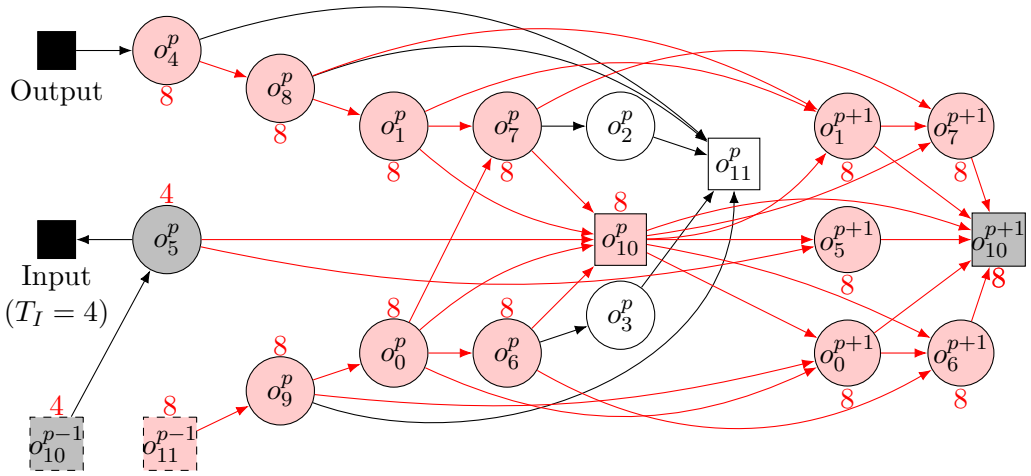


Figure 4.15: Second deadline backward propagation phase.

Figure 4.16 shows the forward loop propagation that is is then performed. Finally, a last backward propagation phase is applied starting from operation o_{11}^{p}.

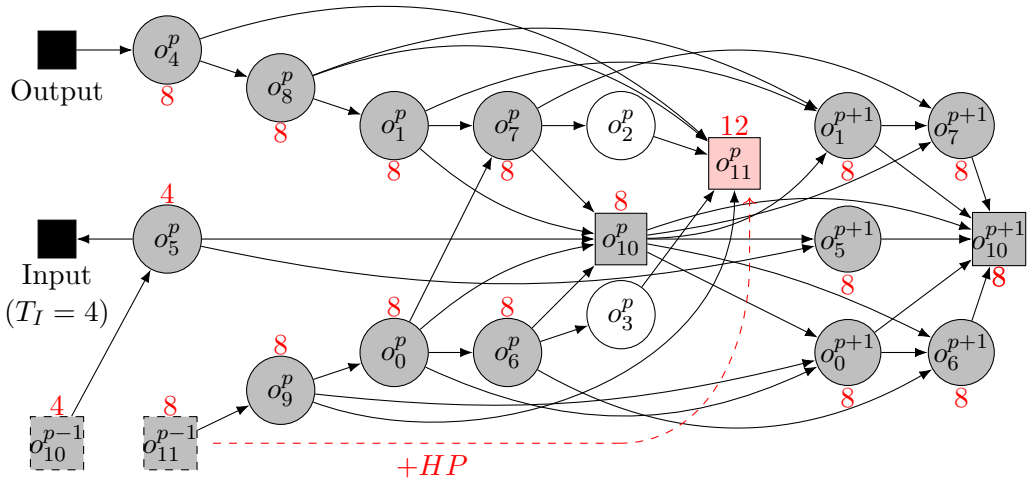


Figure 4.16: Second deadline forward loop propagation phase.

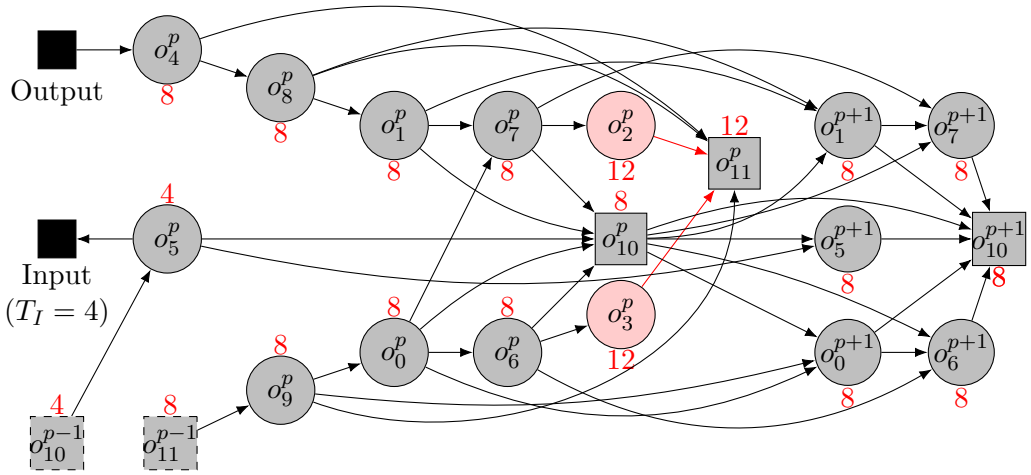


Figure 4.17: Third deadline backward propagation phase.

Necessary Condition for Covering the Operation Graph

After presenting the method for assigning real-time constraints to the operations, we give, here, a necessary condition to satisfy Property 4.4.1. We recall that if Property 4.4.1 is not satisfied, it is not possible to apply a real-time multi-core scheduling algorithm on the operation graph. For the propagation of real-time constraints to result in an operation graph that satisfies Property 4.4.1, the necessary condition stated in Theorem 4.4.1 must be satisfied.

Theorem 4.4.1. *Let $G(V, A)$ be an operation graph representing an FMU co-simulation under real-time constraints. Let o_i and o_j denote the input and an output gate operations respectively. The propagation of the release (resp. deadline) constraint applied on a gate*

operation o_i (resp. o_j) leads to assigning a release (resp. deadline) constraint to every operation $o_k \in V$ only if there exists at least one path from an operation $o_{i'} : f_m(o_{i'}) = f_m(o_i)$ (resp. $o_{k'} : f_m(o_{k'}) = f_m(o_k)$) to at least one operation $o_{k'} : f_m(o_{k'}) = f_m(o_k)$ (resp. $o_{j'} : f_m(o_{j'}) = f_m(o_j)$).

Proof. Suppose that there is an FMU whose operations are denoted $o_k \in V : k \neq i$ such that there exists no path from an operation $o_{i'} : f_m(o_{i'}) = f_m(o_i)$ to an operation $o_{k'} : f_m(o_{k'}) = f_m(o_k)$. Running the forward propagation phase, no operation $o'_k : f_m(o_{k'}) = f_m(o_k)$ (including o_k) is traversed. Thus, none of these operations is a assigned a release constraint during the forward propagation phase. As a consequence, it is not possible to loop back on the state operation of the FMU $f_m(o_k)$. Therefore, the operation o_k is not reached by the forward propagation phase nor by the back loop phase. □

4.4.4 Propagation of Multiple Real-time Constraints

In the previous section, we considered that the operation graph is subject to real-time constraints that are equal, i.e. there is a single value for the sampling periods of the inputs and outputs of the real-time component. In industrial applications, this is not always the case. In this section, we consider that multiple input and output gate operations are subject to different release and deadline constraints respectively. This means the sampling periods of the inputs and the outputs of the real component are different. As before, we consider that every sampling period is a multiple of the communication step size of the gate operation that is applied to.

In order to propagate multiple real-time constraints, we follow the propagation process described previously. The release constraints are propagated by iteratively running forward propagation and back loop propagation phases whereas deadline constraints are propagated by iteratively running backward propagation and forward loop propagation phases. The main difference here is that during propagation, several constraints may be applied on the same operation. We handle this situation in a similar way to [1].

Gate operations are always subject to only one constraint because they are directly assigned the constraints imposed by the real component. Therefore, assigning constraints to gate operations remains unchanged. On the other hand, the rest of the operations of the operation graph may have several predecessors and successors. In fact, if an operation $o_i \in V$ has no more than one predecessor, i.e. $|pred(o_i)| \leq 1$, the propagation of a release constraint towards this operation remains unchanged. Similarly, if an operation $o_j \in V$ has no more than one successor, i.e. $|succ(o_j)| \leq 1$, the propagation of a deadline constraint towards this operation remains unchanged. Hence, we are interested in this section in propagating release (resp. deadline) constraints towards operations which have more than one predecessor (resp. successor), i.e. $|pred(o_i)| > 1$ (resp. $|succ(o_j)| > 1$).

We denote by $\mathcal{R}(o_i)$ the set of release constraints propagated towards the operation o_i and by $\mathcal{D}(o_j)$ the set of deadline constraints propagated towards the operation o_j. The set $\mathcal{R}(o_i)$ is built as the union of all the release constraints propagated towards o_i from its predecessors, i.e. $\mathcal{R}(o_i) = \{R(o_{i'}) : o_{i'} \in pred(o_i)\}$. Each operation

o_i must be assigned only one release constraint which is chosen from the set $\mathcal{R}(o_i)$. A release constraint specifies the earliest date the associated operation can start its execution. As such, the most constraining release constraint has to be chosen from the set $\mathcal{R}(o_i)$ as stated by expression 4.26.

$$\forall o_i \in V, R(o_i) = \max_{R(o_{i'}) \in \mathcal{R}(o_i)} (R(o_{i'})) \tag{4.26}$$

The process of propagating the release constraints described previously becomes as follows:

1. Let $G(V, A)$ be the operation graph representing the simulated component and let $o_1, o_2, \ldots o_{in}, \in V$ be input gate operations connected to outputs of the real component whose sampling periods and their occurrences are denoted $T_1, T_2, \ldots, T_{on}$ and $z_1 \times T_1, z_2 \times T_2, \ldots, z_{on} \times T_{on}$ respectively.

2. For every input gate operation o_i, assign to the occurrence $o_i^p, 0 \leq p < r(o_i)$, the release constraint $R(o_i^p) = z_k \times \frac{T_k}{H(o_i)} : p = z_k \times \frac{T_k}{H(o_i)}$.

3. For every operation o_i assigned a release constraint in the previous step: if $succ(o_i) = \emptyset$, stop. Otherwise, propagate the constraint $R(o_i)$ towards all the successors $o_i' \in succ(o_i)$.

4. Repeat the previous step for every operation that is newly assigned a release constraint.

5. For every operation o_i towards which multiple release constraints have been propagated, build the set $\mathcal{R}(o_i)$. Assign to o_i a single release constraint using expression 4.26.

In the same way, the set $\mathcal{D}(o_j)$ is built as the union of all the deadline constraints propagated towards o_j from its successors, i.e. $D(o_{j'}) : \ o_{j'} \in succ(o_j)$. Each operation must be assigned only one deadline constraint which is chosen from the set $\mathcal{D}(o_j)$. A deadline constraint specifies the latest date by which the associated operation must finish its execution. Therefore, the most constraining deadline constraint is selected from the set $\mathcal{D}(o_j)$ as specified by expression 4.27.

$$\forall o_j \in V, D(o_j) = \min_{D(o_{j'}) \in \mathcal{D}(o_j)} (D(o_{j'})) \tag{4.27}$$

The process of propagating the deadline constraints described previously is adapted as follows as follows:

1. Let $G(V, A)$ be the operation graph representing the simulated component and let $o_1, o_2, \ldots o_{jn}, \in V$ be output gate operations connected to inputs of the real component whose sampling periods and their occurrences are denoted $T_1, T_2, \ldots, T_{in}$ and $w_1 \times T_1, w_2 \times T_2, \ldots, w_{in} \times T_{in}$ respectively.

2. For every output gate operation o_j, assign to the occurrence $o_j^q, 0 \leq q < r(o_j)$, the deadline constraint $D(o_j^k) = w_k \times \frac{T_k}{H(o_j)} : \ q = w_k \times \frac{T_k}{H(o_j)}$.

3. For every operation o_j assigned a deadline constraint in the previous step: if $pred(o_i) = \emptyset$, stop. Otherwise, propagate the constraint $R(o_j)$ towards all the predecessors $o'_j \in succ(o_j)$.

4. Repeat the previous step for every operation that is newly assigned a deadline constraint.

5. For every operation o_j towards which multiple release constraints have been propagated, build the set $\mathcal{D}(o_j)$. Assign to o_j a single deadline constraint using expression 4.27

4.4.5 Propagation Algorithms

We propose two algorithms to perform the propagation of release and deadline constraints, respectively, in an operation graph. The proposed algorithms are based on the previously presented concepts. As stated previously, the propagation is based on graph traversal and therefore the propagation algorithms can be based on graph traversal algorithms such as BFS or DFS. Since only graphs that satisfy Property 4.4.1 are considered, we, first check that the necessary condition stated in Theorem 4.4.1 is satisfied. Otherwise, we consider that the co-simulation cannot be properly run under real-time constraints. We aim at formulating propagation algorithms with low complexity. The order in which the different phases of the the propagation are performed may affect the algorithm's complexity. Therefore, this order should be chosen carefully as detailed hereafter.

Let's start with the propagation of the release constraint. The propagation starts with the forward phase where, starting from the gate operations, the release constraints are propagated form each operation subject to the release constraint to its successors. The proposed algorithm performs these propagations iteratively, in a decreasing order of the periods of the release constraints. Following such order may avoid performing unnecessary propagations. To show this, consider, for instance, that operation o_i is subject to two different release constraints T_1 and T_2 such that $T_1 > T_2$. If the propagation of T_2 is performed first, all the paths in the operation graph that start at o_i are traversed twice, first to propagate T_2 and then to propagate T_1. However, if the T_1 is propagated first, the said paths are only traversed once, propagating T_1. The propagation of T_2 is stopped at o_i with the use of expression 4.26.

The same idea applies to the order in which occurrences of a release constraint are propagated. For each release constraint, the proposed algorithm successively performs the propagation of its occurrences in a decreasing order, i.e. starting with the last occurrence and finishing with first one. Following the same reasoning presented in the previous paragraph, it can be seen how such order may avoid unnecessary traversals. Consider, for example, the propagation of occurrences of the release constraint T_y applied to operation o_i. Suppose that the first occurrence $0 \times T_y$ is propagated first. This assigns

the release constraint $0 \times T_y$ to occurrence o_i^0 and leads to propagating it to all subsequent occurrences $o_i^p : 0 < p < r(o_i)$ and their successors. Then, propagating the subsequent occurrence $1 \times T_y$ assigns the release constraint $1 \times T_y$ to occurrence $o_i^{p'} : p' = 1 \times \frac{T_y}{H(o_i)}$ and leads to propagating it to all subsequent occurrences $o_i^{p''} : p' < p'' < r(o_i)$ and their successors. As such, all the paths that start at $o_i^{p'}$ are traversed twice. However, performing the propagation in a decreasing order of the occurrences of T_y remedies to this issue For instance, in above example, after $R(o_i^{p'})$ is propagated, the propagation of $R(o_i^0)$ stops at $o_i^{p'}$ with the use of expression 4.26. Therefore, the paths starting at $o_i^{p'}$ are not unnecessarily traversed again.

The last phase of the propagation of the release constraints is the loop back phase. This phase is performed following the same technique, i.e. starting with the greatest constraint. Similarly, this avoids performing unnecessary traversals. Algorithm 5 lists the proposed algorithm for propagation of release constraints.

We now describe, in a similar way, the algorithm of deadline propagation. The propagation starts with the backward porpagation phase. Deadline constraints are defined for gate operations and then propagated to the successors of every operation that is assigned a deadline constraint. The propagations of different deadline constraints are performed following an increasing order of the the periods of these constraints. As stated previously for the release propagation, this order is chosen in order to minimize the complexity of the propagation algorithm by avoiding unnecessary traversals of the graph. For example, consider an operation o_j that is subject to two deadline constraints T_1 and T_2 such that $T_1 < T_2$. Propagating T_2 first leads to traversing twice all the paths that end at o_j, propagating T_2 and then T_1. If, instead, T_1 is propagated first, those paths are only traversed once, propagating T_1. Then, the propagation of T_2 is stopped at o_j thanks to the rule given by expression 4.27.

In a similar way, the occurrences of a deadline constraint are propagated in an increasing order. For instance, let the operation o_j be subject to the deadline constraint T_x. Suppose that the occurrences of T_x are propagated in a decreasing order. Therefore, the last occurrence $w \times T_x$ is propagated first. This assigns the deadline constraint $w \times T_x$ to the operation occurrence $o_j^q : q = w \times \frac{T_x}{H(o_j)}$ and all the preceding occurrences $o_j^{q'} : 0 \leq q' < q$ and their predecessors. The propagation of the preceding occurrence $(w-1) \times T_x$ assigns the deadline constraint $(w-1) \times T_x$ to the operation occurrence $o_j^{q''} : q'' = (w-1) \times \frac{T_x}{H(o_j)}$ and propagates $(w-1) \times T_x$ to all the preceding operation occurrences $o_j^{q'''} : 0 < q''' < q''$ and their predecessors. This leads to traversing all the paths that end at $o_j^{q'}$ twice. In order to avoid these unnecessary traversals, the occurrences of T_x are traversed in a decreasing order. In the aforementioned example, after $o_j^{q''}$ is propagated, the propagation of o_j^q stops at o_j^q by using expression 4.27. Consequently, the paths that end at $o_j^{q''}$ are not traversed again.

Lastly, the loop forward phase of propagation is performed, always, in an increasing order of the deadline constraints. Algorithm 6 details the proposed algorithm for the propagation of deadline constraints.

Algorithm 5: Release propagation algorithm

Input : Operation graph $G(V, A)$, sampling periods of the real component
Output : Assignment of release constraints to operations $o_i \in V$
Set O the set of input gate operations;
Set T_i the period of the output of the real component that is connected to o_i;
Sort O in a decreasing order of $T_i : o_i \in O$;
Set HP the hyperperiod;
foreach $o_i \in V$ **do**
  $R(o_i) \leftarrow -\infty$;
// Forward propagation phase
foreach *operation* $o_i \in O$ **do**
  for $p \leftarrow r(o_i) - 1$ **downto** 0 **by** $\frac{T_i}{H(o_i)}$ **do**
    $z \leftarrow p \times \frac{H(o_i)}{T_i}$;
    $R(o_i^p) \leftarrow z \times T_i$;
    `propagate`(o_i^p);
// Back loop propagation phase
foreach *FMU* M *whose last occurrence of the state operation* o_j^u *has been assigned a release date* **do**
  Let $o_j^{u'}$ be the occurrence of the state operation o_j of FMU M which precedes the operation graph pattern;
  $R(o_j^{u'}) \leftarrow R(o_j^u) - HP$;
  `propagate`(o_j^0);
Procedure `propagate`(o_i)
  if $succ(o_i) \neq \emptyset$ **then**
    foreach $o_{i'} \in succ(o_i)$ **do**
      if $R(o_{i'}) < R(o_i)$ **then**
        $R(o_{i'}) \leftarrow R(o_i)$;
        `propagate`$(o_{i'})$;
  return;

Algorithm 6: Deadline propagation algorithm

Input : Operation graph $G(V, A)$, sampling periods of the real component
Output : Assignment of deadline constraints to operations $o_j \in V$
Initialization;
Set O the set of output gate operations;
Set T_j the period of the input of the real component that is connected to $o_j \in O$;
Sort O in an increasing order of $T_j : o_j \in O$;
Set HP the hyperperiod;
foreach $o_j \in V$ **do**
  $D(o_j) \leftarrow \infty$
// Backward propagation phase
foreach *operation* $o_j \in O$ **do**
  for $q \leftarrow 0$ **to** $r(o_j) - 1$ **by** $\frac{T_j}{H(o_j)}$ **do**
    $w \leftarrow q \times \frac{H(o_j)}{T_j}$;
    $D(o_j^q) \leftarrow w \times T_j$;
    `propagate`(o_j^q);
// Forward loop propagation phase
foreach *FMU* M *whose the occurrence of the state operation* o_i^s *which precedes the operation graph pattern has been assigned a deadline constraint* **do**
  Let $o_i^{s'}$ be the last occurrence of FMU M in the operation graph;
  $D(o_i^{s'}) \leftarrow D(o_i^s) + HP$;
  `propagate`(o_i^u);
Procedure `propagate`(o_j)
  if $pred(o_j) \neq \emptyset$ **then**
    foreach $o_{j'} \in pred(o_j)$ **do**
      if $D(o_{j'}) > D(o_j)$ **then**
        $D(o_{j'}) \leftarrow D(o_j)$;
        `propagate`$(o_{j'})$;
  return;

5

Multi-core Scheduling of FMU Dependence Graphs

Contents

This chapter presents methods for scheduling an operation graph on a multi-core architecture. Once the operation graph has been constructed and undergone the different phases of transformations as described in the previous chapter, it is scheduled on the multi-core platform. First, we consider scheduling the operation graph with the goal of accelerating the execution of the co-simulation. Second, we consider scheduling the operation graph while satisfying real-time constraints.

5.1 Scheduling of Dependence Graphs for Co-simulation Acceleration

In order to achieve fast execution of the co-simulation on a multi-core processor, an efficient allocation and scheduling of the operation graph has to be achieved. The scheduling algorithm has to be applied taking into account functional and non functional specification in order to produce an allocation of the operation graph vertices (operations) to the cores of the processor, and assign a starting time to each operation. We present hereafter a linear programming model and a heuristic for scheduling operation graphs on multi-core processors with the aim of accelerating the execution of the co-simulation. Note that no real-time constraints are involved when the goal is to accelerate the co-simulation. Therefore, the transformations that are applied on the operation graph ahead of applying the scheduling algorithm are only the multi-rate transformation and the acyclic orientation.

5.1.1 Problem Formulation

The acceleration of the co-simulation corresponds to the minimization of the makespan of the operation graph. The makespan is the total execution time of the whole graph. The operation graph that is fed as input to the scheduling algorithm is a DAG, therefore, it represents a partial order relationship in the execution of the operations, since two operations connected by an arc must be executed sequentially whereas the other ones can be executed in parallel. A scheduling algorithm makes decisions on allocating the operations to the cores while respecting this partial order and trying to minimize the total execution time of the operation graph. In addition to the execution time of the operations, the scheduling algorithm has to take into consideration, the cost of inter-core synchronization. The set of cores is denoted $P = \{p_1, p_2, \ldots, p_m\}$ where m is the number of cores. In this thesis we adopt a non preemptive scheduling solution. The scheduling problem can be stated as an optimization problem as follows:

Input	Operation graph $G(V, A)$
Output	Offline Schedule of operations on a multi-core processor
Find	Allocation of operations to cores: $\alpha : V \rightarrow P$ Assignment of start times to operations: $\beta : V \times P \rightarrow \mathbb{N}$
Minimize	Makespan of the graph: $\min(mkp) : \ mkp = \max_{o_i \in V}(E(o_i))$
Subject to	Precedence constraints: $\forall \ (o_i, o_j) \in A, \ S(o_j) \geq E(o_i)$

5.1.2 Resolution using Linear Programming

As a first attempt to solve the problem of scheduling the operation graph, we decided to use the ILP approach which is an exact algorithm. By using ILP, we guarantee that the obtained schedule is optimal. In addition, ILP allows us to model the problem by means of linear relationships between variables and feed this model to an existing ILP solver to compute the solution. Below, we give our ILP formulation of the problem of scheduling the operation graph for co-simulation acceleration.

Variables and Constants

Tables 5.1 and 5.2 summarize respectively the variables and the constants that are used in the ILP formulation of the scheduling problem for co-simulation acceleration.

Table 5.1: Variables used in the ILP formulation of the acyclic orientation problem

Variable	Type	Description
x_{ik}	Binary	Decision variable for scheduling operation o_i on core p_k
$S(o_i)$	Integer	Start time of operation o_i
$E(o_i)$	Integer	End time of operation o_i
$sync_{ijk}$	Binary	Synchronization between o_i and o_j if o_j scheduled on p_k
b_{ij}	Binary	o_i is executed before o_j
Q_{ik}	Integer	Earliest start time of successors o_i oi that are scheduled on p_k
V_{ik}	Binary	o_i not scheduled on p_k
mkp	Integer	Makespan

Table 5.2: Constants used in the ILP formulation of acyclic orientation problem

Constant	Type	Decription
$C(o_i)$	Integer	Execution time of operation o_i
M	Integer	Large positive number
$synCost$	Integer	Cost of synchronization

Constraints

We define the decision binary variables x_{ik} which indicates whether the operation o_i is allocated to core p_k or not. Expression 5.1 states the constraint that each operation has to be allocated to one and only one core.

$$\forall\ o_i \in V, \sum_{p_k \in P} x_{ik} = 1 \tag{5.1}$$

The end time of each operation o_i is computed using the expression 5.2

$$\forall o_i \in V, E(o_i) = S(o_i) + C(o_i) \tag{5.2}$$

For operations that are allocated to the same core and that are completely independent, i.e. no path exists between them, we have to ensure that they are executed in non overlapping time intervals. Expressions 5.3 and 5.4 capture this constraint. b_{ij} is a binary variable that is set to one if o_i is executed before o_j.

$$\forall p_k \in P, \forall\ o_i, o_j \in V, (o_i, o_j), (o_j, o_i) \notin A, E(o_i) \leq S(o_j) + M \times (3 - x_{ik} - x_{jk} - b_{ij}) \quad (5.3)$$

$$\forall p_k \in P, \forall\ o_i, o_j \in V, (o_i, o_j), (o_j, o_i) \notin A, E(o_j) \leq S(o_i) + M \times (2 - x_{ik} - x_{jk} + b_{ij}) \quad (5.4)$$

The cost of synchronization is taken into account as follows. A synchronization cost is introduced in the computation of the start time of an operation o_j, if it has a predecessor o_i that is allocated to a different core and if its start time is the earliest among the successors of o_i that are allocated to the same core as the operation o_j. $sync_{ijk}$ is a binary variable which indicates whether synchronization is needed between o_i and o_j if o_j is allocated to p_k. Therefore, $sync_{ijk} = 1$ iff $\alpha(o_j) = p$ and $\alpha(o_i) \neq p$ and $S(o_j) = \max_{o_{j'} \in succ(o_i):\ \alpha(o_{j'})=p}(S(o_{j'}))$. Expressions 5.5 and 5.6 capture this constraint. V_{ik} is a binary variable that is set to one only if $\alpha(o_i) \neq p$. It is used to define for which cores a synchronization is needed between o_i and its successors. In other words, if a successor is allocated to the same core as o_i, no synchronization is needed. Expressions 5.7 and 5.8 capture this constraint. Variable Q_{ik} denotes the earliest start time among the start times of all the successors of o_i that are allocated to processor p_k. It is computed using expressions 5.9 and 5.10.

$$\forall o_i \in V, \sum_{\forall p \in P, \forall o_j \in pred(o_i)} sync_{ijk} = Q_{ik} \quad (5.5)$$

$$\forall o_i \in V, \forall o_j \in succ(o_i), sync_{ijk} \leq x_{jp} : \forall o_i \in V \quad (5.6)$$

$$\forall o_i \in V, \forall o_j \in succ(o_i), V_{ik} \geq x_{jp} - x_{ik} : \forall o_i \in V \quad (5.7)$$

$$\forall o_i \in V, V_{ik} \leq \sum_{\forall o_j \in succ(o_i)} (x_{jp} - x_{ik}) \quad (5.8)$$

$$\forall o_i \in V, \forall o_j \in succ(o_i), Q_{ik} \leq S(o_j) + M \times (1 - x_{jp}) \quad (5.9)$$

$$\forall o_i \in V, \forall o_j \in succ(o_i), Q_{ik} \geq S(o_j) - M \times (1 - sync_{ijk}) \quad (5.10)$$

The start time of each operation o_j is computed using expression 5.11. The synchronization cost is introduced taking into account the synchronizations with all the predecessors of o_j that are allocated to different cores.

$$\forall o_j \in V, \forall o_i \in pred(o_j), S(o_j) \geq \Big[E(o_i) + \sum_{\forall p_k \in P, \forall o_{i'} \in pred(o_j)} sync_{i'jk} \times synCost\Big] \quad (5.11)$$

The makespan is equal to the latest end time among the end times of all the operations as stated by expession 5.12

$$\forall o_i \in V, mkp \geq E(o_i) \quad (5.12)$$

Objective

The objective of this linear program is to minimize the makespan of the operation graph.

$$min(mkp) \quad (5.13)$$

5.1.3 Multi-core Scheduling Heuristic

Multi-core scheduling problems are known to be NP-hard resulting in exponential resolution times when exact algorithms are used. Heuristics have been extensively used in order to solve multi-core scheduling problems. In most situations they lead to results of good quality in practicle resolution times. In particular, list heuristics presented in Chapter 2 are widely used in the context of offline multi-core scheduling.

A variety of list multi-core scheduling heuristics exist in the literature and each heuristic may be suitable for some specific kinds of multi-core scheduling problems. We detail in this section a heuristic that we have chosen to apply on the operation graph $G(V, A)$ in order to minimize its makespan. Because of the number of fine-grained operations, and since the execution times and the dependence between the operations are known before runtime, it is more convenient to use an offline scheduling heuristic which has the advantage of introducing lower overhead than online scheduling heuristics. We use an offline scheduling heuristic similar to the one proposed in [81] which is a fast greedy algorithm whose cost function corresponds well to our minimization objective. In accordance with the principle of list scheduling heuristics, this heuristic is priority-based, i.e. it builds a list of operations that are ready to be scheduled, called candidate operations and selects one operation based on the evaluation of the cost function. We denote by ρ the cost function and call it the *schedule pressure*. It expresses the degree of criticality of scheduling an operation. The schedule pressure of an operation is computed using its flexibility and the penalty of scheduling which refers to the increase in the critical path resulting from scheduling an operation as stated by expression 5.14.

$$\rho = S(o_i) + C(o_i) + \overline{E}(o_i) - CP \quad (5.14)$$

The heuristic considers the different timing attributes of each operation o_i in order to compute a schedule that minimizes the makespan of the graph. It schedules the operations on the different cores iteratively and aims at minimizing the schedule pressure

of an operation on a specific core while taking into account the synchronization costs. The heuristic updates the set of candidate operations to be scheduled at each iteration. An operation is added to the set of candidate operations if it has no predecessor or if all its predecessors have already been scheduled. For each candidate operation, the schedule pressure is computed on each core and the operation is allocated to its best core, the one that minimizes the pressure. Then, a list of candidate operation-best core pairs is obtained. Finally, the operation with the largest pressure on its best core is selected and scheduled. Synchronization operations are added between the scheduled operation and all its predecessors that were allocated to different cores. The heuristic repeats this procedure and finally stops when all the operations have been scheduled. Algorithm 7 details the scheduling heuristic.

Complexity

The scheduling heuristic contains three nested loops. The outermost loop is executed until all the operations are scheduled. At each iteration, one operation is scheduled. Therefore, the outermost loop is executed n times where n is the number of operation in the operation graph. In the inner loops, the heuristic attempts to schedule all the ready operations on all the available cores. As such, the inner loops execute in $\mathcal{O}(nm)$, where m is the number of cores. From the foregoing, the complexity of the heuristic is evaluated to $\mathcal{O}(mn^2)$.

5.2 Scheduling of FMU Co-simulation under Real-time Constraints

In this section, we are interested in multi-core scheduling of FMU co-simulation under real-time constraints. We consider FMU co-simulation in the context of HiL consisting of a simulated component and a real component. Also, we consider that the real-time constraints that are applied by the real component have been propagated through the operation graph as described in Section 4.4. Therefore, the aim here consists in scheduling the operations of the operations graph on a multi-core architecture, such that these constraints are satisfied. Note that in contrast to the previous section, how much the execution is sped up is not of a crucial importance here as long as the real-time constraints are respected. Hereafter, we present an ILP formulation and a heuristic for scheduling operation graphs under real-time constraints.

5.2.1 Problem Formulation

The problem of scheduling FMU co-simulation under real-time constraints can be considered as a satisfaction problem instead of an optimization problem. In fact, in its basic form, the problem does not involve an objective function to be optimized, the goal being to ensure the real-time constraints are satisfied. More precisely, the problem consists in scheduling the operations of the operation graph such that each operation starts its execution no earlier than its release date and finishes its execution by its deadline

Algorithm 7: Multi-core scheduling heuristic

Input : Operation graph $G(V, A)$, set of cores P;
Output : Schedule of operations $o_i \in V$ on cores $p_k \in P$;
Set O the set of operations without predecessors;
Set *sync* the cost of one synchronization operation;
Set $L_k : p_k \in P$ the length of schedule of core p_k;
foreach $p_k \in P$ **do**
  $L_k \leftarrow 0$;
while $O \neq \emptyset$ **do**
  foreach $o_i \in O$ **do**
    $\rho \leftarrow \infty$; // Initialize the schedule pressure of o_i
    $S(o_i) \leftarrow \max_{o_j \in pred(o_i)}(E(o_j))$;
    foreach $p_k \in P$ **do**
      $syncCost \leftarrow 0$;
      $S(o_i) \leftarrow \max(S(o_i), L_k)$; // Start time of o_i if executed on p_k
      foreach $o_j \in pred(o_i)$ **do**
        if o_j *is scheduled on a core* $p_{k'} \neq p_k$ **then**
          $syncCost \leftarrow syncCost + sync$;
      $S(o_i) \leftarrow S(o_i) + syncCost$;
      $E(o_i) \leftarrow S(o_i) + C(o_i)$;
      $\rho' \leftarrow S(o_i) + C(o_i) + \overline{E}(o_i) - CP$; // Cost of o_i if executed on p_k
      if $\rho' < \rho$ **then**
        Set $\rho \leftarrow \rho'$;
        $BestCore(o_i) \leftarrow p_k$;
  Find $o_{i'}$ with maximal cost ρ in O;
  Schedule $o_{i'}$ on its core $BestCore(o_{i'})$;
  $p_{best} \leftarrow$ BestCore($o_{i'}$);
  $L_{best} \leftarrow E(o_{i'})$;
  Remove $o_{i'}$ from the set O;
  Add to the set O all successors of $o_{i'}$ for which all predecessors are already scheduled;

date. There are other constraints that are common with the problem of scheduling for co-simulation acceleration, namely, respecting the partial order of the operation graph. Also, the cost of inter-core synchronization is taken into account in computing the schedule in the same way. Finally, the computed schedule is non preemptive. The scheduling of FMU co-simulation under real-time constraints can be stated as a satisfaction problem as follows:

Input	Operation graph $G(V, A)$
Output	Offline Schedule of operations on a multi-core processor
Find	Allocation of operations to cores: $\alpha : V \rightarrow P$ Assignment of start times to operations: $\beta : V \times P \rightarrow \mathbb{N}$
Subject to	Precedence constraints: $\forall (o_i, o_j) \in A, S(o_j) \geq E(o_i)$ Release constraints: $\forall o_i \in V, S(o_i) \geq R(o_i)$ Deadline constraints: $\forall o_i \in V, E(o_i) \leq D(o_i)$

5.2.2 Accounting for Dependence in Real-time Scheduling

The model of computation for (co-)simulation is close to the synchronous paradigm [82, 83]. In this paradigm, a program evolves according to a sequence of ticks of logical time at which computations are considered to produce their results instantaneously. The propagation of the release and deadline constraints presented in Chapter 4 follows this model of computation. However, when real-time constraints are involved, co-simulation becomes incompatible with the synchronous paradigm. In fact, each operation takes a certain execution time to run and, therefore, cannot produce the result instantaneously. In order to proceed to scheduling the operation graph, it is necessary to account for the execution times of the operations.

We adopt an approach similar to the one proposed in [84] to modify the release and deadline dates assigned to each operation in order to account for execution times. This modification is needed given that:

- The execution of an operation can start no earlier than its release but also only after the execution of all its predecessors is finished.
- The execution of an operation must be finished before its deadline and also be finished so that the execution of its successors can be finished before their deadlines.

Let o_i and o_j be two operations such that $o_j \in pred(o_i)$. For a given schedule of the operation graph to be valid, the relations $S(o_i) \geq R(o_i)$ and $S(o_i) \geq E(o_j)$ must be satisfied. Therefore, a new release date for o_i can be computed using expression 5.15.

$$R(o_i) = \max(R(o_i), \max_{o_j \in pred(o_i)}(E(o_j))) \tag{5.15}$$

Consider now two operations o_i and o_j such that $o_j \in succ(o_i)$. For the operation graph to be schedulable, the relations $E(o_i) \leq D(o_i)$ and $E(o_i) \leq D(o_j) - C(o_j)$ must be satisfied. In fact, $D(o_j) - C(o_j)$ represents the latest time to start the execution of the successor o_j such that its deadline can be met. Therefore, a new deadline date of o_i can be computed using expression 5.16.

$$D(o_i) = \min(D(o_i), \min_{o_j \in succ(o_i)}(D(o_j) - C(o_j))) \tag{5.16}$$

5.2.3 Scheduling Interval

In offline scheduling, the schedule is computed over an interval of time. This schedule is then executed repetitively. For the acceleration of co-simulation, we have seen that the length of the schedule interval is equal to the hyperstep. For co-simulation under real-time constraints, a natural approach is to apply techniques that are used for classical real-time systems (such co-simulation is considered a real-time system after all). For this, we need first to represent the operation graph with a model that involves the parameters that are usually used for classical real-time systems. In particular, we need to define a relative deadline and a period for each operation. Note that so far, we have only spoken about sampling periods of data exchange between the real and the simulated component. Although related to the sampling periods, the periods that we seek to define for each operation are different and correspond to task periods that are found in classical real-time systems. We handle this requirement as follows. We consider that every operation that appears in the hyperperiod pattern of the operation graph is a distinct operation. In other words, occurrences of one operation are not regarded as repetitions of a single operation. Accordingly, we consider that the operation graph is *mono-period*, i.e. all the operations have the same period. The value of this period is equal to the hyperperiod (see Definition4.4.3). The relative deadline of each operation can then be defined as the duration between its release and deadline.

In the real-time literature, we find contributions regarding the schedule interval targeting different kinds of real-time tasks, schedulers, and architectures. For instance, in [85], the authors study synchronous task systems, i.e. where the release dates of all tasks are equal to zero, with constrained deadlines, i.e. where the relative deadline of each task is less or equal to its period. They show that the schedule of such task system on uniform multiprocessors reaches a cyclic behavior after one hyperperiod. The length of the schedule interval for co-simulation under real-time constraints cannot be chosen in a straightforward manner to be equal to the hyperperiod. This is because the operation graph features *arbitrary deadlines*, i.e. relative deadlines that are greater than the periods which may result in *hyperperiod spill* [86]. The latter refers to operations that are not scheduled in their hyperperiod and spill over the next one. In [87], a schedule interval is given for preemptive scheduling of tasks with arbitrary deadlines on uniform multiprocessors. A more general result is given in [88] taking into account different constraints (mutual exclusion, precedence constraints, non-preemptive tasks, etc.) for uniprocessor and multiprocessor scheduling. The authors give an upper bound for the schedule interval when a *deterministic memoryless* scheduler is used. A scheduler is deterministic and memoryless if and only if, when building the schedule, the scheduling decision is the same for any identical configuration encountered. The given bound is:

$$\prod_{i=1}^{n}((\max(O_i + D_i - T_i, 0) + 1)) \times H \tag{5.17}$$

where n is the number of tasks, O_i, D_i, and T_i are the offset (release date), relative deadline, and period of task τ_i respectively, and H is the hyperperiod. This result is applicable to non-preemptive scheduling with arbitrary deadlines which is the case in our problem. However, the proposed bound is intractable, i.e. as the size of the operation graph grows and depending on the parameters of the operations, it results in very large schedule intervals and we cannot guarantee to compute the schedule within an acceptable time. Therefore, we choose to start with a schedule interval whose length is equal to the hyperperiod and iteratively increase it if we cannot determine the schedulability of the operation graph.

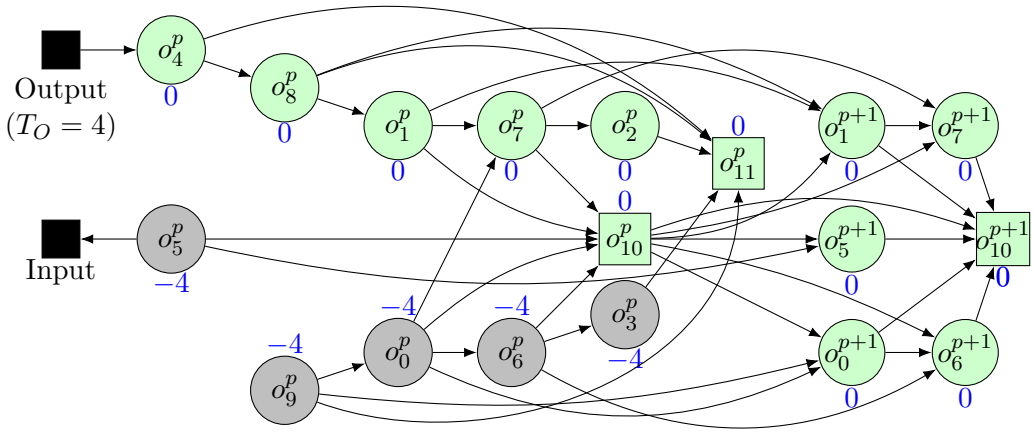
As shown in Section 4.4, the propagation of the real-time constraints in an operation graph may lead to assigning negative release dates to some operations. This means that such operations can be executed before launching the co-simulation under real-time constraints. Therefore, they are not scheduled and are removed from the first repetition of the operation graph. See, for example, the operations colored in gray in Figure 5.1a. However, since these operations must appear in the subsequent repetitions of the schedule of the operation graph, we have to add the repetitions of these operations that belong to the second repetition of the operation graph to the operation graph pattern that is scheduled. Figure 5.1 shows an example of the operations that belong to the operation graph pattern that is scheduled. The operations that are colored in green belong to this pattern. It can be seen that operations that are assigned negative release dates are not scheduled in the first repetition of the operation graph. Instead, their occurrences that belong to the second repetition of the operation graph are added to the pattern.

5.2.4 Resolution using Linear Programming

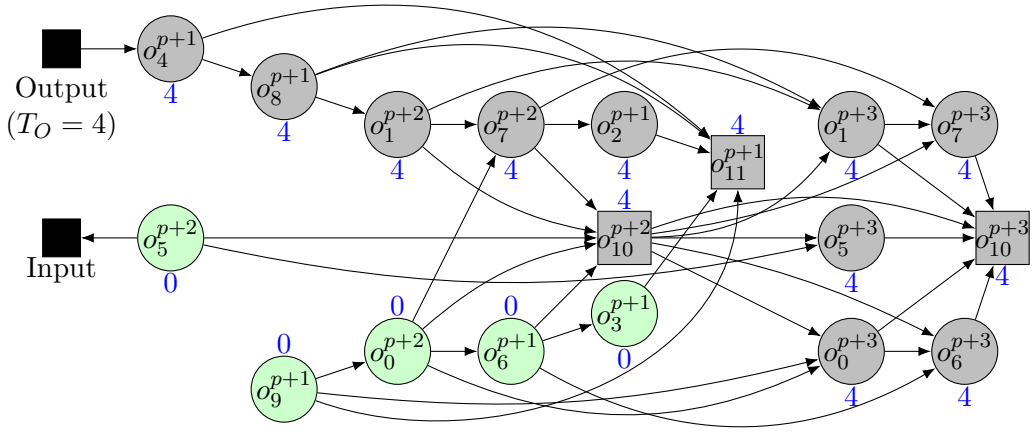
The ILP formulation that we propose is in most part similar the ILP formulation that we proposed for co-simulation acceleration. The main differences consist, first, in adding the inequalities that express the real-time constraints. Second, we need not set an objective function since the real-time scheduling consists in a satisfaction problem.

The ILP formulation for multi-core scheduling of co-simulation under real-time constraints is given below. We do not explain most of the constraints since these are common with the ILP formulation given in Section 5.1.2 where we explained them in detail. We explain only the following additional constraints. The start date of every operation must be at the earliest equal to its release date. Expression 5.20 captures this constraint. The deadline date of every operation is the latest time before which the operation has to finish its execution. Expression 5.21 specifies this constraint.

$$\forall\ o_i \in V, \sum_{p_k \in P} x_{ik} = 1 \tag{5.18}$$



(a) First repetition of the operation graph



(b) Second repetition of the operation graph

Figure 5.1: Example of operation graph pattern for real-time scheduling

$$\forall o_i \in V, E(o_i) = S(o_i) + C(o_i) \tag{5.19}$$

$$\forall\ o_i \in V, S(o_i) \geq R(o_i) \tag{5.20}$$

$$\forall\ o_i \in V, E(o_i) \leq D(o_i) \tag{5.21}$$

$$\forall p_k \in P, \forall\ o_i, o_j \in V, (o_i, o_j), (o_j, o_i) \notin A, E(o_i) \leq S(o_j) + M \times (3 - x_{ik} - x_{jk} - b_{ij}) \tag{5.22}$$

$$\forall p_k \in P, \forall\ o_i, o_j \in V, (o_i, o_j), (o_j, o_i) \notin A, E(o_j) \leq S(o_i) + M \times (2 - x_{ik} - x_{jk} + b_{ij}) \tag{5.23}$$

$$\forall o_i \in V, \sum_{\forall p \in P, \forall o_j \in pred(o_i)} sync_{ijk} = Q_{ik} \tag{5.24}$$

$$\forall o_i \in V, \forall o_j \in succ(o_i), sync_{ijk} \leq x_{jp} : \forall o_i \in V \tag{5.25}$$

$$\forall o_i \in V, \forall o_j \in succ(o_i), V_{ik} \geq x_{jp} - x_{ik} : \forall o_i \in V \tag{5.26}$$

$$\forall o_i \in V, V_{ik} \leq \sum_{\forall o_j \in succ(o_i)} (x_{jp} - x_{ik}) \tag{5.27}$$

$$\forall o_i \in V, \forall o_j \in succ(o_i), Q_{ik} \leq S(o_j) + M \times (1 - x_{jp}) \tag{5.28}$$

$$\forall o_i \in V, \forall o_j \in succ(o_i), Q_{ik} \geq S(o_j) - M \times (1 - sync_{ijk}) \tag{5.29}$$

$$\forall o_j \in V, \forall o_i \in pred(o_j), S(o_j) \geq \left[E(o_i) + \sum_{\forall p_k \in P, \forall o_{i'} \in pred(o_j)} sync_{i'jk} \times synCost\right] \tag{5.30}$$

5.2.5 Multi-core Scheduling Heuristic

Removing the objective function from the scheduling problem might indicate that the complexity of the scheduling problem is reduced in the real-time case compared to the acceleration case. However, adding the strict release and deadline constraints adds to the complexity of the problem which remains an NP-Hard problem that is equivalent to the bin packing problem.

In the following, we propose a heuristic for scheduling operation graphs representing FMU co-simulations under real-time constraints. There are some considerations that are common with the scheduling problem for co-simulation acceleration. Mainly, we propose an offline heuristic which we consider to be more suitable given the fine granularity of the operations and since information about the execution times of the operations and the dependence between them is available before runtime.

We propose to adapt the scheduling heuristic that we use for the acceleration of FMU co-simulation. In particular, we modify the computation of the scheduling priority such that the criticality of a given operation expresses how close it is to miss its deadline if scheduled on a specific processor. The priority of an operation is a dynamic priority as its computation depends on the partial scheduling solution that has already been computed. This priority is given by expression 5.31.

$$\rho_{i,k} = D(o_i) - E(o_i) \tag{5.31}$$

Where $\rho_{i,k}$ and $E_j(o_i)$ are the scheduling priority and the end date of operation o_i respectively, computed when the latter is scheduled on core p_k.

The proposed heuristic is a list scheduling heuristic. It builds the multi-core schedule iteratively. At each iteration, a list of candidate operations is constructed. An operation is added to the list of candidate operation if all its predecessors have been scheduled. The heuristic computes the priority for each candidate operation on every core and selects the core for the which the priority is maximized. After that, a list of operation-best core pairs is obtained. The heuristic selects from this list the operation whose priority is the smallest among all operation in the list. Synchronization operations are added between the scheduled operation and all its predecessors that were allocated to different cores. The heuristic repeats this procedure and finally stops when all the operations have been scheduled. Algorithm 8 lists the proposed real-time multi-core scheduling heuristic.

Algorithm 8: Multi-core scheduling heuristic

Input : Operation graph $G(V, A)$, set of cores P;
Output : Schedule of operations $o_i \in V$ on cores $p_k \in P$;
Set O the set of operations without predecessors;
Set *sync* the cost of one synchronization operation;
Set $L_k : p_k \in P$ the length of schedule of core p_k;
foreach $p_k \in P$ **do**
  $L_k \leftarrow 0$;
while $O \neq \emptyset$ **do**
  foreach $o_i \in O$ **do**
    $\rho \leftarrow \infty$; // Initialize the priority of o_i
    $S(o_i) \leftarrow \max(R(o_i), \max_{o_j \in pred(o_i)}(E(o_j)))$;
    foreach $p_k \in P$ **do**
      $syncCost \leftarrow 0$;
      $S(o_i) \leftarrow \max(S(o_i), L_k)$; // Start time of o_i if executed on p_k
      foreach $o_j \in pred(o_i)$ **do**
        if o_j *is scheduled on a core* $p_{k'} \neq p_k$ **then**
          $syncCost \leftarrow syncCost + sync$;
      $S(o_i) \leftarrow S(o_i) + syncCost$;
      $E(o_i) \leftarrow S(o_i) + C(o_i)$;
      $\rho' \leftarrow D(o_i) - E(o_i)$; // priority of o_i if executed on p_k
      if $\rho' > \rho$ **then**
        Set $\rho \leftarrow \rho'$;
        Set $BestCore(o_i) \leftarrow p_k$;
  Find $o_{i'}$ with the smallest priority ρ in O;
  Schedule $o_{i'}$ on its core $BestCore(o_{i'})$;
  $p_{best} \leftarrow BestCore(o_{i'})$;
  $L_{best} \leftarrow E(o_{i'})$;
  Remove $o_{i'}$ from the set O;
  Add to the set O all successors of $o_{i'}$ for which all predecessors are already scheduled;

Complexity

Algorithms 7 and 8 are two variations of the same list scheduling heuristic and have the same complexity as they consist of the same steps. Therefore, the complexity of Algorithm 8 is $\mathcal{O}(mn^2)$.

5.3 Code Generation

In this section, we describe how the FMU co-simulation code is generated based on the schedule tables produced by the proposed scheduling algorithms. Note that while the schedule tables are produced using different algorithms, the code generation is done in a similar way for both acceleration of co-simulation and co-simulation under real-time constraints. Since the FMU co-simulation is intended to be executed on multi-core desktop computers running general purpose or real-time operating systems, the implementation is achieved using *native* threads. Such threads consist in threads that are provided by the operating system in contrast to threads that are related to a specific programming language and/or rely on a specific runtime library.

In the generated code, as many threads are created as there are cores. Each thread is responsible for the execution of the schedule of one core. Therefore, each thread reads from the schedule table of its corresponding core and executes the operations that are saved in this table. These operations can be computational operations, i.e. input, output, and state operations, or synchronization operations. The synchronization operations are implemented using semaphores provided by the operating system. They are of two types: *signal* and *wait* operations. The execution of a signal operation by a thread consists in signaling the corresponding semaphore. The execution of a wait operation by a thread consists in block waiting for the corresponding semaphore. Each thread executes its associated schedule table repeatedly, and thus executes FMU operations and synchronizes with the other threads. Hereafter, we refer to these threads as *schedule threads*.

The orchestration of the co-simulation is ensured by a *master* thread which runs the FMI master algorithm. The master thread creates and launches the schedule threads. During the execution, the master thread and the schedule threads are synchronized at fixed points. First, the master thread signals to the schedule threads the start of the co-simulation which launches their execution. Each thread starts, then, the execution of its associated schedule table as described in the previous paragraph. When it finishes the execution of the whole schedule table, its signals this to the master thread and waits for a new signaling from it. The master thread block waits until all the schedule threads signal that they finished the execution of their respective schedule tables. Then, the master thread launches a new iteration by signaling to the schedule threads to start executing their corresponding schedule tables again. This process is repeated until the desired simulation time is reached. Figure 5.2 shows an example of the execution of the generate code for an FMU co-simulation on a two-core processor.

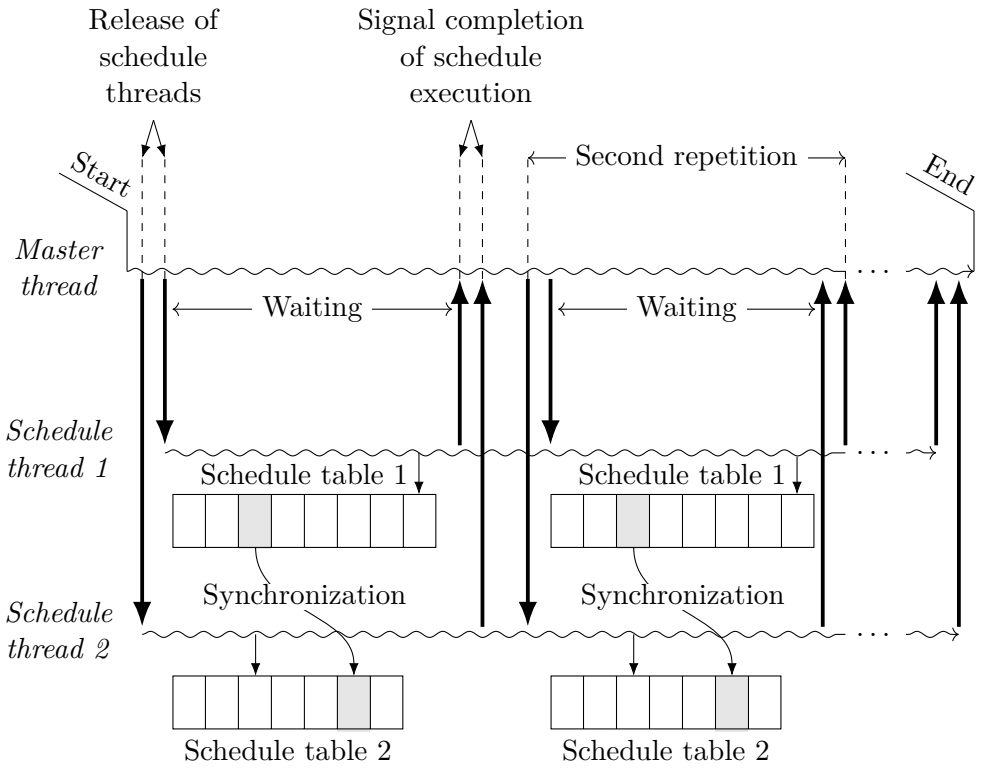


Figure 5.2: Illustration of the execution of generated code.

6 Evaluation

Contents

In this chapter, we evaluate our proposed approach. We start by describing a method for randomly generating benchmark operation graphs. Then, we present the evaluation of the performances of the acyclic orientation and the scheduling heuristics. Finally, we give runtime performance and numerical accuracy results obtained by applying our approach on an industrial use case.

6.1 Random Generator of Operation Graphs

Due to the difficulty in acquiring enough industrial FMU co-simulation applications for assessing our approach, we had to use a random generator of FMU dependence graphs. The generator creates the graphs and characterizes them with attributes. In the case

of co-simulation under real-time constraints, the generator generates real components and connects them to the operation graph.

6.1.1 Generation of Random Operation Graphs

The random generator that we have implemented is inspired by the random generator presented in [89]. However, it differs from this work in that the generation is done in two stages. First, we generate the different FMUs of the co-simulation and their internal structures. Second, we generate the dependence graph by creating inter-FMU dependence in such a way that the resulting operation graph is a DAG. The proposed generator is based on a technique of assignment of operations to levels. The level of an operation is the number of operations on the longest path from a source operation to this operation. The dependence graph can then be visualized on a grid of levels as depicted in Figure 6.1. The generator uses the following parameters:

- The graph size n: the number of operations;
- The number of FMUs m;
- The graph height h: the maximum number of levels in the graph;
- The graph width w: the maximum number of nodes on one level.

Note that parameters n and m are related. In other words, for a given size of a graph n, an adequate number of FMUs m has to be chosen.

The generation of the dependence graph is performed as follows:

- **Input:** Size of the graph n, number of FMUs m, height of the graph h, and width of the graph w.
- **Step 1:** Randomly distribute the n operations across the m FMUs. Given the number of operations of each FMU, we randomly determine the number of its input operations and the number of its output operations. Every FMU has one state operation.
- **Step 2:** Randomly generate the intra-FMU arcs. This step is controlled by two parameters. The number of arcs to generate and the number of NDF outputs of the FMU. These outputs are not considered when randomly generating the arcs.
- **Step 3:** Randomly assign the operations to the grid levels. This step is performed by assigning output operations and then input operations repeatedly.
  1. Assign all NDF operations to level 0 of the grid.
  2. Randomly assign remaining output operations to even levels $(2, 4, \ldots, h-3)$ of the grid.

3. Assign the input operations to the odd levels $(1, 3, \ldots, h-4)$ of the grid such that any input operation o_i that is connected to an output operations o_j (intra-FMU dependence) is assigned to the level preceding the level to which o_j has been assigned.
4. Assign the remaining input operations (each of which is not connected with any output operation) to the level $h-2$ of the grid. These operations will be connected only with the state operations of their respective FMUs.
5. Add the state operations to the last level of the grid.

- **Step 4:** Create the arcs of the dependence graph. At this step, we randomly generate inter-FMU dependence. For each operation o_i on the level lvl of the grid, we randomly select an output operation o_j from the preceding level $lvl-1$ and which belongs to a different FMU than o_i. We create an arc from o_j to o_i. If no such output operation is found at level $lvl-1$, we select randomly an output operation from any level $lvl' < lvl-1$ and connect it with the operation o_i. Finally the arcs from input and output operations to state operations are created.

Figure 6.1 illustrates the steps of our proposed random operation graph generator.

In the case of co-simulation under real-time constraints, the following additional steps are performed:

- **Step 5:** Create the real component and randomly generate the numbers of its inputs and outputs. These numbers are generated taking into account the size of the operation graph n.
- **Step 6:** Randomly connect the inputs (resp. outputs) of the real component with outputs (resp. inputs) of the operation graph. This step is performed in such a way that the condition stated in Theorem 4.4.1 is satisfied.

6.1.2 Random Operation Graph Characterization

In addition to random generation of the dependence graph structure, we need to generate the attributes of the graph. In particular, the following attributes are generated by our random generator:

- Communication step sizes of the FMUs: A range for the values of the communication step sizes is specified. The generator randomly assigns a communication step size within this range to every FMU.
- Execution times of the operations: Different ranges of the execution times are specified for input, output, and state operations. Execution times are generated randomly in such a way that state operations have longer execution times than output and input operations.

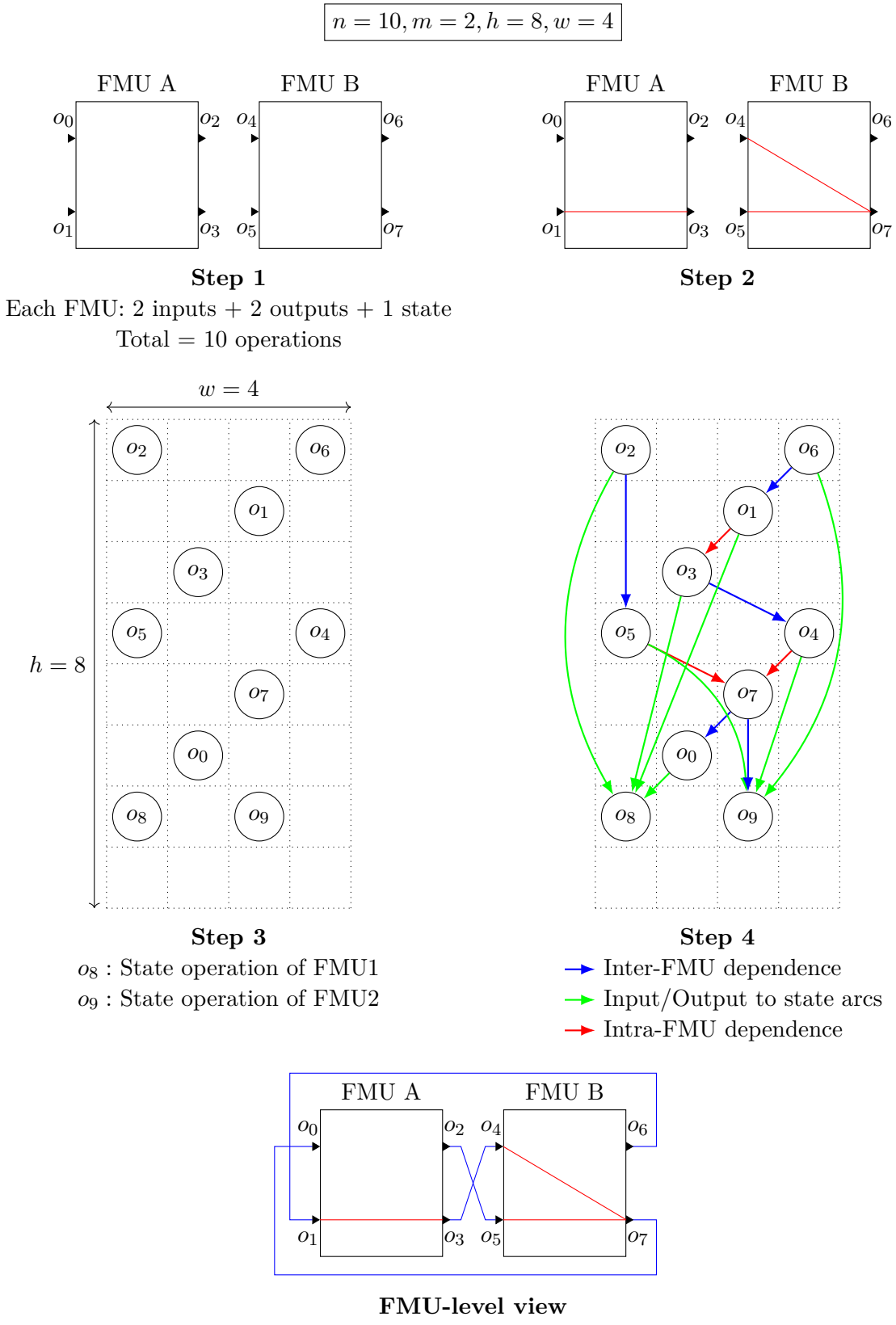


Figure 6.1: Random generation of an operation graph.

- Sampling periods of the real component: In the case of co-simulation under real-time constraints, each input (resp. output) of the real component is assigned a sampling period that is a multiple of the communication step size of the output (resp. input) operation that is connected with it.

6.2 Performances of the Algorithms

We have carried out different tests in order to evaluate our proposed approach. For both the acyclic orientation and the scheduling for acceleration, we compared the execution time of our proposed heuristic with the execution time of the ILP, and the value of the objective function of the heuristic with the value of the objective function of the ILP. For real-time scheduling we compared the execution time of our proposed heuristic with the execution time of the ILP, and evaluated the schedulability rate of our proposed heuristic. For ILP resolution, we used three solvers: lpsolve [90], Gurobi [91], and CPLEX [92]. With lpsolve, we were only able to solve small instances of the scheduling problem. Gurobi was much more efficient but we obtained the best performance using CPLEX. Therefore, the results presented hereafter were obtained using CPLEX. Tests were performed on a desktop computer with a 6-core Intel Xeon processor running at 2.7 GH with 16GB RAM.

6.2.1 Acyclic Orientation Algorithms

Comparison of Execution Times

In order to compare the execution time of our acyclic orientation heuristic with the execution time of the acyclic orientation ILP, we have generated 200 random operation graphs of different sizes between five and 10000. We considered 10000 as the maximum size of the operation graph because it corresponds to the size of typical large industrial applications.

We executed the acyclic orientation heuristic and ILP on all of the generated random graphs and measured the elapsed time between the start and the end of the execution. For the ILP, the execution is stopped if the optimal solution is not found within two days. The obtained execution times are shown on a logarithmic scale in Figure 6.2. The acyclic orientation ILP cannot be resolved in practical times when the size of the operation graph exceeds 250. When The number of operations is less than 250 the ILP finds the optimal solution in reasonable times, except for two graphs. In addition, we observe that an increase in the graph size does not always result in an increase in the execution time. This can be explained by the fact that other factors impact the speed of resolution, e.g. number of conflict edges. Still, it is important to notice that the application of the acyclic orientation ILP is limited to relatively small graphs. On the other hand, the acyclic orientation heuristic produces results in practical execution times even for very large operation graphs (10000).

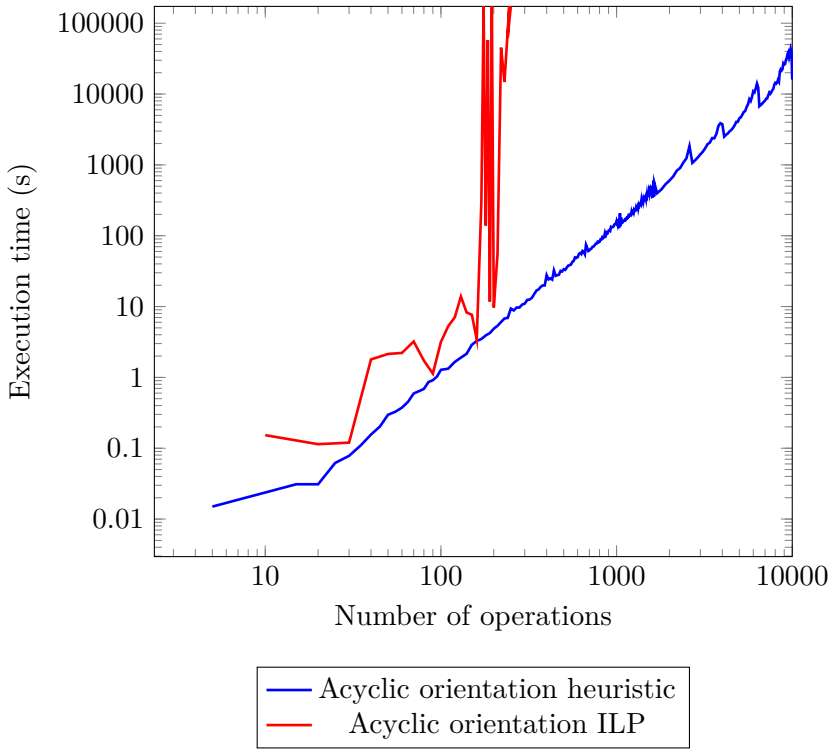


Figure 6.2: Comparison of the execution times of the acyclic orientation algorithms.

Comparison of Critical Path Lengths

We compared the values of the critical path length obtained using the acyclic orientation heuristic and ILP. Tests were performed using the same set of operation graphs described in the previous section. However, we consider only graphs for which the ILP was able to return the optimal solution within the resolution time limit that we set, i.e. two days. Thus, we applied our proposed heuristic and ILP on 12 operation graphs of sizes between 20 and 240 and saved the obtained length of the critical path. Results are depicted in Figure 6.3. For most of the operation graphs, our acyclic orientation heuristic produces a length of the critical path that is equal to the length of the critical path produced by the acyclic orientation ILP. The heuristic returns a longer length of the critical path for three graphs but the gap is very small remaining below 8%.

6.2.2 Scheduling Algorithms for Co-simulation Acceleration

Comparison of Execution Times

Similarly to the acyclic orientation tests, we compared the execution time of the scheduling heuristic with the execution time of the scheduling ILP using 200 random operation graphs of different sizes between five and 10000. We set a two day limit for the resolution of the ILP. Tests were run for the scheduling problem with 2, 4, and 8 cores. Execution times were measured by fixing the number of cores and varying the number of operations

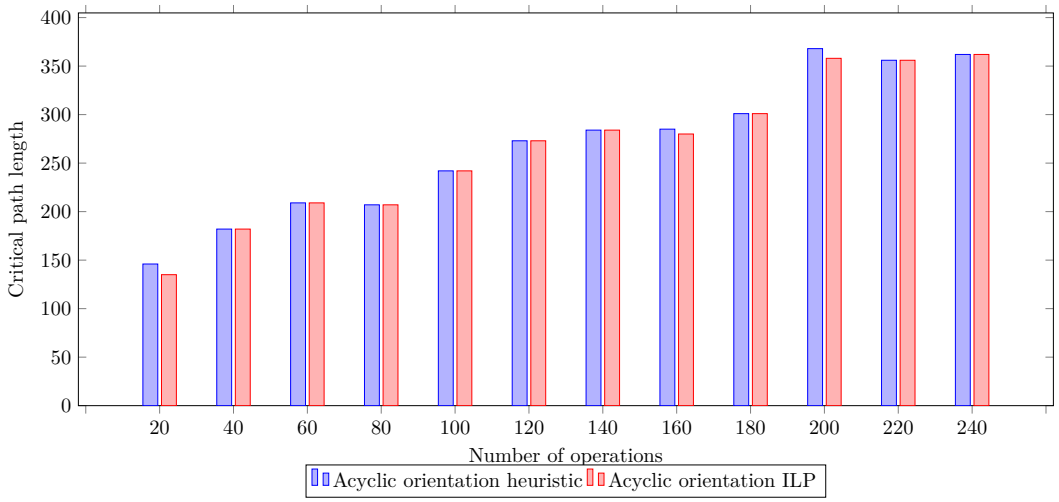


Figure 6.3: Comparison of the critical path length.

(graph size). The results are depicted for 2, 4, and 8 cores in Figure 6.4, Figure 6.5, and Figure 6.6 respectively. All results are plotted on a logarithmic scale. In these figures, we see that the execution time of the ILP resolution increases exponentially as the graph size increases, and only small instances are resolved within acceptable times. On the other hand, the scheduling heuristic is very fast and produces results in short times and even for very large graphs, the execution times remain within practical bounds.

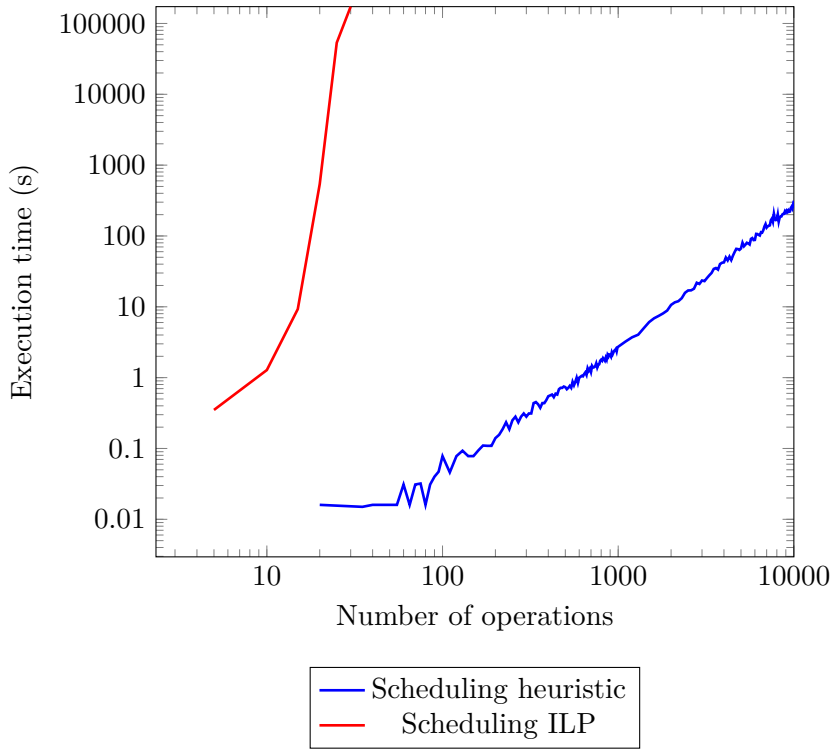


Figure 6.4: Comparison of the scheduling execution time for 2 cores.

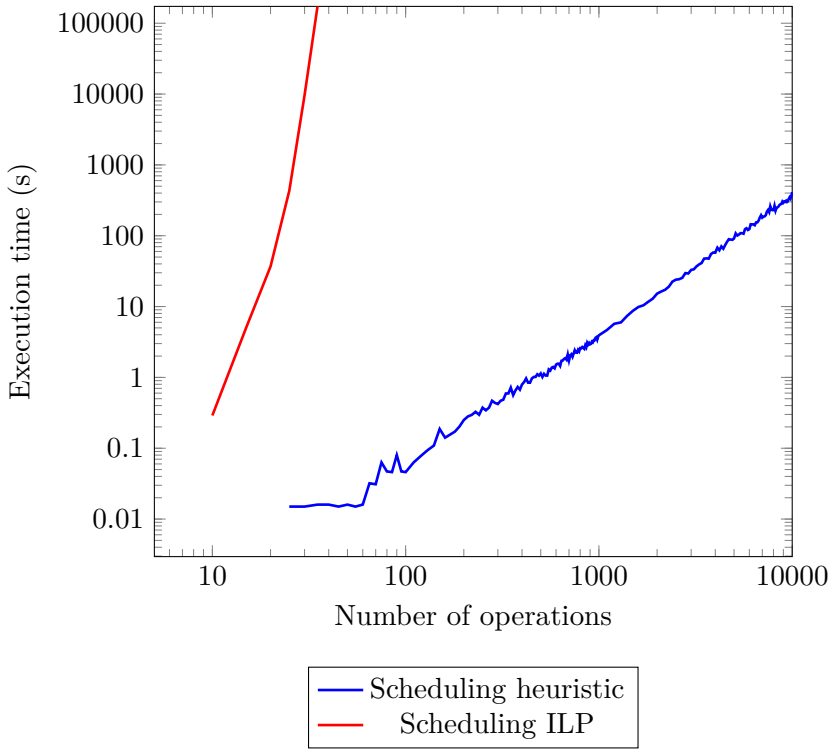


Figure 6.5: Comparison of the scheduling execution time for 4 cores.

Comparison of Makespans

We run tests to compare the value of the makespan obtained using the acyclic orientation heuristic and ILP. For these tests we have generated ten operation graphs of size $n = 15$. We have used graphs of size 15 because the ILP resolution returns the optimal solution in very short times which is not the case for large graphs. The graphs are different from each other because they are generated randomly which leads to different graph structures and execution times of the operations. We run the scheduling heuristic and ILP on these graphs to obtain the values of the makespan. Results are shown in Figure 6.7, Figure 6.8, and Figure6.9 for 2, 4, and 8 cores respectively. Overall, the results show that the scheduling heuristic produces a makespan which is very close to the makespan produced by the scheduling ILP. The gap between the heuristic and the ILP result lies between 0% and 16%. We notice that the gap is smaller when 4 or 8 cores are used than when 2 cores are used. In fact, when 2 cores are used the maximum gap is 16%, whereas when 4 or 8 cores are used the maximum gap is 6%. This shows that the scheduling heuristic performs better when the effective parallelism is increased. It can be explained by the fact that the scheduling heuristic attempts more allocation possibilities which leads to a better exploitation of the potential parallelism.

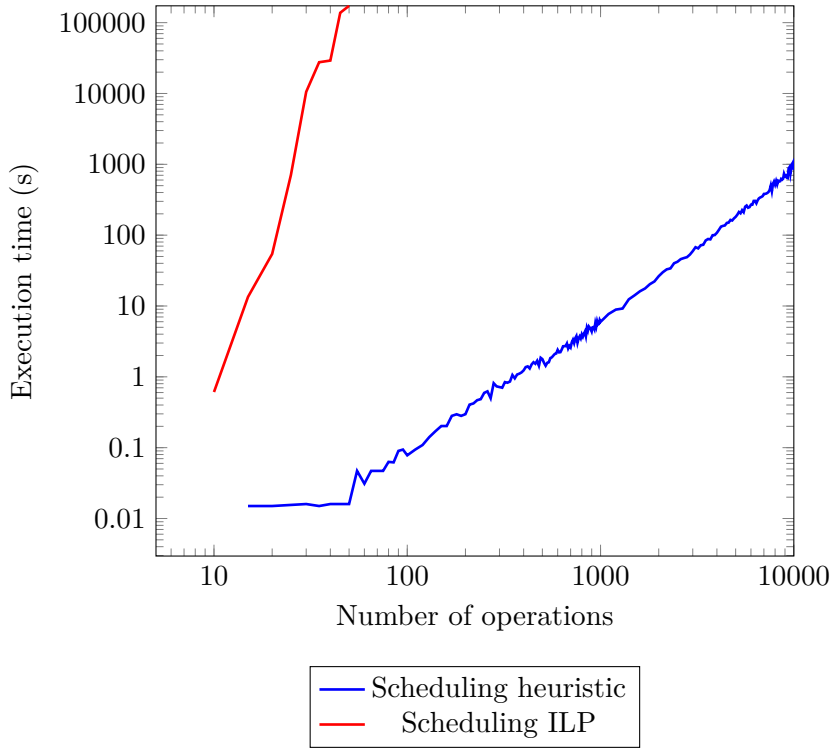


Figure 6.6: Comparison of the scheduling execution time for 8 cores.

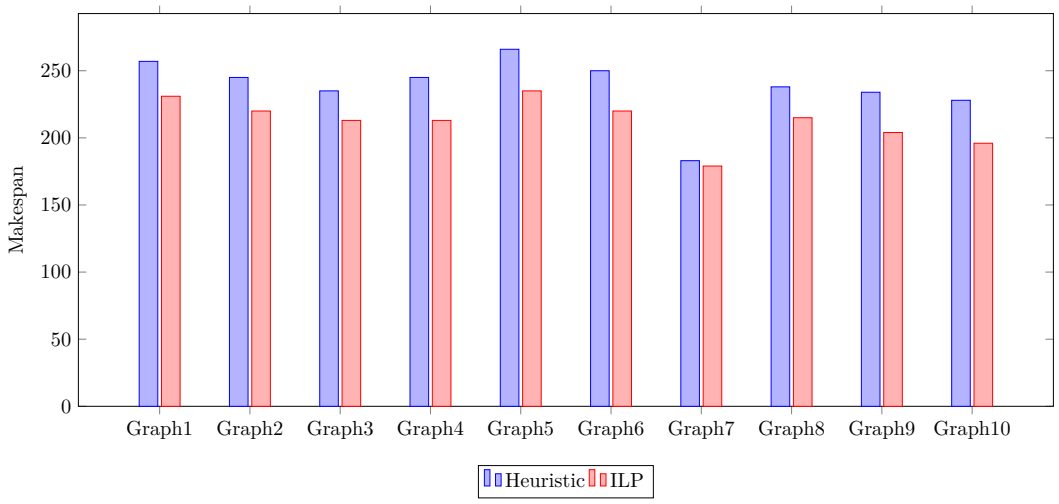


Figure 6.7: Comparison of the makespan for 2 cores.

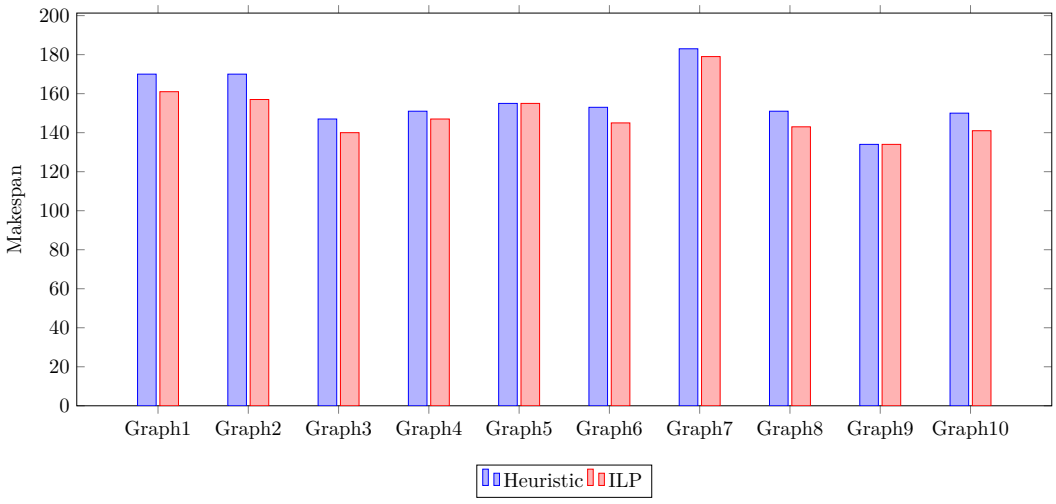


Figure 6.8: Comparison of the makespan for 4 cores.

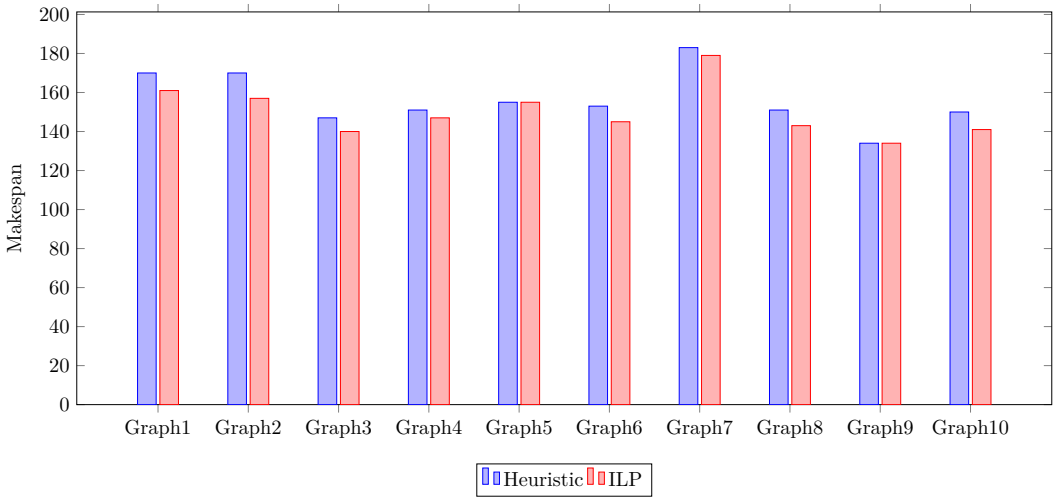


Figure 6.9: Comparison of the makespan for 8 cores.

6.2.3 Scheduling Algorithms for Co-simulation under Real-time Constraints

Comparison of Execution Times

We compared the execution times of the real-time scheduling ILP and heuristic. We ran these tests on a smaller set of operation graphs than the previous ones. In fact, the real-time scheduling heuristic and the scheduling heuristic are two variations of the same list scheduling algorithm and have the same complexity. Therefore, we tested the real-time scheduling heuristic on fewer graphs with a maximum number of operations of 1000. Tests were performed by fixing the number of cores and varying the number of operations. The obtained results are shown in Figure 6.10, Figure 6.11, and Figure 6.12 for 2, 4, and 8 cores respectively. While the real-time scheduling ILP is able to solve

larger graphs than the acceleration ILP within acceptable times, the execution time of the resolution still increases exponentially as the graph size increases. On the other hand, the real-time scheduling heuristic produces results in short times keeping the execution times within practical bounds similarly to the acceleration scheduling heuristic.

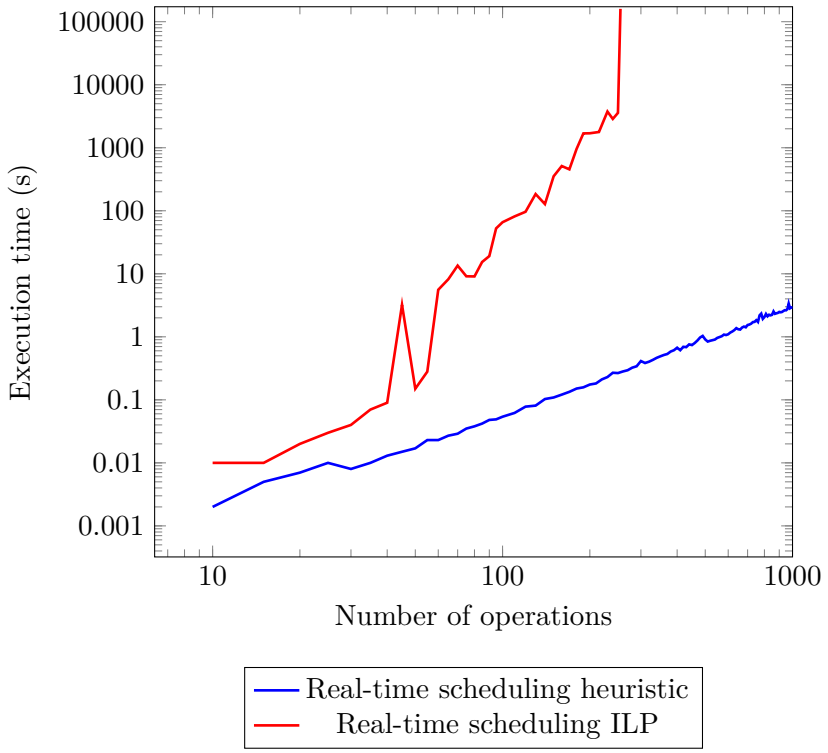


Figure 6.10: Comparison of the real-time scheduling execution time for 2 cores.

Schedulability

We run tests in order to measure the rate of schedulability of our proposed heuristic. Because the execution of the ILP takes long times, we limited these tests to operation graphs of small sizes. We generated five sets of operation graphs containing each 10 graphs of sizes between five and 50. We make sure that all the generated operation graphs are schedulable by applying the ILP. Then, we apply our heuristic and save the number of schedulable operation graphs for which the heuristic is able to find a solution. Figure 6.13 shows the obtained rates of schedulability for different numbers of cores. It can be seen the application of the heuristic results in interesting schedulability rates, especially when considering its very fast execution time compared to the ILP algorithm. As expected, the rate of schedulability increases as the number of cores increases. Indeed, the more there are cores, the more chances the heuristic has to find a solution.

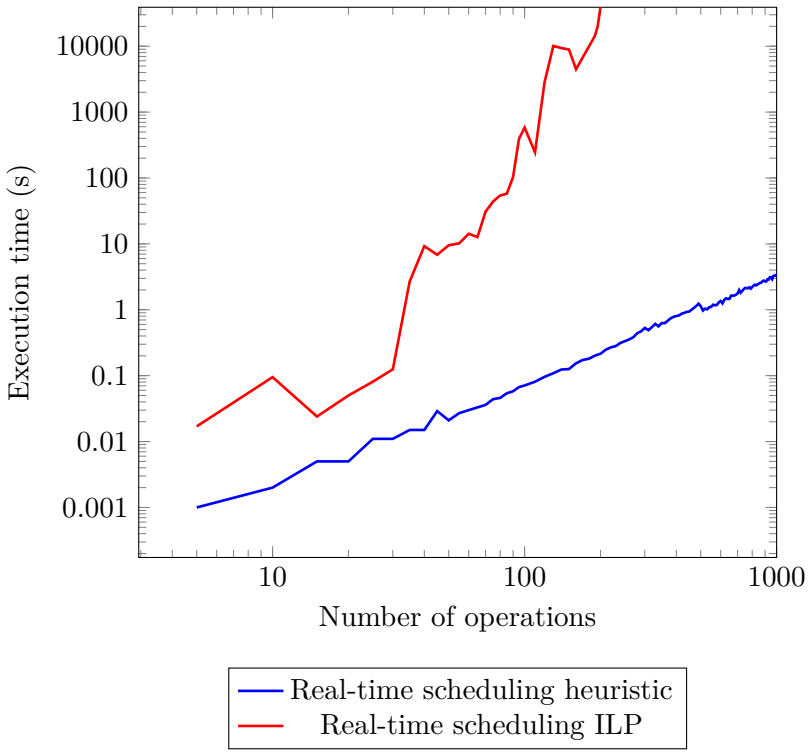


Figure 6.11: Comparison of the real-time scheduling execution time for 4 cores.

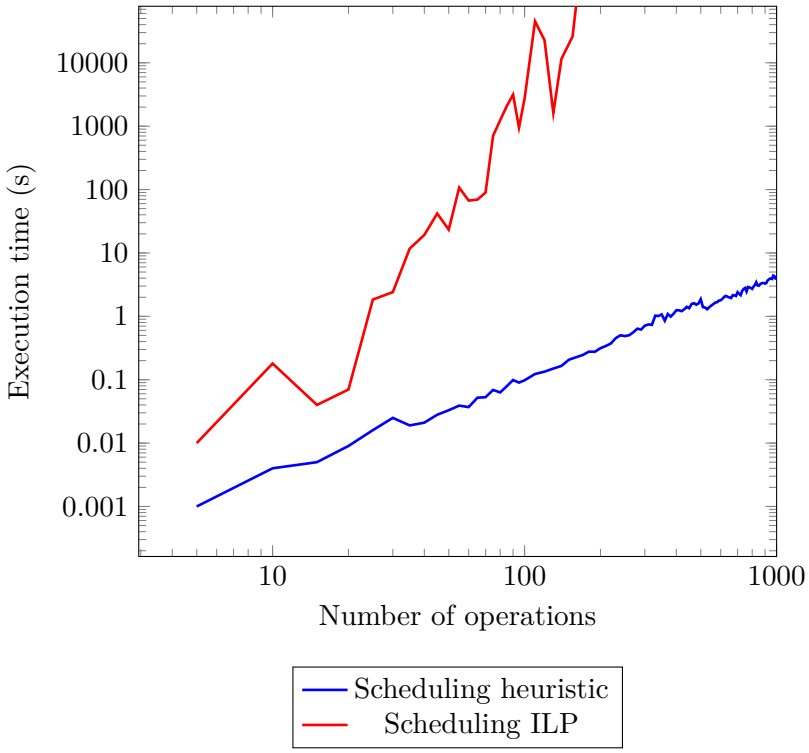


Figure 6.12: Comparison of the real-time scheduling execution time for 8 cores.

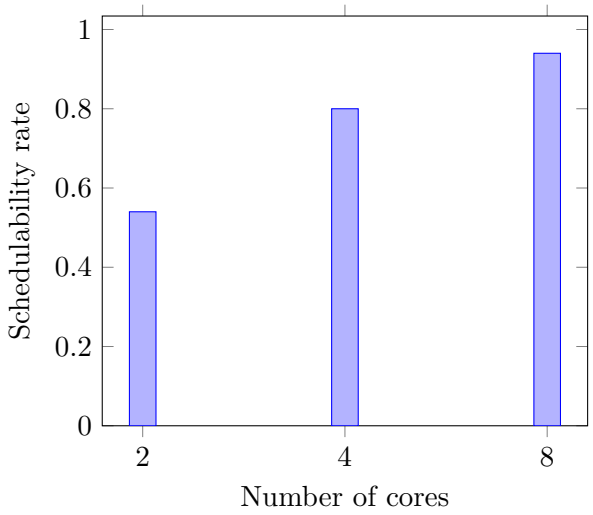


Figure 6.13: Rate of schedulable operation graphs.

6.3 Industrial Use Case

We tested our proposed approach for co-simulation acceleration on an industrial use case. Tests have been performed on a computer with an 8-core Intel core i7 processor running at 2.7 GH with 16GB RAM. In the rest of this section, we first give a description of the use case and then present the tests and the obtained results.

6.3.1 Use Case Description

Our use case consists in a Spark Ignition (SI) RENAULT F4RT engine co-simulation. It is a four-cylinder in line Port Fuel Injector (PFI) engine in which the engine displacement is 2000 cm^3. The air path is composed of a turbocharger with a mono-scroll turbine controlled by a waste-gate, an intake throttle and a downstream-compressor heat exchanger. See Figure 6.14. This co-simulation is composed of six FMUs: an FMU of the airpath, four FMUs of the four cylinders, and one FMU of the controller. The engine model was developed using ModEngine library [93]. ModEngine is a Modelica library that allows the modeling of a complete engine with diesel and gasoline combustion models. The engine model was imported into xMOD using the FMI export features of the Dymola[1] tool. The operation graph of this use-case contains over 100 operations.

6.3.2 Test Campaign

We based our tests on three different versions of RCOSIM. We refer to our proposed method as MUO-RCOSIM (Multi-Rate Oriented RCOSIM). We compared the obtained results with two approaches: The first one is RCOSIM which is mono-rate and thus we had to use the same communication step size for all the FMUs. We used a communication step size of 20μs. The second one consists in using RCOSIM with the multi-rate graph

[1] http://www.3ds.com/products-services/catia/products/dymola

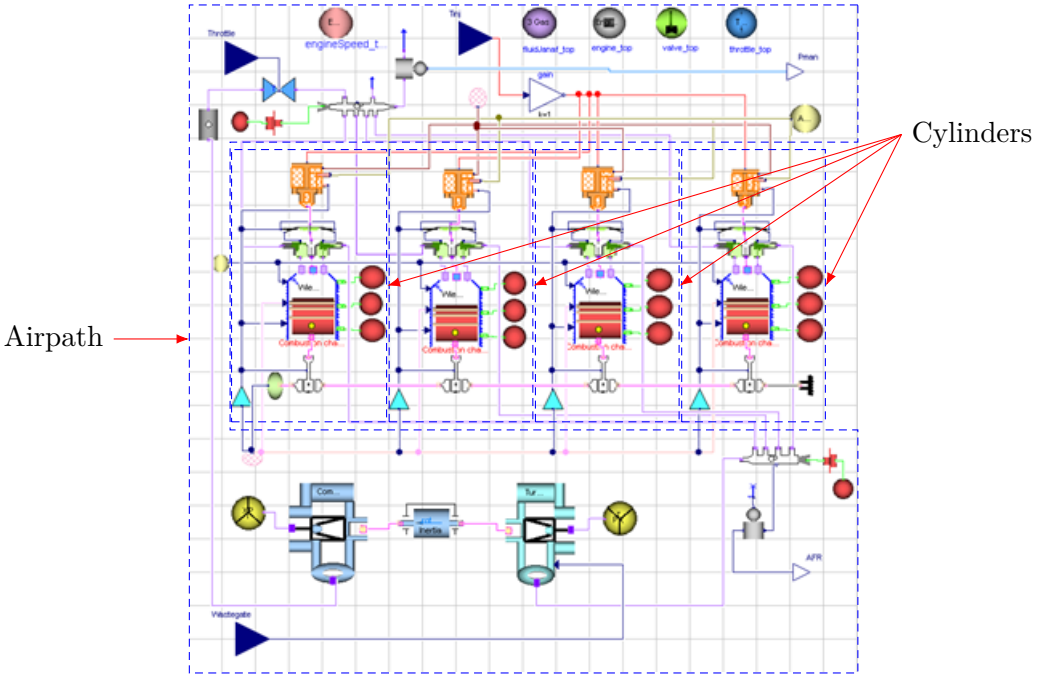


Figure 6.14: Spark Ignition (SI) RENAULT F4RT engine model.

transformation algorithm. We refer to it as MU-RCOSIM (for Multi-Rate RCOSIM). For MUO-RCOSIM and MU-RCOSIM we used the recommended configuration of the communication step sizes for this use case. For each cylinder, we used a communication step size of $20\mu s$. The communication step size used for the airpath is $100\mu s$. The airpath has slower dynamics than the cylinders and this configuration of the communication step sizes corresponds to the specification given by engine engineers. For each FMU, we used a Runge-Kutta 4 solver with a fixed integration step size that is equal to the communication step size. The graph of this use case is transformed by Algorithm 1 into a graph containing over 280 operations that are scheduled by the multi-core scheduling heuristic.

6.3.3 Numerical Accuracy

The validation of the numerical results of the co-simulation using the proposed method is achieved through the comparison of the co-simulation outputs with reference outputs. Since it is not possible to solve the equations of the FMU analytically, the reference outputs are obtained by using RCOSIM which has been shown in [5] to give a very good accuracy of the numerical results. Figure 6.15 shows the obtained results for the torque (an output of the airpath). We note that the results match with the reference, and the generated error is very small remaining within an acceptable bound ($< 1\%$). Similar accuracy results were obtained for the different outputs of the co-simulation.

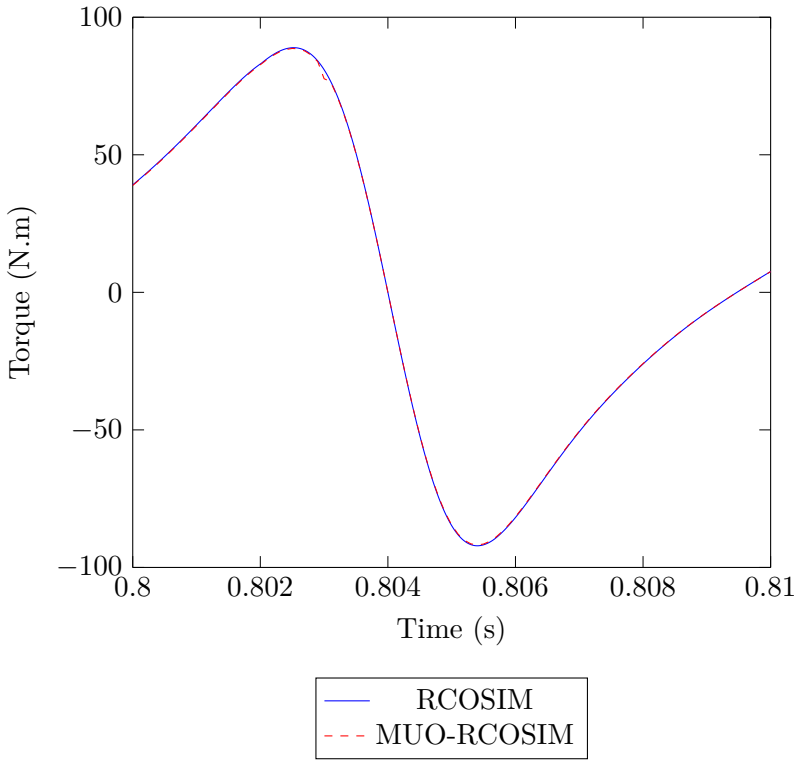


Figure 6.15: Numerical results.

6.3.4 Speedup

The speedup obtained using MUO-RCOSIM is compared with the speedups obtained using RCOSIM and MU-RCOSIM. The speedup was evaluated by running the co-simulation in xMOD. Execution times measurements were performed by getting the system time stamp at the beginning and at the end of the co-simulation. For a given run of the co-simulation, the speedup is computed by dividing the single-core co-simulation execution time of RCOSIM by the co-simulation execution time of this run on a fixed number of cores. Figure 6.16 sums up the results. The same speedup obtained using both MUO-RCOSIM and MU-RCOSIM is higher than the one obtained using RCOSIM even when only 1 core is used. This speedup is obtained thanks to using the multi-rate configuration. More specifically, increasing the communication step size of the airpath from $20\mu s$ to $100\mu s$ results in fewer calls to the solver leading to an acceleration in the execution of the co-simulation. By using multiple cores, speedups are obtained using both MUO-RCOSIM and MU-RCOSIM. Additionally, MUO-RCOSIM outperforms MU-RCOSIM with an improvement in the speedup of approximately 30% when 2 cores are used, and approximately 10% when 4 cores are used. This improvement is obtained thanks to the acyclic orientation heuristic which defines an efficient order of execution for the operations of each FMU that are mutually exclusive. This defined order tends to allow the multi-core scheduling heuristic to better adapt the potential parallelism of the operation graph to the effective parallelism of the multi-core processor (number of cores) resulting in an

improvement in the performance. MU-RCOSIM, on the other hand, uses the solution of RCOSIM which consists in simply allocating mutual exclusive operations to the same core introducing restrictions on the possible solutions of the multi-core scheduling heuristic. When using 8 cores, no further improvement is possible since the potential parallelism is fully exploited. Worse still, the overhead of the synchronization between the cores becomes counter-productive, which explains why the speedup with 8 cores is less than the speedup with 4 cores for all the approaches. The best performance is obtained using 5 cores with slight improvement compared to using 4 cores.

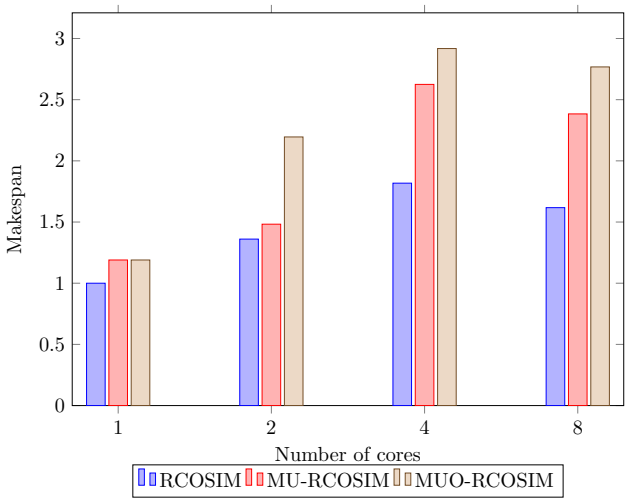


Figure 6.16: Speedup results.

6.3.5 Comparison of Offline and Online Scheduling

In this thesis, we adopted an offline scheduling heuristic assuming it is more efficient than online scheduling since it introduces lower overhead. This choice was based on the fact that the grain size of the operation graph is small which makes it unsuitable for online scheduling which involves more decision overhead in runtime than offline scheduling. In fact, the decision overhead in runtime may become much more costly then the execution of the operations. Moreover, the different operations perform different tasks and have different execution times in contrast to applications that exhibit data parallelism which could be efficiently handled by online scheduling. In addition, the execution times of the operations and the dependence between them are known before the execution which allows the application of offline scheduling. In order to confirm this assumption, we have compared our approach with a runtime scheduling approach, i.e. online scheduling. For this end, we have used Intel TBB library for the parallelization of the co-simulation. We performed several speedup tests and compared the results obtained using the two approaches.

Intel TBB Flow Graph

We have chosen Intel TBB to implement an online scheduling because it offers a programming interface introduced in Intel TBB 4.0, which allows easy parallelization of programs represented as graphs. It can be combined with loop parallelism supported by Intel TBB to further improve the parallelism exploitation. In Intel TBB, we distinguish between dependence graphs and data flow graphs. In dependence graphs, a dependence represents a precedence constraint between two nodes. During execution, this dependence acts as a signal to inform a node that a predecessor has finished its execution. In data flow graphs, a dependence is accompanied by data transfer from a predecessor to a node. In our implementation we used dependence graphs as explianed hereafter. Intel TBB offers a wide range of classes that can be used to implement dependence graphs. In particular the graph class and other related classes are used for this purpose. In general, a dependence graph involves three main components: a graph object, nodes, and edges. A graph object provides methods for the execution of tasks created from the nodes of the graph and to wait for the execution of the dependence graph to finish. Provided node classes allow the creation of different types of nodes. These nodes can be classified into four categories as shown in Figure 6.17.

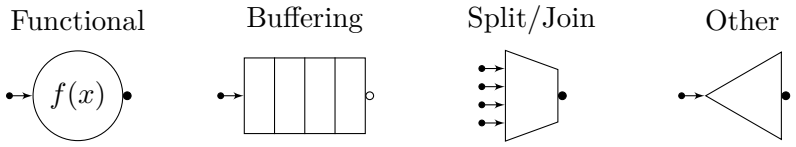


Figure 6.17: Types of nodes supported by the Intel TBB Flow Graph interface.

Functional nodes can be used to execute user code provided as a body object. Buffering nodes allow accumulating messages as they flow through the graph and forwarding them to successors. Different buffering protocols are supported by Intel TBB such as FIFO, arbitrary order, or priority order. Split/Joint nodes can be used for aggregation and deaggregation of messages. There exist several other specific purpose node types, e.g. broadcast node. Inputs and outputs of nodes are called ports. The user creates a graph, its nodes and then specifies dependence between them. In Intel TBB, edges are used to create the dependence. These edges can be created using dedicated functions provided by Intel TBB. Such functions can be used for the creation and removal of edges in the graph, as well as managing ports of nodes. The classes and functions of Intel TBB are highly parametrized to allow many possibilities of implementation.

The execution of the dependence graph follows the partial order specified by the created edges. When a node receives a signal of completion, a task is spawned to execute the body of this node.

We present here the fundamental concepts necessary to describe how we used Intel TBB. The official documentation[2] of Intel TBB should be consulted for more detailed explanation.

[2]software.intel.com/en-us/tbb-reference-manual

Scheduling in Intel TBB

Intel TBB is based on programming with tasks instead of threads. Tasks are atomic units of execution that are allocated to threads to be executed. The objective is to make programming simpler by thinking at a higher level, i.e. specifying the potential parallelism of the program without having to handle the adaptation to the effective parallelism. The threads that run the tasks are called worker threads. The allocation is automatically done in runtime using an online scheduling algorithm known as *work stealing*. Each thread keeps a pool of tasks that are ready to be executed in a deque which is a double-ended queue. Elements can be pushed onto or popped from a deque from both ends. Threads are responsible for task creation, known as task spawning. When a task is spawned by a thread, it is pushed onto the deque of this thread from the top. The thread always pops the task on the top of its deque and executes it. As such, a thread uses its local deque as a stack. If the local deque is empty, the thread tries to pick a task from another randomly chosen thread, called the victim. It pops a task from the bottom of the deque of the victim thread, therefore using the deque of the victim as a queue.

In the case of an application implemented as a dependence graph, tasks are spawned on behalf of the nodes of the graph. When a node receives messages from all its predecessors, a task is spawned on behalf of this node. When run, this task executes the body of the node. When a task finishes its execution it sends a message that is transferred to its predecessors.

Implementation

We used Intel TBB to implement parallel FMI co-simulation in xMOD. The first part which consists in creating the operation graph through the analysis of inter and intra-FMU dependence is the same as in RCOSIM. If the co-simulation is multi-rate, the multi-rate graph transformation is performed as well. Once the operation graph is constructed, an Intel TBB dependence graph which represents this operation graph is automatically created. The graph is of a dependence graph type because we do not manage explicitly data transfer between the different operations since the functions of the FMUs are provided in the form of binaries. Data transfer is implicitly managed by the partial order defined in the operation graph. In other terms, an operation that produces data is necessarily executed before the operation that consumes it. Data writing and reading is done through shared memory and is hidden from the developer. It follows from this that data flow graphs provided by Intel TBB are not suitable for representing such co-simulations because they require explicit management of data transfer between the nodes.

The creation of the dependence graph is done as follows: First, a graph is created and then nodes and edges are added to this graph. A node is created for each operation and added to an array that stores all the created nodes. The first node that is created is a source node which has no predecessor. This node becomes a predecessor of all the nodes that have no predecessor in the operation graph. The body of this node contains the initialization of the co-simulation. Then, for each operation in the operation graph, a function node is created. A function node can have multiple ports to be connected

with multiple predecessors and successors. The body of each function node contains the FMU function calls of the corresponding operation. Finally, the edges that connect the nodes are added to the graph. All the created edges are of type continue message. Such edges are used to signal that the execution is finished.

The execution of the co-simulation consists in executing this dependence graph repeatedly, similarly to our offline scheduling approach, i.e. the whole graph is executed at each iteration before a new execution can begin. Initially, one thread is responsible for the creation of the dependence graph and launching the source node. Only the source node, which performs the initialization of the co-simulation, is executed explicitly using a function provided by the Intel TBB library. When this function is called, a task is spawned to execute the body of the source node. Afterward, the runtime library handles the flow of messages in the graph. When the execution of the source node body is finished, it sends a continue message to all its predecessors. Tasks are spawned for the nodes that receive the messages to be executed which in turn send continue messages when their execution is finished and so forth. After all the nodes are executed, the execution is restarted in the same way. The scheduling is managed by the runtime library which creates a pool of working threads and uses the work stealing algorithm described above.

Comparison

We implemented a parallelization approach of FMI co-simulations using Intel TBB for the purpose of comparing it with our proposed offline scheduling approach. We have measured the speedups obtained on different numbers of cores using both approaches. First of all, let's summarize the differences between the two approaches. Figure 6.18 illustrates the main steps of both approaches. As stated above, the two first two phases which consist in the construction of the operation graph and the graph transformation in the case of a multi-rate co-simulation are performed in the same way in both approaches. If online scheduling is used, the next step is execution. On the other hand, if offline scheduling is used, two more phases are performed before the execution. The acyclic orientation heuristic is applied on the operation graph to handle mutual exclusion constraints. After this, the offline scheduling heuristic is used to compute a schedule of the operations. During execution, in both the offline and online scheduling approaches, a thread is executed on each core. In the case of offline scheduling, each thread reads the schedule table of its corresponding core and executes the operations in the order of this schedule, which does not change during execution. In the case of online scheduling, since no schedule is computed before execution, the runtime library distributes the operations across the the threads during execution in such a way to balance the load. Each thread pushes the operations onto its deque form the top. It executes these operations by popping the operation on the top from its deque, or if its deque is empty, it steals work form another thread by popping an operation from the bottom of this victim thread. Mutual exclusion constraints are handled in online scheduling using lightweight mutex locks provided by the runtime library. These locks have lower cost than mutex locks provided by the OS.

We ran the co-simulation of the use case on an 8-core Intel core i7 processor running at

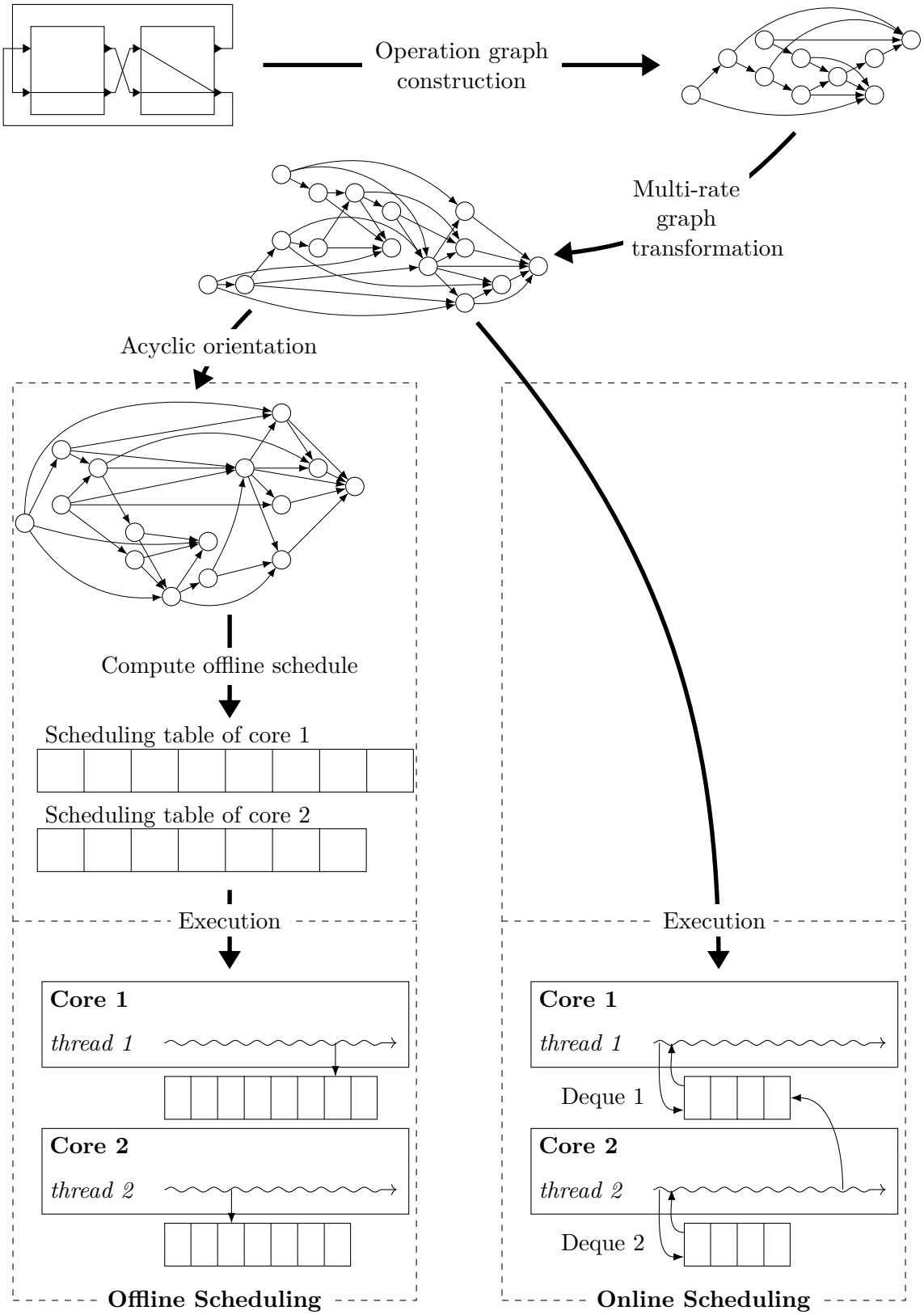


Figure 6.18: Comparison of the different phases of the offline and online scheduling approaches.

2.7 GH with 16GB RAM. The results are shown in table 6.1. The speedup obtained using offline scheduling is better than the one obtained using online scheduling which confirms our assumption. The decision overhead of online scheduling is very costly compared to the execution times of operations which decreases the performance.

Table 6.1: Comparison of speedup obtained using offline and online scheduling

Scheduling Approach	Offline	Online
Speedup	2.76	1.64

7 Conclusion

Contents

This chapter concludes the thesis. First we give a summary of the contributions presented in the previous chapters. Second, we give some perspectives for future work.

7.1 Summary

The complexity of CPS is steadily increasing due to several factors. A lot of efforts is being made in industry as well as in academia in order to implement technologies and methods that respond to the requirements and challenges in the design of complex CPS. Co-simulation is increasingly being adopted as a system-level simulation approach in the context of CPS design thanks to its advantages over monolithic simulation. Strengths of co-simulation include easy upgrade, reuse, and exchange of models, improved computational performance compared to monolithic simulation, and allowing better intervention of experts at the subsystem level in multi-domain design projects. This being said, co-simulation faces a number of challenges that have to be addressed. This thesis constitutes a contribution towards solving some of these challenges.

In this thesis, we are interested in the rising requirements on the computational performance of FMI co-simulation. We build on the work that was previously developed at IFP Energies nouvelles and aim at improving the existing methods. The focus of the thesis is on multi-core execution of co-simulation. In particular, there are two main goals for the research in this thesis. First, we aim at overcoming the limitations of the RCOSIM approach in order to allow the acceleration of different kinds of co-simulation. Second,

we aim at extending the use of RCOSIM to co-simulation under real-timje constraints in the context of HiL. Below we summarize the contributions of this thesis.

In Chapter 4 we propose extensions to the operation graph model used in RCOSIM to represent the co-simulation. The first extension targets multi-rate co-simulation. We propose some rules for transforming a multi-rate operation graph into a mono-rate one in order to prepare its multi-core scheduling. Based on these rules, we propose an algorithm that performs this transformation.

The second extension consists in transforming the operation graph in order to handle mutual exclusion constraints between operations. First, the operation graph is transformed into a mixed graph by adding (non oriented) edges between mutually exclusive operations. Then, an acyclic orientation is computed for the mixed graph by assigning a direction to each edge. We propose two algorithm to perform the acyclic orientation: an ILP-based exact algorithm and a heuristic.

The last extension aims at completing the operation graph of a co-simulation under real-time constraints to enable the application of a real-time multi-core scheduling algorithm. We focus on HiL co-simulation composed of a real and a simulated components. We propose methods for propagating real-time constraints imposed by inputs and outputs of the real component on gate operations of the simulated component. Such constraints are propagated to all the operation of the graph assigning a release and a deadline date to each operation. We propose two algorithms to perform the propagation of release and deadline constraints respectively.

In Chapter 5 we propose multi-core scheduling algorithms. First, we focus on co-simulation acceleration. For this we propose two multi-core scheduling algorithms. The first algorithm is an ILP-based exact algorithm and the second one is a list scheduling heuristic. The schedule is computed using either of these algorithms over the hyperstep. During execution, this schedule is executed repeatedly.

Second, we propose algorithms for multi-core scheduling of co-simulation under real-time constraints. Similarly, we propose an ILP-based exact algorithm and a list scheduling heuristic. Also, we propose a simple technique to study the schedulability of the operation graph and determining a periodic pattern of the schedule.

Finally, in Chapter 6, we evaluate our proposed approach. First, we propose a random generator of operation graphs. We use this graph to generate a large number of synthetic operation graphs of different sizes and structures and with different attributes. For co-simulation under real-time constraints, the generator generates also a random number of inputs and outputs of the real component and connect it to the operation graph.

We evaluate the performances of our proposed ILP-based exact algorithms and heuristics for the acyclic orientation, scheduling for co-simulation acceleration, and real-time scheduling respectively. The obtained results show the efficiency of our heuristics. While the proposed ILP algorithms give optimal results for small operation graphs they suffer from intractable execution times. Our proposed heuristics, on the other hand, give acceptable results within acceptable execution times.

Last, we validate our approach for co-simulation acceleration against an industrial

use case. The obtained results show the improvements made thanks to using multi-rate co-simulation and also using the acyclic orientation to handle mutual exclusion constraints. In addition, we compare our approach with a runtime (online) scheduling approach. Our approach outperforms it which consolidates our choice of adopting an offline scheduling approach.

7.2 Perspectives

We present below some possible research directions for future work.

Grain Size Determination

Our proposed approach relies on partitioning the co-simulation into operations that are scheduled in parallel. The size of the operations, referred to as *grain size* in parallel computing terminology, may have an important impact on the achievable performance. In fact, there is a tradeoff between the grain size and the overhead of scheduling.

Addressing the problem of grain size determination in operation graphs should strengthen our proposed approach. There exist some approaches in the literature such as [94] that can be tested on operation graphs at first. Then, the use of graph clustering algorithms can be explored in order to address this problem.

Quantized State Systems

In this thesis we were interested only in co-simulation using solvers based on time discretization. A interesting alternative to such solvers are QSS solvers. The latter are known for performing well especially for solving ODEs with discontinuities [95]. Therefore, it is worth investigating the impact of using QSS solvers on the acceleration and the execution under real-time constraints of co-simulation

Real-time Schedulability Analysis

In our proposed approach for scheduling co-simulation under real-time constraints, we used a schedulability analysis based on simulation. In addition, we do not determine the schedulability interval before computing the schedule. Moreover, the existing results on the schedulability interval for real-time systems with arbitrary deadlines are very pessimistic. Therefore, it is very important to derive an analytic schedulability condition. Alternatively, finding a more optimistic schedulability interval than the existing results for performing schedulability analysis based on simulation would represent a major enhancement.

Latency Constraints

In our work, we enabled the execution of co-simulation under real-time constraints by first propagating these constraints to all the operations of the graph. We made this choice in order to be able to apply real-time scheduling algorithms which require that each operation be characterized by classical real-time parameters. We briefly explored an alternative to

our method which consists in defining latency constraints on gate operations only. By using such technique, scheduling algorithms suitable for latency constraints can be applied. For example, in future work, the algorithms proposed in [96] can be tested for this purpose.

Preemptive Scheduling

In this thesis, we adopted a non preemptive scheduling policy for both the acceleration and the execution under real-time constraints of co-simulation. In future work, it is worth investigating preemptive scheduling policies. This may be related to the grain size determination question. In fact, preemption may be beneficial only if it does not introduce an overhead more costly than the execution of the operations.

References

[1] C. Faure. “Real-time simulation of physical models toward hardware-in-the-loop validation”. PhD thesis. France: Université Paris-Est, Oct. 2011.

[2] A. Ben Khaled. “Distributed real-time simulation of numerical models: application to powertrain”. PhD thesis. France: Université de Grenoble, May 2014.

[3] E. A. Lee and S. A. Seshia. *Introduction to embedded systems: a cyber-physical systems approach*. 2nd ed. MIT Press, 2017.

[4] FMI development group. *Functional mock-up interface for model exchange and co-Simulation*. July 2014.

[5] A. Ben Khaled et al. “Fast multi-core co-simulation of cyber-physical systems: application to internal combustion engines”. In: *Simulation Modelling Practice and Theory* 47 (2014), pp. 79–91.

[6] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O’Reilly Media, Inc.", 2007.

[7] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems*. Academic press, 2000.

[8] D. Harel. “Statecharts: a visual formalism for complex systems”. In: *Science of Computer Programming* 8.3 (1987), pp. 231–274.

[9] C. A. Petri. “Kommunikation mit automaten”. In: (1962).

[10] E. Kofman and S. Junco. “Quantized-state systems: a DEVS Approach for continuous system simulation”. In: *Transactions of The Society for Modeling and Simulation International* 18.3 (2001), pp. 123–132.

[11] A. Ben Khaled et al. “Context-based polynomial extrapolation and slackened synchronization for fast multi-core simulation using FMI”. In: *10th International Modelica Conference*. Linköping University Electronic Press. 2014, pp. 225–234.

[12] A. Ben Khaled-El Feki et al. “CHOPtrey: contextual online polynomial extrapolation for enhanced multi-core co-simulation of complex systems”. In: *Simulation* 93.3 (2017), pp. 185–200.

[13] IEEE Standards Association et al. *1516–2010-IEEE Standard for modeling and simulation (M&S) High Level Architecture (HLA)*. 2012.

[14] K. Lamberg and P. Wältermann. “Using HIL simulation to test mechatronic components in automotive engineering”. In: *dSPACE GmbH, Munich* 15 (2000), p. 16.

[15] Modelica Association. *The Modelica Language Specification Version 3.4*. 2017.

[16] M. O. Faruque et al. “Real-time simulation technologies for power systems design, testing, and analysis”. In: *IEEE Power and Energy Technology Systems Journal* 2.2 (2015), pp. 63–73.

[17] G. E. Moore. “Cramming more components onto integrated circuits”. In: *Electronics* 38.8 (Apr. 1965), pp. 114–117.

[18] G. M. Amdahl. “Validity of the single processor approach to achieving large scale computing capabilities”. In: *AFIPS ’67 (Spring) Proceedings of the April 18-20, 1967, spring joint computer conference*. Apr. 1967.

[19] M. J Flynn. “Some computer organizations and their effectiveness”. In: *IEEE transactions on computers* 100.9 (1972), pp. 948–960.

[20] OpenMP. “OpenMP application programming interface”. In: (Nov. 2015). Available at www.openmp.org, version 4.5, November, 2015.

[21] MPI. *MPI: A Message-Passing Interface Standard*. Available at www.mpi-forum.org, version 3.1. June 2015. URL: `http://www.mpi-forum.org/`.

[22] J. Diaz, C. Munoz-Caro, and A. Nino. “A survey of parallel programming models and tools in the multi and many-core era”. In: *IEEE Transactions on Parallel and Distributed Systems* 23.8 (2012), pp. 1369–1386.

[23] R. I. Davis and A. Burns. “A survey of hard real-time scheduling for multiprocessor systems”. In: *ACM Computing Surveys* 43.4 (Oct. 2011).

[24] J. Y.-T. Leung, ed. *Handbook of scheduling: algorithms, models, and performance analysis*. Boca Raton, FL, USA: Chapman&Hall/CRC, 2004.

[25] M. R. Garey and David S. Johnson. *Computers and intractability*. Vol. 29. W. H. Freeman & Co. New York, 2002.

[26] T. C. Hu. “Parallel sequencing and assembly line Problems”. In: *Operations Research* 9 (1961), pp. 841–848.

[27] E. G. Coffman Jr. and R. L. Graham. “Optimal scheduling for two-processor systems”. In: *Acta Informatica* 1 (1972), pp. 200–213.

[28] C. H. Papadimitriou and M. Yannakakis. “Scheduling interval-ordered tasks”. In: *SIAM Journal on Computing* 8.3 (1979), pp. 405–409.

[29] P. C. Fishburn. *Interval orders and interval graphs: a study of partially ordered sets*. New York, NY.: John Wiley and Sons, Inc., 1985.

[30] T. L. Adam, K. M. Chandy, and J. R. Dickson. “A comparison of list scheduling for parallel processing systems”. In: *Communications of the ACM* 17.12 (Dec. 1974), pp. 685–690.

[31] H. Kasahara and S. Narita. “Practical multiprocessor scheduling algorithms for efficient parallel processing”. In: *IEEE Transactions on Computers* C-33.11 (Nov. 1984), pp. 1023–1029.

[32] B. Shirazi, M. Wang, and G. Pathak. “Analysis and evaluation of heuristic methods for static task scheduling.” In: *Journal of Parallel and Distributed Computing* 10.3 (Nov. 1990), pp. 222–232.

[33] B. Kruatrachue and T. G. Lewis. *Duplication scheduling heuristics (DSH): a new precedence task scheduler for parallel processor systems.* Tech. rep. Oregon State University, 1987.

[34] M.-Y Wu and D. D. Gajski. “Hypertool: A programming aid for message-passing systems.” In: *IEEE Transactions on Parallel and Distributed Systems* 1.3 (July 1990), pp. 330–343.

[35] J.-J. Hwang et al. “Scheduling precedence graphs in systems with interprocessor communication times”. In: *SIAM Journal on Computing* 18.2 (Apr. 1989), pp. 244–257.

[36] G. C. Sih and E. A. Lee. “A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures”. In: *IEEE Transactions on Parallel and Distributed Systems* 4.2 (Feb. 1993), pp. 75–87.

[37] T. Yang and A. Gerasoulis. “DSC: scheduling parallel tasks on an unbounded number of processors”. In: *IEEE Transactions on Parallel and Distributed Systems* 5.9 (Sept. 1994), pp. 951–967.

[38] E. S. H. Hou, N. Ansari, and H. Ren. "A genetic algorithm for multiprocessor scheduling". In: *IEEE Transactions on Parallel and Distributed Systems* 5.2 (Aug. 1994), pp. 113–120.

[39] A. S. Wu et al. "An incremental genetic algorithm approach to multiprocessor scheduling". In: *IEEE Transactions on Parallel and Distributed Systems* 15.9 (Sept. 2004), pp. 824–834.

[40] F. A. Omara and M. M. Arafa. "Genetic algorithms for task scheduling problem". In: *Journal of Parallel and Distributed Computing* 70.1 (Jan. 2010), pp. 13–22.

[41] J. A. Stankovic. "Misconceptions about real-time computing: a serious problem for next-generation systems". In: *Computer* 21.10 (1988), pp. 10–19.

[42] C. L. Liu and J. W. Layland. "Scheduling algorithms for multiprogramming in a hard-real-time environment". In: *Journal of the ACM (JACM)* 20.1 (1973), pp. 46–61.

[43] J. Y.-T. Leung and J. Whitehead. "On the complexity of fixed-priority scheduling of periodic, real-time tasks". In: *Performance evaluation* 2.4 (1982), pp. 237–250.

[44] A. Mok. "Fundamental design problems of distributed systems for the hard real-time environment". PhD thesis. Cambridge, MA, USA: Massachusetts Institute of Technology, May 1983.

[45] B. Andersson and J. Jonsson. "Fixed-priority preemptive multiprocessor scheduling: to partition or not to partition". In: *Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on*. IEEE. 2000, pp. 337–346.

[46] Y. Sorel. "Real-time embedded image processing applications using the algorithm architecture adequation methodology". In: *Proceedings of IEEE International Conference on Image Processing, ICIP'96*. Lausanne, Switzerland, Sept. 1996.

[47] A. Iserles and S. P. Nørsett. "On the theory of parallel runge-kutta methods". In: *IMA Journal of Numerical Analysis* 10.4 (1990), pp. 463–488.

[48] G. D. Byrne and A. C. Hindmarsh. "PVODE, an ODE solver for parallel computers". In: *International Journal of High Performance Computing Applications* 13.4 (1999), pp. 254–365.

[49] H. Elmqvist et al. "Automatic GPU code generation of modelica functions". In: *11th International modelica conference*. Versailles, France, 2015.

[50] M. Gebremedhin et al. "A data-parallel algorithmic modelica extension for efficient execution on multi-core platforms". In: *Proceddings of the 9th International Modelica Conference*. Munich, Germany, 2012.

[51] H. Elmqvist, S.E. Mattsson, and H. Olsson. "Parallel model execution on many cores". In: *Proceedings of the 10th International Modelica Conference*. Lund, Sweden, 2014.

[52] J. Clauberg and H. Ulbrich. "An adaptive internal parallelization method for multibody simulations". In: *12th Pan-American Congress of Applied Mechanics*. 2012.

[53] J.-L. Lions, Y. Maday, and G. Turinici. "Résolution d'EDP par un schéma en temps «pararéel»". In: *Comptes Rendus de l'Académie des Sciences-Series I-Mathematics* 332.7 (2001), pp. 661–668.

[54] C. Farhat and M. Chandesris. "Time-decomposed parallel time-integrators: theory and feasibility studies for fluid, structure, and fluid–structure applications". In: *International Journal for Numerical Methods in Engineering* 58.9 (2003), pp. 1397–1434.

[55] M. Emmett and M. Minion. "Toward an efficient parallel in time method for partial differential equations". In: *Communications in Applied Mathematics and Computational Science* 7.1 (2012), pp. 105–132.

[56] E. Lelarasmee, A. E. Ruehli, and A. L. Sangiovanni-Vincentelli. "The waveform relaxation method for time-domain analysis of large scale integrated circuits". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 1.3 (July 1982), pp. 131–145.

[57] S. Y. R. Hui and C. Christopoulos. “Numerical simulation of power circuits using transmission-line modelling”. In: *IEE Proceedings A (Physical Science, Measurement and Instrumentation, Management and Education)* 137.6 (Nov. 1990), 379384.

[58] M. Sjölund et al. “Towards efficient distributed simulation in Modelica using transmission line modeling”. In: *3rd International Workshop on Equation- Based Object-Oriented Languages and Tools EOOLT*. Oslo, Norway: Linkping Univ. Electronic Press, 2010, pp. 71–80.

[59] R. Braun and P. Krus. “Multi-threaded real-time simulations of fluid power systems using transmission line elements”. In: *8th International Fluid Power Conference*. Dresden, Germany, 2012.

[60] P. Aronsson. “Automatic parallelization of equation-based simulation programs”. PhD thesis. Sweden: Linkoping University, Mar. 2006.

[61] A. Ben Khaled et al. “Multicore simulation of powertrains using weakly synchronized model partitioning”. In: *IFAC Workshop on Engine and Powertrain Control Simulation and Modeling ECOSM*. Rueil-Malmaison, France, 2012, pp. 448–455.

[62] V. Galtier et al. “FMI-based distributed multi-simulation with DACCOSIM”. In: *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*. Society for Computer Simulation International. 2015, pp. 39–46.

[63] V. Galtier et al. “Experimenting with matryoshka co-Simulation: building parallel and hierarchical FMUs”. In: *12th International Modelica Conference*. 2017.

[64] H. Lundvall and P. Fritzson. “Automatic parallelization of object oriented models executed with inline solvers”. In: *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*. Springer. 2007, pp. 365–372.

[65] W. H. Kohler. “A preliminary evaluation of the critical path method for scheduling tasks on multiprocessor systems”. In: *IEEE Transactions on Computers* 100.12 (1975), pp. 1235–1238.

[66] Omar Kermia and Yves Sorel. “A rapid heuristic for scheduling non-preemptive dependent periodic tasks onto multiprocessor”. In: *Proceedings of ISCA 20th international conference on Parallel and Distributed Computing Systems, PDCS’07*. 2007.

[67] K. Ramamritham. “Allocation and scheduling of precedence-related periodic tasks”. In: *IEEE Transactions on Parallel and Distributed Systems* 6.4 (Apr. 1995), pp. 412–420.

[68] E. Balas. “Machine sequencing via disjunctive graphs: an implicit enumeration algorithm”. In: *Operations research* 17.6 (1969), pp. 941–957.

[69] V. C. Barbosa and J. L. Szwarcfiter. “Generating all the acyclic orientations of an undirected graph”. In: *Information Processing Letters* 72.1-2 (1999), pp. 71–74.

[70] T. R. Jensen and B. Toft. *Graph coloring problems*. Vol. 39. John Wiley & Sons, 2011.

[71] T. Gallai. “On directed paths and circuits”. In: *Theory of Graphs* (1968), pp. 115–118.

[72] B. Roy. “Nombre chromatique et plus longs chemins d’un graphe”. In: *Revue française d’informatique et de recherche opérationnelle* 1.5 (1967), pp. 129–132.

[73] M. Hasse and H. Reichel. “Zur algebraischen Begründung der Graphentheorie. III”. In: *Mathematische Nachrichten* 31.5-6 (1966), pp. 335–345.

[74] L. M. Vitaver. “Determination of minimal coloring of vertices of a graph by means of boolean powers of the incidence matrix”. In: *Doklady Akademii Nauk SSSR* 147 (1962), pp. 758–759.

[75] R. M. Karp. “Reducibility among combinatorial problems”. In: *Complexity of Computer Computations*. Springer, 1972, pp. 85–103.

[76] B. Ries. “Coloring some classes of mixed graphs”. In: *Discrete Applied Mathematics* 155.1 (2007), pp. 1–6.

[77] G. V. Andreev, Y.i N. Sotskov, and F. Werner. “Branch and bound method for mixed graph coloring and scheduling”. In: *Proceedings of the 16th International Conference on CAD/CAM, Robotics and Factories of the Future, CARS and FOF*. 2000, pp. 1–8.

[78] Y. N. Sotskov, V. S. Tanaev, and F. Werner. “Scheduling problems and mixed graph colorings”. In: *Optimization* 51.3 (2002), pp. 597–624.

[79] F. S. Al-Anzi et al. “Using mixed graph coloring to minimize total completion time in job shop scheduling”. In: *Applied Mathematics and Computation* 182.2 (2006), pp. 1137–1148.

[80] J. Bélanger, P. Venne, and J. N. Paquin. “The what, where and why of real-time simulation”. In: *Planet RT* 1.0 (2010), p. 1.

[81] T. Grandpierre, C. Lavarenne, and Y. Sorel. “Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors”. In: *Proceedings of the 7th International Workshop on Hardware/Software Co-Design, CODES’99*. Rome, Italy, May 1999.

[82] A. Benveniste and G. Berry. “The synchronous approach to reactive and real-time systems”. In: *Proceedings of the IEEE* 79.9 (1991), pp. 1270–1282.

[83] A. Benveniste et al. “The synchronous languages 12 years later”. In: *Proceedings of the IEEE* 91.1 (2003), pp. 64–83.

[84] H. Chetto, M. Silly, and T. Bouchentouf. “Dynamic scheduling of real-time tasks under precedence constraints”. In: *Real-Time Systems* 2.3 (1990), pp. 181–194.

[85] L. Cucu and J. Goossens. “Feasibility intervals for fixed-priority real-time scheduling on uniform multiprocessors”. In: *Emerging Technologies and Factory Automation, ETFA’06*. IEEE. 2006, pp. 397–404.

[86] B. P. Dave, G. Lakshminarayana, and N. K. Jha. “COSYN: hardware-software co-synthesis of heterogeneous distributed embedded systems”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 7.1 (1999), pp. 92–104.

[87] L. Cucu and J. Goossens. “Feasibility intervals for multiprocessor fixed-priority scheduling of arbitrary deadline periodic systems”. In: *Design, Automation & Test in Europe Conference & Exhibition, DATE’07*. IEEE. 2007, pp. 1–6.

[88] E. Grolleau, J. Goossens, and L. Cucu-Grosjean. “On the periodic behavior of real-time schedulers on identical multiprocessor platforms”. In: *arXiv preprint arXiv:1305.3849* (2013).

[89] H. Kalla. “Génération automatique de distributions/ordonnancements temps réel fiables et tolérant les fautes”. PhD thesis. Grenoble: Institut National Polytechnique de Grenoble, 2004.

[90] M. Berkelaar, K. Eikland, P. Notebaert, et al. “lpsolve: open source (mixed-integer) linear programming system”. In: *Eindhoven University of Technology* (2004).

[91] Inc. Gurobi Optimization. *Gurobi Optimizer Reference Manual*. 2016. URL: `http://www.gurobi.com`.

[92] IBM ILOG CPLEX. “V12. 7: users manual for CPLEX”. In: *International Business Machines Corporation* (2017).

[93] Z. Benjelloun-Touimi et al. “From physical modeling to real-time simulation: feedback on the use of modelica in the engine control development toolchain”. In: *8th International Modelica Conference*. Dresden, Germany: Linköping Univ. Electronic Press, Mar. 2011.

[94] Boontee Kruatrachue and Ted Lewis. “Grain size determination for parallel processing”. In: *IEEE software* 5.1 (1988), pp. 23–32.

[95] Gustavo Migoni, Ernesto Kofman, and François Cellier. “Quantization-based new integration methods for stiff ordinary differential equations”. In: *Simulation* 88.4 (2012), pp. 387–407.

[96] O. Kermia. “Ordonnancement temps réel multiprocesseur de tâches non préemptives avec contraintes de précédence, de périodicité stricte et de latence”. PhD thesis. Orsay: Université de Paris Sud, 2009.