THÈSE DE DOCTORAT DE
l'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Salah Eddine SAIDI

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse :

Automatic Parallelization and Scheduling Approaches for Co-simulation of Numerical Models on Multi-core Processors

soutenue le 30 février 2042

devant le jury composé de :

M. Prénom Nom	Directeur de thèse
M. Prénom Nom	Rapporteur
M. Prénom Nom	Rapporteur
M. Prénom Nom	Examinateur
M. Prénom Nom	Examinateur

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 real parts of the system are already available, it is possible to connect these real components 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 inputs from the real components and providing numerical 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 as well as real-time execution without using multiprocessor architectures. FMI (Functional Mocked-up Interface), an increasingly common standard for model exchange and co-simulation, oers new opportunities for multi-core execution of numerical models, by enabling intra-model parallelization. One objective of this thesis is to define algorithms for extracting potential parallelism in a set of interconnected multi-rate models. Another one consists in proposing algorithms for the allocation and scheduling of models, taking into account their real-time, data dependencies and allocation constraints. Such algorithms aim to accelerate the execution of the co-simulation or ensure its real-time execution. Prior to this thesis, an approach has been developed at IFPEN which allows the parallelization of FMI models on multi-core processors. In the first part of the thesis, improvements have been proposed to overcome the limitations of this approach. 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. The proposed algorithms allow also the optimization of the phase of analyzing the models structures and their data dependencies in order to compute their schedule on the multi-core processor. Finally, the proposed improvements allow more accurate measurements of the execution times of the models using a profiling technique and allow also handling some specific constraints like mutual exclusion constraints. This thesis is part of a joint action IFPEN - Inria in which Inria brings its real-time systems experience to the numerical simulation challenges of IFPEN.

Contents

List of Figures

List of Abbreviations

. . . . . .

1
Introduction

Contents

1.1 Context

1.2 Objectives and Contributions

1.3 Thesis Outline

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 is becoming an indisputable step in the design process of complex systems. Simulation is an effective cost reducer since it allows to correct design errors before building the system. Simulation is performed by providing models which describe the behavior of the system and then running these models in order to

produce the behavior of the simulated system on a computer.

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. A system is defined as a set of interacting parts which form a complex whole. A system is characterized by the cohesion between its forming parts which operate as a whole 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 receives inputs from the environment and produces outputs to the environment, and is characterized by an internal state. The state of the system is affected by the inputs that it takes, and, in turn, affects the produced outputs. Figure [ref] 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.

A kind of systems that falls within the scope of this thesis is known as Cyber-Physical Systems (CPS). CPS consist of a combination of tightly 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 controllers run control software whose behavior is affected by the 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 as discrete event systems using formalisms such as DEVS (Discrete Event System Specification), Statecharts, or Petri nets. 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 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). Equation 2.1 is an ODE where x is the vector of the dependent 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.

$$\dot{x} = f(x,t) \tag{2.1}$$

In contrast, a differential equation is called Partial Differential Equation, abbreviated PDE if it involves unknown functions and their partial derivatives. 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, and f is a given function.

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

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.

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

In this thesis, we are particularly interested in modeling dynamical systems using ODEs. Typically, such systems are hybrid dynamic 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 dynamic 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 describe 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.

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

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. Figure shows the trajectory of the ball. This hybrid dynamic system can be modeled using a hybrid automaton as shown in figure

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, the time varying values of the quantities of interest. This data is 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 *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. 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. 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 with slow evolution 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, ...) which should be taken into consideration when choosing a solver for a specific problem. Overall, it is always desirable to choose a solver that is reliable, robust and fast.

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 are modeled in the discrete-time domain. For such systems, it becomes necessary to do the modeling on the subsystem level, usually without a complete view about the the rest of the system. 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 tool instead of using a single modeling software. 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 greater or equal to the integration step size of the models and defines the rate of communication (data exchange) of this model. Figure shows the evolution of time and data exchange between two models A and B. In this example, the equations of model A are solved using a fixed step size h_A whereas the equation of model B are solved using a variable step

step h_B. The communication step size 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 communications steps only. Between communications steps, each model considers that the values of data produced by the other models are held constant. Another alternative is to estimate these values by employing extrapolation techniques.

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

Model-in-the-Loop

Model-in-the-Loop (MiL) co-simulation is performed at an early stage of the controller design. In MiL, the model of the controller is included with the model of the controlled physical process in a co-simulation is used 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

Once the controller model is validated, it 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 real 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. It can ensure reliable tests of the controller and thus shortens the development time and reduces the costs. If any problems are detected, one can go back to SiL or MiL stage to make necessary corrections.

Figure gives a v-cycle view of the different stages of co-simulation performed as part of the process of controller design.

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 MODELISAR project and is currently developed and maintained by the Modelica Association[1]. 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] https://www.modelica.org/association/

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 FMI for Model Exchange 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.
- FMI for Co-Simulation: FMI for Co-Simulation 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 shows the state machine of the supported calling sequences from a master algorithm to a slave.

2.1.5 Co-simulation with Real-time Constraints

In the design process of complex systems, it is often necessary to test the behavior of the system as it would be produced by the real system. Therefore, the simulation is executed with real-time constraints such that the progress of the simulation time matches the real-time. For example, if temperature takes three minutes to reach 25°, the simulation has to take three minutes as well. Similarly, a co-simulation with real-time constraints has to be executed with the same rate of the real-time.

A typical application of real-time co-simulation is HiL. The key advantage of real-time co-simulation is that it allows testing the controller under real-like conditions even if the physical process is not available. Many HiL solutions are developed in such a way to work using 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).

Roughly speaking, the difference between real-time simulation and non real-time (offline) simulation is related to the notion of results validity. For offline simulation, one seeks to obtain results as soon as possible. The validity of the the results depends only on their numerical accuracy. For real-time simulation, the validity of results depends on their numerical accuracy but also on their availability time, i.e. they have to be available within specific time bounds. If such bounds are missed, the results are considered invalid even if their values are correct from a numerical standpoint.

Figure [ref] illustrates the time evolution of a real-time simulation. 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 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 simulation constraints are violated which makes the simulation invalid. This is known as an *overrun*.

2.1.6 Languages and Tools for Modeling and Simulation

Building models of systems can be done manually and than 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 some tools and languages for modeling and simulation.

xMOD

xMOD[2] is the modeling and co-simulation software developed by IFP Energies nouvelles. It supports FMI and provides an environment for the integration of heterogeneous models built by different parties using different languages and tools. xMOD can execute models embedding different solvers with different integration and communication step sizes. xMOD does not replace original modeling and simulation tools. Instead, it promotes and facilitates their coupling and existence.

MATLAB Simulink

Simulink[3], developed by The Mathworks is a graphical modeling environment. Simulink is used in the process of Model-Based Design of embedded systems. Models are built graphically in Simulink using block diagrams. In addition, Simulink is integrated in MATLAB which allows the incorporation of MATLAB algorithms in Simulink models. Finally, Simulink enables automatic code generation form models.

Modelica

Modelica 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 equations. Modelica adopts an acausal approach, i.e. the signal flow 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.

Dymola

Dymola[4], developed by Dassault Systèmes AB, is a modeling and simulation environment based on the Modelica modeling language. Dymola supports the FMI standard and allows

[2] http://www.xmodsoftware.com/
[3] www.mathworks.com/products/simulink/
[4] http://www.3ds.com/products-services/catia/products/dymola

interfacing with other tools such as Simulink

Amesim

Amesim[5] is a modeling and simulation software developed by Siemens PLM Software. Amesim can be used for the modeling and simulation of mechatronic systems. Amesim is based on the Modelica modeling language. It is oriented towards the modeling of complex physical systems instead of controller design. Amesim provides libraries containing collections of components that can be loaded and connected by the user to build models. For simulation, Amesim automatically selects a solver that is adapted to the problem.

Hopsan

Hopsan[6] 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.

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. Its basic idea is to divide a computing 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 was 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 [1]. This prediction proved correct for many years. However, frequency scaling is facing technological limits and the last decade witnessed a wide shift to multicore processors among semiconductor manufacturers. Moore's law can be considered to be still accurate since more transistors are still integrated in chips, not for frequency scaling but to add more processing elements. The rise of multicore 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 [2]. 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 will limit the speedup. The speedup is therefore not linear with the number of processors and probably adding more processors will not make the program run faster. 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 **??** shows the theoretical speedup of a program in function

[5] www.plm.automation.siemens.com/en_us/products/lms/imagine-lab/amesim/
[6] https://www.iei.liu.se/flumes/system-simulation/hopsan?l=en

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.

2.2.1 Parallel Architectures

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 [3] 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 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 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 instructions 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.

Multiple Instruction-Streams–Multiple Data Streams (MIMD) Different instructions are executed on different data 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.

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 needs 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 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 architectures. That being said, taking advantage of such architectures requires efficient ways for executing applications on parallel architectures. A difficult yet integral step in this direction is the process of detecting the parallelism that is inherent in software. It is important that 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.

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, dependencies exist between the tasks, e.g. the result of one task is needed as input by another task.

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.

2.2.3 Parallel Programming

In order to efficiently map software parallelism on parallel architectures, many parallel programming libraries, APIs, and standards have been developed. Basically they differ according to the targeted type of memory.

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. 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. There are many libraries and APIs for shared memory programming. The following are some examples.

Open Multi-Processing (OpenMP) [4] is an API that has been designed to develop applications that are meant to be executed on shared memory parallel computers such as multicore 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) is a C++ parallel programming library. Using Intel TBB, the developer specifies the parallelism in the form of tasks, not threads. These tasks are automatically mapped by the library onto threads. 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) [5]. 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.

2.2.4 Parallel Scheduling

Parallelization consists in the transformation of a sequential program into a multithreaded one 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 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, and project management. The objective of scheduling for parallel computing is to take full advantage of a parallel architecture for the execution of a parallel application . 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 mult-icore 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 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 $T = \{t_1, t_2, \ldots, t_n\}$ and a set of m processors is denoted $P = \{p_1, p_2, \ldots, p_m\}$. Scheduling consists allocating tasks form T to processors from P with respect to predefined criteria. 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 [6]:

- *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 a the execution of a task can be preempted and resumed later. If a schedule is not preemptive, it is called non preemptive. Furthermore, a scheduling algorithm is either dynamic or static. Dynamic scheduling algorithms are used when some information about the tasks are not known before the execution. The scheduling algorithm makes scheduling decisions online as the information becomes available. Static 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 [7]. Makepsan is the time needed by a computer to process a set of tasks. The general objective of parallel computing is to accelerate the execution of application which corresponds to the makespan objective.

Task Dependence Graph

A set of tasks 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 is represented by a vertex $v_i \in V : 0 \leq i < n$ where n is the number of tasks. 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 the execution. Generally, dependence between two tasks is due to data movement, i.e. one task is executed and produces a data element, and the second task requires this data element 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 (see Figure **??**). In the remainder of the thesis 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 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 be executed in parallel, e.g. t_2 and t_3 in Figure **??**. 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 because which describes how hardware parallelism limits the exhibition of the application parallelism.

List Scheduling

Heuristics are usually used to solve parallel scheduling problems because these problems are NP-copmlete [gary] and using exact algorithms results in exponentially increasing execution times. In particular, list scheduling heuristics have been successfully used in the context of static 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 in the literature to minimize the makespan of a graph with no transitive arcs is Hu's algorithm [8]. It assigns a level to each vertex in G as follows: All vertices that have no immediate successor are at level 1. Then for each of the other vertices, 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) and which has the highest level among all ready tasks to the first available processor.

Coffman-Graham algorithm [9] 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 [10] studied the problem of scheduling interval-ordered task graphs. In such a graph, two vertices are precedence-related if and only if they can be mapped to non-overlapping intervals on the real line [11]. 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 assigned in this order.

In [12] level-based algorithms for scheduling dependence graphs are presented. The proposed Highest Level First with Estimated Times algorithm labels the vertices of the DAG with levels where the level corresponds to the sum of computation costs on the longest path from the vertex to a sink vertex. It then allocates the tasks in a highest-level first fashion, therefore, the level of a task represents its priority. Highest Levels First with No Estimated Times algorithm works similarly but with the assumption that all tasks have unit computation costs. [13] proposes a similar algorithm with the improvement of breaking ties by selecting the vertex with the largest number of successors.

In [14] two algorithms are proposed: First, the Heavy Node First algorithm which is based on a local analysis of the vertices at each level. It allocates the heaviest vertice first. The second algorithm, WL (Weighted Length), considers a global view of the dependence graph by taking into account the relationships among the nodes at different levels.

[15] proposed the ISH algorithm. The main idea of ISH is to fill the "scheduling holes" which

are the idle time slots as the schedule is being constructed.

The MCP (Modified Critical Path) algorithm proposed by [16] uses the measure of how late can a task be delayed without increasing the makespan of the schedule. MCP assigns priorities to tasks in an ascending order of their latest start dates.

The Earliest Start Time algorithm [17] computes at each step, for each task, the earliest start date and selects the task that has the smallest one to allocate it.

The DLS (Dynamic Level Scheduling) algorithm [18] assigns dynamic levels to tasks. The dynamic level of a task is equal to the difference between the b-level (longest path from the corresponding vertice to a sink vertice) 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 [19] Yang and Gerasoulis present the DSC algorithm which uses an attribute called the dominant sequence which is the critical path of the partially ordered graph.

A current trend in multiprocessor scheduling is to use meta-heuristics such as Genetic Algorithms (GA) [20–22].

2.2.5 Parallel Real-time Scheduling

Real-time scheduling concerns the scheduling of tasks in real-time systems. Here, real-time does not mean fast but it refers to systems that must be able to respond to external events within specified deadlines. 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 is a kind of real-time systems 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 temporal parameters. A real-time task t_i is characterized by the following parameters (Figure **??**):

- 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 instance. The time at which an instance becomes ready to be executed is called the activation or the release time. r_i^k is the release time of the k^{th} instance of the task t_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} instance finishes its execution;
- Response time R_i^k: The duration between the release time and the finishing time of the k^{th} instance: $R_i^k = f_i^k - r_i^k$;
- Absolute deadline d_i^k: The time at which the k^{th} instance must finish its execution;

- Relative deadline D_i: The duration, starting from the release time, that 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 $l_i^k = d_i^k - (t + C_i(t))$.

According to how consecutive instances of a task are activated, three kinds of tasks are distinguished:

- Periodic tasks: The instances of a given task are activated periodically with a known period. A periodic task is characterized by its period T_i;
- Sporadic tasks: The instances of a task are activated by an event and 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.

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 few deadlines to be missed but if a task's deadline is missed, its result is no more useful. 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 erroneous results and probably causes 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 may be changed 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. Schedulability tests are performed to check whether the tasks can be scheduled using a given scheduling algorithm in such a way to satisfy all the requirements. The 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): Tasks are assigned priorities inversely proportional to their periods. A set of tasks is schedulable by RM if: $U \leq n(2^{\frac{1}{n}} - 1)$, $D_i = T_i$.
  - Deadline Monotonic (DM): Tasks are assigned priorities inversely proportional to their relative deadlines. Tasks are schedulable using DM if: $\Delta \leq n(2^{\frac{1}{n}} - 1)$, $D_i \leq T_i$.

- Dynamic priorities
  - Earliest Deadline First (EDF): Priorities of tasks are inversely proportional to their absolute deadlines. The priority of a task is fixed for one instance but may change from one instance to another. EDF can schedule a set of tasks if and only if: $U \leq 1, D_i = T_i$.
  - Least Laxity First (LLF): Priorities of tasks are inversely proportional to their laxities. The priority may change for the same instance and from one instance to another. The schedulability test is the same as for EDF.

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

- 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. That 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:

- First Fit (FF): A task is tested on all cores and for each task the test starts from the first core. 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.
- Next Fit (NF): Similar to FF but the search of the core that can schedule the task does not start always from the first one. After allocating a task to a core, the core search for the next task starts from the next core.
- Best Fit (BF): Test the task on all cores and allocate it to the one that gives the minimum of U.
- Worst Fit (WF): Allocate the task to the core that gives the maximum of U.

2.3 Parallelization 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 simulations have been porposed 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 simulations.

2.3.1 Methods

In order to achieve simulation acceleration through parallel execution, different approaches are possible and were already explored. The parallelization approaches can be classified into three categories based on the level at which the parallelization is intorduced.

Parallelization across the method

In this category, we find approaches that seek to parallelize the integration method. For instance, a multi-satge solver requires several computations within one integration step and it is possible to perform multiple computations within one step in parallel. Such approach is studied in [23] by proposing a theoretical framework for parallelization of Runge-Kutta methods. Another approach consists in parallelizing operations on vectors for ODEs resolution like in the PVODE solver [24] implemented using MPI.

In [25], 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 [26], ParModelica, an algorithmic extension to 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 [27]. This approach organizes the equations into a set of layers containing each a number of sections that can be executed in parallel and computes a static schedule for their execution. In [28] an approach for the parallelization of multi-body simulations on shared memory multiprocessors is processors. 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 Parreal algorithm [29], the Parallel Implicit Time-Integrator (PITA) [30], and the Parallel Full Approximation Scheme in Space and Time (PFASST) [31]. 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 of the simulation are solved in parallel. A well known approach that parallelizes across the system is Waveform Relaxation (WR) [32], 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.

Transmission Line Modeling (TLM) [33] 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 [34, 35] are based on the TLM method.

Co-simulation is naturally adapted to parallelization across the system. In fact, as shown in [36], splitting a model into several FMUs, by isolating discontinuities, may reduce the simulation time, even in the case of a sequential execution. In [37] 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.

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. Thus, this feature 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 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 prallelization approach of Dymola is detailed in [27]. 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 [28].

3
Problem Position

4

Dependence Graph Model for FMU Co-simulation

Contents

This chapter describes a dependence graph model that is used in this thesis for representing a co-simulation program. 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, 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. The model has also to represent other features of the program such as data dependence between different pieces of the code. Task dependence graphs are commonly used for this purpose. A task dependence graph is a DAG denoted $G(V, A)$ where:

- V is the set of vertices of the graph. The size of the graph n is equal to the number

of its vertices. Each vertex $v_i : 0 \leq i < n$ represents a task which is an atomic unit of computation.

- A is the set of arcs of the graph. A directed arc is denoted as a pair (v_i, v_j) and describes a precedence constraint between v_i and v_j, i.e. v_i has to finish its execution before v_j can start its execution. v_i is called the *tail* vertex and v_j is called the *head* vertex.

The task dependence graph defines the partial order to be respected when executing the set of tasks. This partial order describes the potential parallelism of the program.

The co-simulation of FMUs lends itself to the task 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 as a task 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 [37].

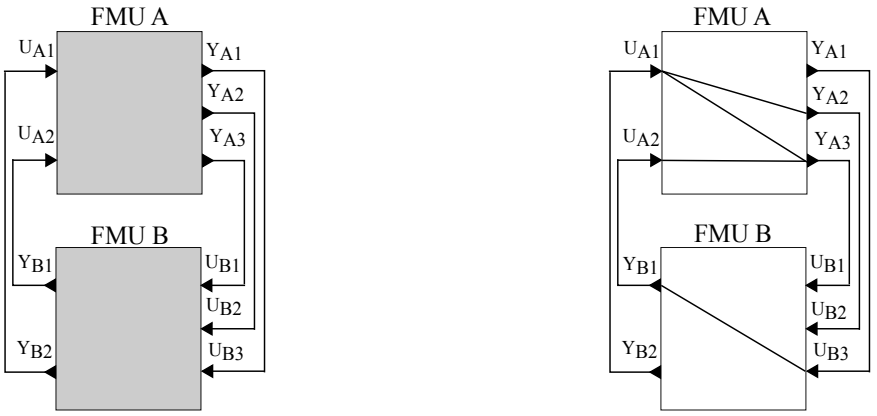
4.1.1 Construction of the Dependence Graph of an FMU Co-Simulation

The entry point for the construction of a task dependence graph of an FMU co-simulation is a user-specified set of interconnected FMUs as depicted in Figure 4.1a. The execution of each FMU is seen as computing a set of input operations (one operation for each of the inputs of the FMU), a set of output operations (one operation for each of the outputs of the FMU), and one state operation for updating the state variables of the FMU. An operation is defined by a number of FMU C function calls. An input (resp. output) operation is executed by calling *fmiSet* (resp. *fmiGet*) function and a state operation is executed by calling *SetTime, GetDerivatives, SetContinuousStates, etc.*, functions in the case of FMI for Model Exchange or *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 input operations and output operations are given. The computation of the state at a given simulation step k is considered necessary for the computation of each one of the output operations at the same simulation step k. It is considered that the computation of the state at a simulation step $k + 1$ requires the computation of each of the input operations at the simulation step 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 operations at a given simulation step depend on the state computation at the same step. Also, it is specified which input operations at a simulation step k need to be computed before the computation of the state at the step $k + 1$.

FMU information on input/output dependence allows building a graph with an increased granularity. The co-simulation is described by a task dependence graph $G(V, A)$ called the



(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

operation dependence 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 dependence graph. The operation dependence 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 variables 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 simulation step k of an output must be performed with the same value of the state (computed at simulation step k) as for all the outputs belonging to the same FMU. Running the co-simulation consists in repeatedly executing the graph obtained that way. A new execution of the graph cannot be started unless the previous one was totally finished. The operation dependence graph corresponding to the FMUs of Figure 4.1 is shown in Figure 4.2.

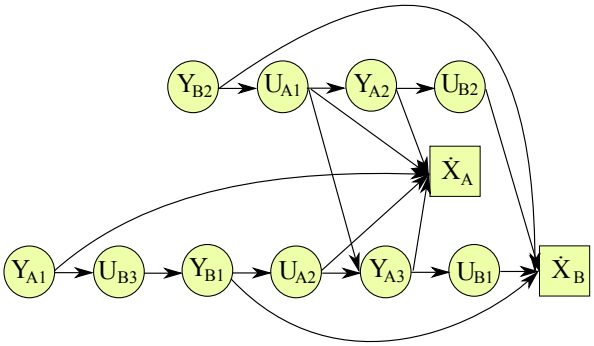


Figure 4.2: Operation dependence graph obtained from the FMUs of Figure 4.1

4.1.2 Dependence Graph Attributes

In the remainder of the thesis, we shall use the term operation graph instead of operation dependence graph. The operation graph is used as input to the scheduling algorithm. In addition to the structure of 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 []. We define a set of attributes and notations to characterize the operation graph.

The notation $f_m(o_i)$ is used to refer to the FMU to which the operation o_i belongs, and $T(o_i)$ to denote the type of the operation o_i, i.e. $T(o_i) \in \{update_{input}, update_{output}, update_{state}\}$. o_j is a predecessor of 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 $pred(o_i)$. o_j is an ancestor of 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 $ance(o_i)$. o_j is a successor of 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 $succ(o_i)$. o_j is a descendant of 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 $C(o_i)$. For each operation, the average execution time of multiple co-simulation runs is used. When real-time execution 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 of its FMU. The earliest start time from start $S(o_i)$ and the earliest end time form start $E(o_i)$ are defined by equations 4.1 and 4.2 respectively. $S(o_i)$ defines the earliest time at which the operation o_i can start its execution. $S(o_i)$ is constrained by the precedence relations. The earliest time the operation o_i can finish its execution is defined by $E(o_i)$.

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

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

The latest end time from end denoted by $\overline{E}(o_i)$ and the latest start time from end denoted by $\overline{S}(o_i)$ 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 R 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.

$$R = \max_{o_i \in V_I}(E(o_i)) \tag{4.5}$$

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

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

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

The operation graph model presented in the previous sections allows elementary modeling of FMU co-simulation programs. 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 steps. 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 the use of specific values of the communication step. It is therefore highly unrecommended, and in some cases even impossible, to change the communication step of the FMU. In other cases, even if it is possible and acceptable to change the communication step of a given FMU, better performance and/or accuracy could be obtained when using specific communication steps. As a consequence, our operation graph model has to be extended in order to accommodate multi-rate data exchange between operations.

Consider an operation graph that is constructed as described in the previous section from a multi-rate co-simulation, i.e. some FMUs have different communication steps. Such graph is referred to as a multi-rate operation graph. One way for making such operation graph suitable for multi-core scheduling is to transform it into a mono-rate graph. This section presents an algorithm that transforms the initial multi-rate operation operation graph $G(V, A)$ into a mono-rate operation operation graph $G_M(V_M, A_M)$. The aim of this transformation is to ensure that each operation is executed according to the communication step of its respective FMU and also to maintain a correct data exchange between the different FMUs, whether they have different or identical communication steps. Similar algorithms have been used in the real-time scheduling literature [38, 39].

We define the *Hyper-Step (HS)* as the least common multiple (lcm) of the communication steps of all the operations: $HS = lcm(H(o_1), H(o_2), \ldots, H(o_n))$ where $n = |V_I|$ is the number of operations in the initial graph. The Hyper-Step is the smallest interval of time for describing a repeatable pattern of all the operations. The transformation algorithm consists first of all in repeating each operation o_i, r_i times where r_i is called the repetition factor of o_i and $r_i = \frac{HS}{H(o_i)}$. 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 simulation step. We use a superscript to denote the number of each occurrence, for instance o_i^s denotes the s^{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 of the FMU they belong to. Therefore, we define the repetition factor of an FMU to be equal to the repetition factor of its operations. Then, arcs are added between operations following the rules presented hereafter. Consider two operations $o_i, o_j \in V_I$ connected by an arc $(o_i, o_j) \in A_I$, then adding an arc (o_i^s, o_j^u) to A_M, depends on the simulation steps (time) at which o_i^s and o_j^u are executed. In other words, if k_s and k_u are the simulation steps associated with o_i^s and o_j^u respectively, then the inequality $k_s \leq k_u$ is a necessary condition to add the arc (o_i^s, o_j^u) to A_M. In addition, o_j^u is connected with the latest occurrence of o_i that satisfies this condition, i.e. with the occurrence o_i^s such that $s = \max(0, 1, \ldots, r_i - 1) : k_s \leq k_u$. In the case where $H(o_i) = H(o_j)$ (and therefore $r_i = r_j$), occurrences o_i^s and o_j^u which correspond to the same number, i.e. $s = u$, 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 $(o_i, o_j) \in A_I$ 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 $(o_i, o_j) \in A_I$, one occurrence of o_i is executed while several occurrences of o_j are executed. In this case, arcs

are added between each occurrence $o_i^s : s \in \{0, 1, \ldots, r_i - 1\}$, and the occurrence o_j^u such that:

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

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 $(o_i, o_j) \in A_I$, arcs are added between each occurrence o_i^s, and the occurrence $o_j^u : u \in \{0, 1, \ldots, r_j - 1\}$ such that:

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

Arcs are added also between the occurrences of the same operation, i.e. $(o_i^s, o_i^{s'})$ where $s \in \{0, 1, \ldots, r_i - 2\}$ and $s' = s + 1$. Finally, for each FMU, arcs are added between the s^{th} occurrence of the state operation, where $s \in \{0, 1, \ldots, r_i - 2\}$, and the $(s+1)^{th}$ occurrences of the input and output operations. 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 4.3 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.

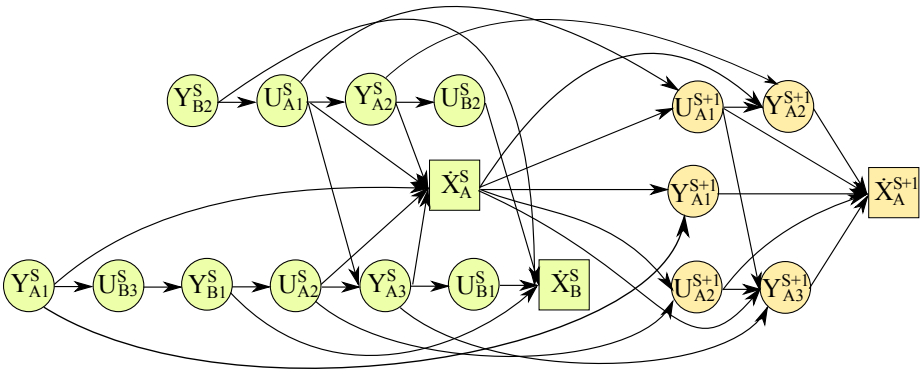


Figure 4.3: Graph obtained by applying the multi-rate transformation algorithm on the graph of Figure 4.2

Without 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 clarity. Each vertex of the graph $G(V, A)$ represents an operation that is referred to using the notation o_i.

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 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 lightweight handling of these constraints.

Algorithm 1: Multi-rate graph transformation algorithm

Input: Initial operation operation graph $G_I(V_I, A_I)$;
Output: Transformed operation operation graph $G_M(V_M, A_M)$;
Set $G_I(V_I, A_I)$ the initial multi-rate operation operation graph and $G_M(V_M, A_M)$ the new mono-rate graph;
$V_M \leftarrow \emptyset$; $A_M \leftarrow \emptyset$;
foreach *operation* $o_i \in V_I$ **do**
  Compute the repetition factor of o_i: $r_i \leftarrow \frac{HS}{H(o_i)}$;
  Repeat the operation o_i: $V_M \leftarrow V_M \cup \{o_i^s\}, s \in \{0, \ldots, r_i - 1\}$;
end
foreach *arc* $(o_i, o_j) \in A_I$ **do**
  if $H(o_i) > H(o_j)$ *((o_i, o_j) is a slow to fast dependence)* **then**
    Compute $u = \left\lceil s \times \frac{H(o_i)}{H(o_j)} \right\rceil$ and add the arc (o_i^s, o_j^u) to the new graph $G_M(V_M, A_M)$:
    $A_M \leftarrow A_M \cup \{(o_i^s, o_j^u)\}, s \in \{0, \ldots, r_i - 1\}$;
  end
  else if $H(o_i) < H(o_j)$ *((o_i, o_j) is a fast to slow dependence)* **then**
    Compute $s = \left\lfloor u \times \frac{H(o_i)}{H(o_j)} \right\rfloor$ and add the arc (o_i^s, o_j^u) to the new graph $G_M(V_M, A_M)$:
    $A_M \leftarrow A_M \cup \{(o_i^s, o_j^u)\}, u \in \{0, \ldots, r_j - 1\}$;
  end
  else
    Add the arc (o_i^s, o_j^s) to the new graph $G_M(V_M, A_M)$:
    $A_M \leftarrow A_M \cup \{(o_i^s, o_j^s)\}, s \in \{0, \ldots, r_i - 1\}$;
  end
end
foreach *operation* $o_i \in V$ **do**
  Add an arc between successive occurrences of o_i:
  $A_M \leftarrow A_M \cup \{(o_i^s, o_i^{s+1})\}, s \in \{0, \ldots, r_i - 2\}$;
end
foreach *operation* $o_i \in V$ *such that* $T(o_i) = state$ **do**
  Add arcs (o_i^s, o_j^{s+1}) to the new graph $G_M(V_M, A_M)$ between o_i and every operation o_j such that $M(o_j) = M(o_i)$ and $T(o_j) \in \{input, output\}$:
  $A_M \leftarrow A_M \cup \{(o_i^s, o_j^{s+1})\}, s \in \{0, \ldots, r_i - 2\}$;
end

4.3.1 Motivation for Offline Handling of Mutual Exclusion Constraints

In order to study the impact of mutual exclusion constraints, we have evaluated the performance obtained using two mutual exclusion strategies. The first one uses a dedicated lock (system object that guarantees mutual exclusion) 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. The second solution is explained in [37] and consists in allocating the operations of a same FMU to the same core (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 graph. Results are given in Figure 4.4. It shows that the expected speed-up in the case of 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 $update_{output}$ operations can be

efficiently allocated only if the unconstrained allocation is used: the speed-up difference between the constrained and unconstrained allocation cases is due to this restriction on the allocation. Five is the minimal number of cores for enabling the execution of each $update_{state}$ operation on a different core. Due to the predominant execution times of the $update_{state}$ operations, their impact on the speed-up overrides the possibility of optimizing the allocation of the other operations. This explains why the speed-up difference between the unconstrained and the constrained allocation cases becomes very small with five cores or more.

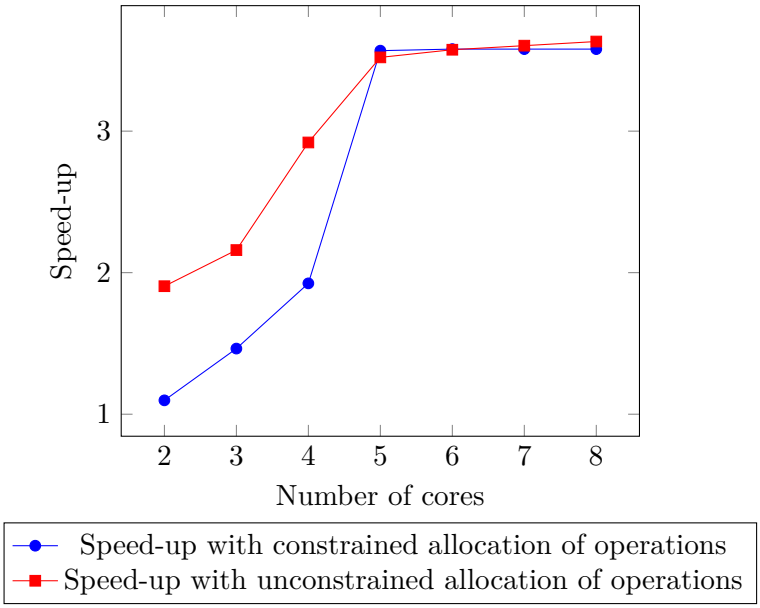


Figure 4.4: Theoretical speed-up.

We implemented and tested both mutual exclusion strategies in order to compare their runtime performance. Tests were performed using 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, we compared the speed-up by dividing the mono-core co-simulation execution time by the co-simulation execution time on a fixed number of cores. Figure 4.5 sums up the results, where unconstrained allocation corresponds to the use of lock objects. It shows the impact of mutex 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 gives similar results in terms of speed-up improvement when increasing the number of cores until 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.

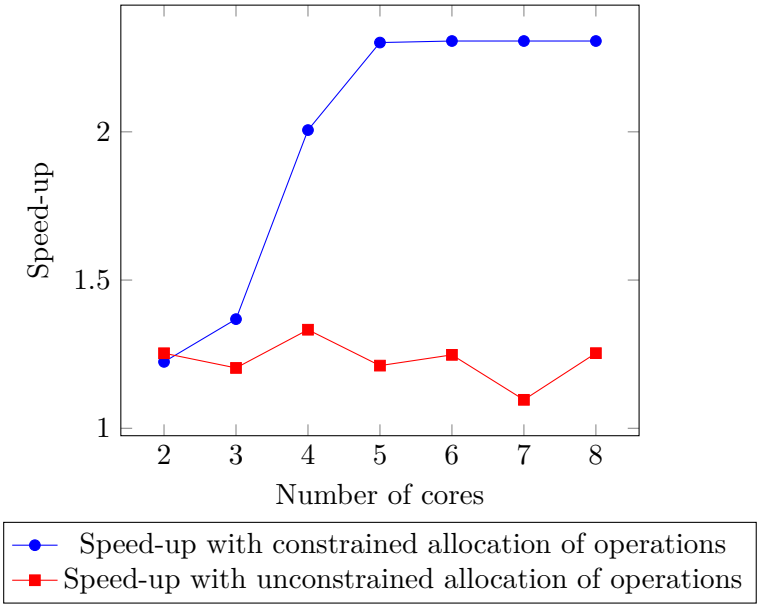


Figure 4.5: Measured speed-up.

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 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. 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. In order to compute a schedule for a mixed graph, an execution order has to be defined for each pair of operations connected by an 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 []. Since the final goal is to accelerate the execution of the co-simulation, and in other words, minimize the makespan of the execution of the operation graph, the acyclic orientation of the mixed graph has to minimize the length of the critical path of the graph.

The acyclic orientation problem is closely related to vertex coloring. 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 is called the chromatic number and is denoted $\chi(G)$. The Gallai–Hasse–Roy–Vitaver theorem [40–43] links the length of the longest path in orientations of the graph 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.

The acyclic orientation of a mixed graph can be obtained similarly. 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 $\alpha : V \to \{1, 2, \ldots, k\}$ 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 [44], 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 [45–47]. Efficient algorithms have been proposed for special types of graphs, however in the general case it has been shown that the problem is NP-Hard.

4.3.3 Problem Formulation

Let $G(V, A)$ be a 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 define an order of execution for mutually exclusive operations. The timing constraints represented by the mixed graph $G(V, A, D)$ are given by expressions 4.11 and 4.12. 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.

An edge $[o_i, o_j]$ is called a conflict edge if the intervals $(S(o_i), E(o_i)]$ and $(S(o_j), E(o_j)]$ as computed in the graph $G(V, A)$ overlap (equation 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. Still, we have to ensure mutual exclusion between them due to the non-thread-safe implementation of FMI. Following the definition given the previous section, the corresponding coloring is a function $\alpha : V \rightarrow \{1, 2, \ldots, k\}$ 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 formulation as follows:

Input: Mixed graph $G(V, A, D)$

Output: DAG $G(V, A)$

Find: Coloring $\alpha : V \rightarrow \{1, 2, \ldots, k\}$

Minimize: Number of colors k

Subject to: $\forall\ (o_i, o_j) \in A,\ \alpha(o_i) < \alpha(o_j)$
$\forall\ [o_i, o_j] \in D,\ \alpha(o_i) \neq \alpha(o_j)$

This formulation is stated as a vertex coloring problem, however, it is possible to state the problem and its solution using either vertex coloring notation or the scheduling notation. Thus, we note the following equivalence:

Coloring $\alpha : V \rightarrow \{1, 2, \ldots, k\} \equiv$ Mapping $\alpha : V \rightarrow \{[S(o_1), E(o_1)],\ [S(o_2), E(o_2)],\ \ldots,\ [S(o_n), E(o_n)]\} \equiv$ Assignment $\alpha : D \rightarrow \{(o_i, o_j), (o_j, o_i) : [o_i, o_j] \in D\}$

$\forall\ (o_i, o_j) \in A,\ \alpha(o_i) < \alpha(o_j) \equiv \forall\ (o_i, o_j) \in A,\ E(o_i) \leq S(o_j)$

$\forall\ [o_i, o_j] \in D,\ \alpha(o_i) \neq \alpha(o_j) \equiv \forall\ [o_i, o_j] \in D,\ (S(o_i), E(o_i)] \cap (S(o_j), E(o_j)] = \emptyset$

Optimal coloring $min(k) \equiv$ Optimal length of the critical path $min(R)$

4.3.4 Resolution using Integer 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 $[o_i, o_j] \in D$
P	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 $orient : const_1$ indicates that the start time of operation o_j if 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. $\alpha([o_i, o_j]) = (o_i, o_j)$ 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 P is equal to the maximum of the end times of all the operations (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).

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

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. $\forall [o_i, o_j] \in D, S(o_i) \leq S(o_j), \alpha([o_i, o_j]) = (o_i, o_j)$. This a fast greedy acyclic orientation, however it can be improved as we will show hereafter.

Let d 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:

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

It is 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 and o_j correspond to the same occurrence. Even if 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 have to finish their execution before the next occurrence 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:

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

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

Lemma 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. 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 $[o_i, o_j]$ that has been transformed into the arc (o_i, o_j) and $S(o_i) > S(o_j)$. 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. □

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 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 loss of generality, suppose that the edge $[o_i, o_j]$ is transformed into the arc (o_i, o_j), then the start time $S(o_j)$ is changed to $S(o_j) \leftarrow E(o_i)$. 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 to $\overline{E}(o_i) \leftarrow \overline{S}(o_j)$. 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) ∈ A_{k,l} do
    if S(o_h) < E(o_t) and S(o_t) < E(o_h) /* (o_t, o_h) is a conflict edge */ then
        S(o_h) ← E(o_t);
        E(o_h) ← S(o_h) + C(o_h);
    end
end
Procedure update(o_h)
    if succ(o_h) ≠ ∅ then
        foreach o*_h ∈ succ(o_h) do
            if S(o*_h) < E(o_h) then
                S(o*_h) ← E(o_h);
                E(o*_h) ← S(o*_h) + C(o*_h);
                update(o*_h);
            end
        end
    end
    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 $[o_i, o_j] \in D$. In the first step, the graph $G(V, \emptyset, D)$ obtained by removing all the arcs $(o_i, o_j) \in A$ from the mixed graph $G(V, A, D)$ is partitioned into d subgraphs where d is the number of all occurrences 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: $G(V, \emptyset, D) = \bigcup_{k=1}^{d} G_k(V_k, \emptyset, D_k)$. 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 $o_i \in V$,

Algorithm 3: Update of the start and end times from end 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 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) ∈ A_{k,l} do
    if S(o_h) < E(o_t) and S(o_t) < E(o_h) /* (o_t, o_h) is a conflict edge */ then
        if Ē(o_t) < S̄(o_h) then
            Ē(o_t) ← S̄(o_h);
            S̄(o_t) ← Ē(o_t) + C(o_t);
            update(o_t);
        end
    end
end
Procedure update(o_t)
    if pred(o_t) ≠ ∅ then
        foreach o*_t ∈ pred(o_t) do
            if Ē(o*_t) < S̄(o_t) then
                Ē(o*_t) ← S̄(o_t);
                S̄(o*_t) ← Ē(o*_t) + C(o*_t);
                update(o*_h);
            end
        end
    end
    return;
```

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 $o_{i'} \in L_k$ such that $\alpha(o_{i'}) \geq \alpha(o_i)$, are increased $\alpha(o_{i'}) = \alpha(o_{i'}) + 1$. The corresponding edges are then accordingly assigned directions. Afterward, the timing attributes of the operations are updated as described above. At this point, the increase in R, 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 the color $\phi(o_i) = \phi(o_i) + 1$. The increase in the critical path is evaluated again similarly. After repeating this process for all the edges $[o_i, o_j] \in D_k : o_j \in L_k$, the pending operation is finally assigned the color which leads to the least increase in the critical path, and edges $[o_i, o'_i] \in D_k : o'_i \in l$ 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 ranks of all its predecessors which guarantees that no cycle is generated. The heuristic finally stops when all the operations have been assigned colors.

Algorithm 4: Acyclic orientation heuristic

```
/* Create and initialize the lists L_k */
for k = 1 to d do
    L_k ← ∅;
end
Set Ω the set of all the operations not already colored: Ω ← V;
while Ω ≠ ∅ do
    Select the operation o_i ∈ Ω, o_i ∈ V_k whose start time S(o_i) = max_{o_j∈Ω}(S(o_j)). Break
     ties by selecting the operation with the least flexibility;
    Set σ ← ∞; /* Initialize the increase in the critical path) */
    foreach [o_i, o_j] ∈ D_k : o_j ∈ L_k do
        foreach c ∈ {α(o_j), α(o_j) + 1} do
            if S(o_i) < E(o_j) and S(o_j) < E(o_i) /* (o_i, o_j) is a conflict edge */ then
                α(o_i) ← c;
                foreach o_i' ∈ L_k such that α(o_i') ≥ α(o_i) do
                    α(o_i') ← α(o_i') + 1;
                    Update the timing attributes using Algorithms 2 and 3;
                end
                Compute the new critical path and set σ' the increase in the critical path;
                if σ' < σ then
                    color ← α(o_i);
                    σ ← σ';
                end
                foreach o_i' ∈ L_k such that α(o_i') > α(o_i) do
                    Reassign o_i' its previous color: α(o_i') ← α(o_i') − 1;
                end
            end
        end
        α(o_i) = color;
        foreach o_i' ∈ L_k do
            if α(o_i') > α(o_i) then
                Assign a direction to the edge [o_i, o_i'] ∈ D_k : α([o_i, o_i']) ← (o_i, o_i')
            end
            else
                Assign a direction to the edge [o_i, o_i'] ∈ D_k : α([o_i, o_i']) ← (o_i', o_i)
            end
        end
        Update the timing attributes using Algorithms 2 and 3;
        Remove o_i from Ω;
    end
end
```

Complexity

Soundness

Completeness

Termination

4.4 Dependence Graph with Real-time Constraints

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 shown in the previous chapter, it is scheduled on the multi-core platform. First, we consider scheduling the operation with the goal of accelerating the execution of 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 takes 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 here-after a linear programming model and a heuristic for scheduling FMU dependence graphs on multi-core processors with the aim of accelerating the execution of the co-simulation.

5.1.1 Problem Formulation

The acceleration of the co-simulation corresponds to the minimization of the makespan of the dependence graph. The makespan is the total execution time of the whole graph. The dependence 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. The 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 dependence graph. In addition to the execution time of the operations, the scheduling algorithm has to take into considerations, the cost of inter-core synchronization. The scheduling problem can be stated as an optimization problem as follows:

Input	Operation graph $G_F(V_F, A_F)$
Output	Offline Schedule of operations on 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 $P = max(E(o_i))_{o_i \in V}$
Subject to	Precedence constraints of the graph $G_F(V_F, A_F)$

5.1.2 Resolution using Linear Programming

In this section, we give our ILP formulation of the task scheduling problem for the acceleration of FMU co-simulation.

Constraints

We define the decision binary variables x_{ij} which indicate whether the operation o_i is allocated to core p_j or not. Expression 5.1 gives the constraint that each operation has to be allocated to one and only one core.

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

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.

$$\forall p \in P, \forall\ o_i \in V, \forall\ o_j \in V, (o_i, o_j), (o_j, o_i) \notin A_F, E(o_i) \leq S(o_j) + M \times (3 - x_{ip} - x_{jp} - b_{ij}) \tag{5.3}$$

$$\forall p \in P, \forall\ o_i \in V, \forall\ o_j \in V, (o_i, o_j), (o_j, o_i) \notin A_F, E(o_j) \leq S(o_i) + M \times (2 - x_{ip} - x_{jp} + b_{ij}) \tag{5.4}$$

The cost of synchronization is taken into account according to the synchronization model described in chapter 3. In other words, a synchronization cost is introduced in the computation of the start time of an operation o_j, if it has a predecessor that is allocated to a different core and if its start time is the earliest among the successors of this predecessor that are allocated to the same core as the operation o_j. $sync_{ijp}$ is a binary variable which indicates whether synchronization is needed between o_i and o_j if o_j is allocated to p. Therefore, $sync_{ijp} = 1$ iff $\alpha(o_j) = p$ and $\alpha(o_i) \neq p$ and $S(o_j) = max_{o_{j'} \in succ(o_i) \text{ and } \alpha(o_{j'})=p}(S(o_{j'}))$. Expressions 5.5 and 5.6 capture this constraint. V_{ip} 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 the 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_{ip} denotes the earliest start time among the start times of all successors of o_i that are allocated to processor p. It is computed using expressions 5.9 and 5.10.

$$\forall o_i \in V, \sum_{\forall p \in P, \forall o_j \in pred(o_i)} sync_{ijp} = V_{ip} \tag{5.5}$$

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

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

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

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

$$\forall o_i \in V, \forall o_j \in succ(o_i), Q_{ip} \geq S(o_j) - M \times (1 - sync_{ijp}) \tag{5.10}$$

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

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

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

$$\forall o_i \in V, P \geq E(o_i) \tag{5.12}$$

Objective function

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

$$min(P) \tag{5.13}$$

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 final graph $G_F(V_F, A_F)$ in order to minimze 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 [48] 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.

The heuristic considers the different timing attributes of each operation $o_i \in V_F$ in order to compute a schedule that minimizes the makespan of the graph. The heuristic schedules the operations of the graph $G_F(V_F, A_F)$ 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.

Complexity

5.1.4 Code Generation

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

Algorithm 5: Multi-core scheduling heuristic

```
Initialization;
Set Ω the set of all the operations;
Set P the set of all the available cores;
Set O the set of operations without predecessors;
while O ≠ ∅ do
    foreach operation o_i ∈ O do
        Set σ to ∞; (cost of o_i is set to the maximum value);
        foreach p ∈ P do
            S'(o_i) ← max(S(o_i), L_p); (new start time of o_i when executed on p);
            σ' ← S'(o_i) + C(o_i) + Ē(o_i) − R; (cost of o_i when executed on p);
            if σ' < σ then
                Set σ ← σ';
                Set BestCore(o_i) ← p;
            end
        end
    end
    Find o_i' with maximal cost σ in O;
    Schedule o_i' on its core BestCore(o_i');
    Set p' := BestCore(o_i');
    L_p' := L_p' + C(o_i'); (Advance the time of p');
    Remove o_i' from the set O;
    Add to the set O all successors of o_i' for which all predecessors are already scheduled;
end
```

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 obtained results. Finally, we give the speedup and accuracy results obtained by applying our approach on an industrial use case.

6.1 Random Generator of Dependence 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.

6.1.1 Random Dependence Graph Generation

The random generator that we have implemented is inspired by the work of [Kalla:04]. However it differs from this work because the generation is done in two stages. In fact, we have to generate, first, 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. Our generator is based on a technique of assignment of operations to levels. The level of an operation is the number 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 **??**. The generator uses of the following parameters:

- The graph size n: The number of operations;
- The number of FMUs m;

- The graph height h: The 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. The number of FMUs can be derived automatically from the size of the graph using a predefined formula. For instance, one of the formulas that we have used is $m = 5 \times log10(n/5)$. This allows to have adequate size of the graph and number of FMUs. Suppose for example that we have a size $n = 1000$ and a number of FMUs $m = 1$. Obviously, this example does not represent a realistic application.
- **Step 1:** Randomly distribute the n operations to 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 $lvl \in 2, 4, \ldots, h-3$ of the grid.
  3. Assign the input operations to the odd levels $lvl \in 1, 3, \ldots, h-4$ 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 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.

6.1.2 Random Dependence 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 steps of the FMUs.
- Execution times of the operations. The execution times are generated randomly in such a way that state operations have longer execution times than the output and input operations

6.2 Results

6.3 Industrial Use Case

The proposed parallelization approach has been applied on an industrial use case. In this section, we give a description of the use case and then present the obtained results.

Tests have been performed on a computer running the Windows 7 operating system with 16 GB RAM and an Intel core i7 processor running 8 cores at 2.7 GH. Experiments have been carried out on 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. This co-simulation is composed of 5 FMUs: one FMU for each cylinder and one FMU for the airpath. The FMUs were imported into xMOD using the FMI export features of the Dymola tool. This use-case leads to an initial graph containing over 100 operations. We refer to our proposed method as MUO-RCOSIM (for 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 for all the FMUs. We used a communication step of $20\mu s$. The second one consists in using RCOSIM with the multi-rate graph 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 steps for this use case. For each cylinder, we used a communication step of $20\mu s$. The communication step used for the airpath is $100\mu s$. In fact, the airpath has slower dynamics than the cylinders and this configuration of the communication steps corresponds to the specification given by engine engineers. For each FMU, we used a Runge-Kutta 4 solver with a fixed integration step equal to the communication step. 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.

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 [37] to give a very good accuracy of the numerical results. Figure 6.1 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.

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 mono-core co-simulation execution time of RCOSIM by the co-simulation execution time of this run on a fixed number of cores. Figure 6.2 sums up

the results. The same speedup is obtained using MUO-RCOSIM and MU-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 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.

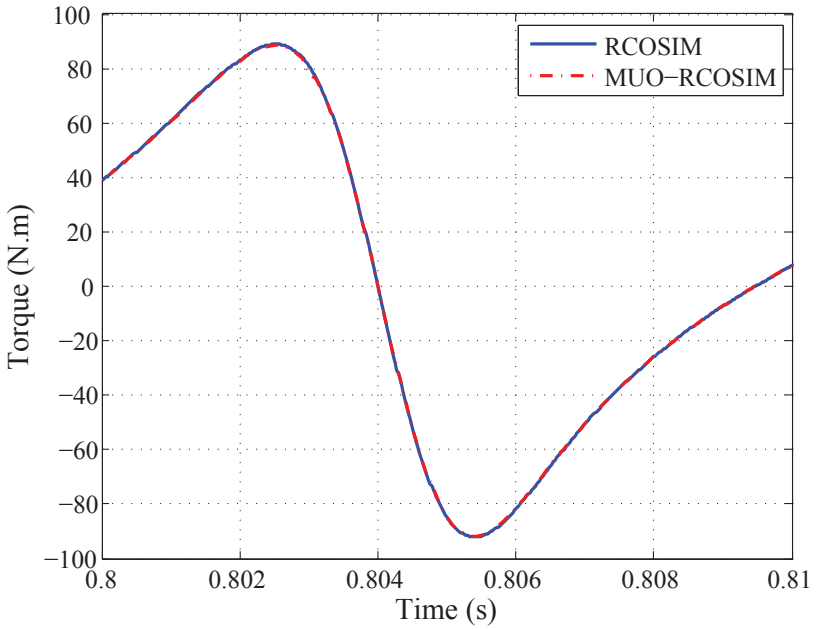


Figure 6.1: Numerical results

The obtained speedup and numerical accuracy results show the efficiency of our proposed method. In future work, we aim at comparing our solution with an exact scheduling algorithm and also an online scheduling approach. Also, we will test our proposed method on other industrial applications. In addition, we envision to extend MUO-RCOSIM to real-time multi-core scheduling in order to perform Hardware-in-the-Loop simulation whose main challenge is to map the real-time constraints on the different operations of the graph.

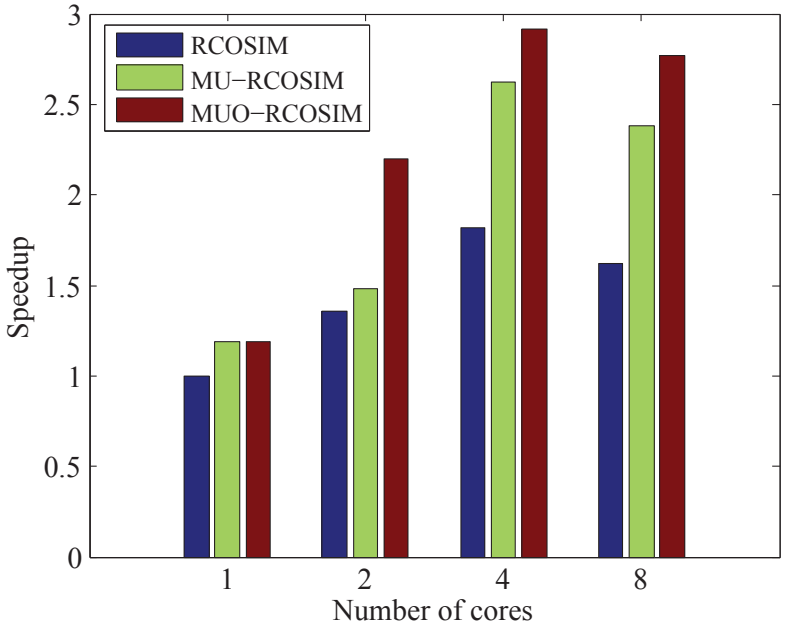


Figure 6.2: Speedup results

7 Conclusion and Future Work

Contents

7.1 Conclusion

7.2 Future Work

References

[1] G. E. Moore. “Cramming more components onto integrated circuits”. In: *Electronics* 38.8 (Apr. 1965), 114117.

[2] 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.

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

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

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

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

[7] Josheph Y-T. Leung, ed. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Boca Raton, FL, USA: Chapman&Hall/CRC, 2004.

[8] T. C. Hu. “Parallel Sequencing and Assembly Line Problems”. In: *Operations Research* 9 (1961), pp. 841–848.

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

[10] C. H. Papadimitriou and M. Yannakakis. “Scheduling interval-ordered tasks”. In: *SIAM Journal on Computing* 8.3 (1979), 405409.

[11] P. C. Fishburn. *Interval Orders and Interval Graphs: A Study of Partially Ordered Sets*. New York, NY.: John Wiley and Sons, Inc., 1985.

[12] 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.

[13] 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.

[14] 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), 222232.

[15] 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.

[16] 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.

[17] J.-J. Hwang et al. “Scheduling precedence graphs in systems with interprocessor communication times”. In: *SIAM Journal on Computing* 18.2 (Apr. 1989), 244257.

[18] 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.

[19] 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), 951967.

[20] 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.

[21] 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.

[22] 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.

[23] A. Iserles and S. P. Nrsett. “On the theory of parallel Runge-Kutta methods”. In: *ima journal of numerical analysisNume* 10.4 (1990), 463488.

[24] 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), 254365.

[25] H. Elmqvist et al. “Automatic GPU Code Generation of Modelica Functions”. In: *11th Int. Modelica Conf.* Versailles, France, 2015.

[26] M. Gebremedhin et al. “A Data-Parallel Algorithmic Modelica Extension for Efficient Execution on Multi-Core Platforms”. In: *9th Int. Modelica Conf.* Munich, Germany, 2012.

[27] 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.

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

[29] Jacques-Louis Lions, Yvon Maday, and Gabriel 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.

[30] Charbel Farhat and Marion 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.

[31] Matthew Emmett and Michael 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.

[32] 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.

[33] 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.

[34] M. Sjölund et al. “Towards efficient distributed simulation in Modelica using Transmission Line Modeling”. In: *3rd Int. Workshop on Equation- Based Object-Oriented Languages and Tools EOOLT*. Oslo, Norway: Linkping Univ. Electronic Press, 2010, 7180.

[35] 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.

[36] 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.

[37] 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. URL: `http://www.sciencedirect.com/science/article/pii/S1569190X14000665`.

[38] O. Kermia. “Ordonnancement temps réel multiprocesseur de tâches non préemptives avec contraintes de précédence, de périodicité stricte et de latence”. PhD thesis. Spécialité Physique: Université de Paris Sud, 2009. URL: `http://www-rocq.inria.fr/syndex/publications/pubs/theses/THOK.pdf`.

[39] 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.

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

[41] Bernard 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.

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

[43] 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), 758759.

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

[45] 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.

[46] Y. N. Sotskov, V. S. Tanaev, and F. Werner. “Scheduling Problems and Mixed Graph Colorings”. In: *Optimization* 51.3 (2002), pp. 597–624.

[47] 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.

[48] 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. URL: `http://www-rocq.inria.fr/syndex/publications/pubs/codes99/codes99.pdf`.