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 :

Scheduling of Co-simulation on Multi-core

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 platforms in order to simulate the whole system and estimate its global performances. If some 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 real-time execution without using multiprocessor architectures. FMI (Functional Mocked-up Interface), an increasingly common standard for model exchange and co-simulation, offers new opportunities for multicore 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 distribution and scheduling of models, taking into account their real-time, data dependencies and allocation constraints. Prior to this thesis, an approach has been developed at IFPEN which allows the parallelization of FMI models on multicore processors. In the first part of the thesis, improvements have been proposed to overcome the limitations of this approach. In particular, new algorithms were proposed in order to allow handling models that exchange data at different rates and schedule them on multicore processors. The 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 multicore processor. Finally, the proposed improvements allow handling some specific constraints like mutual exclusion constraints and allow also more accurate measurements of the execution times of the models using a profiling technique. 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

2.1 Modeling and Simulation

The complex nature of Cyber Physical 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 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 Modeling

Modeling a system consists in creating an 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 this thesis, we focus on the the modeling of dynamical systems using Ordinary Differential Equations (ODEs). Given these equations, the system can be characterized by a number of state variables. The time derivative of each variable is then expressed as a function of the other variables and the inputs. As a simple example, the fall of a punctual body subject only to gravity can be described by the differential equation: o

2.1.2 Simulation

2.1.3 Co-simulation

2.1.4 Co-simulation with Real-time Constraints

2.2 Parallel Computing

2.2.1 Overview

2.2.2 Types of Parallelism

Different kinds of parallelism exist: bit-level, instruction-level, data, and task parallelism.

Bit-level parallelism

Bit-level parallelism is the lowest level of parallelism and is related to the word size of the processor. If a 8-bit processor needs to perform computation on 16-bit data, it has to execute it in at least two steps, first on the 8 lower-order bits of the data and then on the 8 higher-order bits. Increasing the word size of the processor to 16 would allow the computation to be executed in one instruction.

Instruction-level parallelism

Instruction-level parallelism allows the execution of more than one instruction per clock cycle if the instructions don't depend on each other. An example of instruction-level parallelism is instruction pipeline in which an instruction is divided into several steps that can be executed in parallel.

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.

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 contrast to bit-level and instruction-level parallelism, data and task parallelism is apparent to the programmer who should make the decision of how to divide and distribute the data or the tasks of the application. In this thesis, we are interested in particular in task parallelism because the applications that we deal with are more suitable for this kind of parallelism.

2.2.3 Types of Parallel Computers

From a hardware point of view, parallel machines can be classified according to the type of interconnection and communication between the different processors.

multicore computing

A multicore processor contains multiple processing elements, called cores, on the same chip. The different cores can execute instructions simultaneously and thus accelerate the execution time of a program. The different cores are physically close to each other and share the same memory. A multicore processor's computation power can be increased by adding more cores. In order to take advantage of multicore processors, appropriate programming paradigms need to be used.

Distributed computing

A distributed computer has several processing elements that are connected by a network and that has each a memory. We talk then of distributed memory.

Symmetric multiprocessing

A symmetric multiprocessor computer has several processors that share memory and that are connected by a bus.

Cluster computing

A cluster is composed of a number of interconnected optionally heterogeneous machines. If the machines are not identical, balancing the load among the machines becomes difficult.

Massively parallel computing

A massively parallel processor (MPP) has a number of processors that are interconnected by means of a network. MPP differ from clusters in that they use specialized networks to connect processors whereas clusters use common networks to connect standalone machines.

Grid computing

Grid computing is the most distributed kind of parallel computing. A grid uses remote machines connected via the internet.

General-purpose computing on graphics processing units

Graphics Processing Units (GPU) are usually used to perform graphics computations on a computer. General-Purpose computing on Graphics Processing Units (GPGPU) consists in using GPUs to perform computing of applications, different than graphics, which are usually executed by Central Processing Units (CPUs). This form of parallel computing is very efficient for data-parallel applications such as matrix operations.

2.2.4 Parallel Programming

In order to efficiently use parallel machines, many programming approaches have been developed such as libraries, APIs, and programming models. Basically they differ according to the targeted type of memory, i.e. shared memory, distributed memory or shared distributed memory. In a shared memory model, communication is performed by manipulating variables in the shared memory. On the other hand, in a distributed memory model, communication is done by message passing. We present here, one of the most used shared memory libraries (Open Multi-Processing) and one of the most used distributed memory libraries (Message Passing Interface).

Open Multi-Processing

Open Multi-Processing (OpenMP) [**openmp**] 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.

Message Passing Interface

Message Passing Interface (MPI) [**mpi**] is a standard for programming distributed memory parallel computers. It is based on the concept of message passing between threads that are running in parallel on different processors. 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 processing elements. 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.

2.2.5 Parallel Scheduling

One of the most important and central topics in the parallel computing field is automatic parallelization of applications. Automatic 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 and its general behavior to be known. In general, a model of a program can be made by dividing the program into tasks of computations and defining dependencies between them. 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 automatic parallelization consists in defining a schedule for the different tasks, i.e. an allocation to a processor and an execution order for 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. Scheduling is considered to have a crucial role in computing especially with the advent of parallel computing. A significant part of the research carried out in the scheduling theory field treats problems related to scheduling computational tasks on parallel machines. We focus in this section on the state of the art of scheduling from a computing point of view.

In a scheduling problem, the resources are the processors (or cores of a multicore processor) and the tasks are the computation functions of the application to be executed. Resources are traditionally referred to as 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.

Basically, a scheduling problem is characterized by a set of n tasks $T = \{t_1, t_2, \ldots, tn\}$ and a set of m processors $P = \{p_1, p_2, \ldots, p_m\}$. Scheduling means 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. 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 added depending on the problem.

In scheduling problems, processors can be classified based on their speed of execution [**davis:2011**]:

- *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 task's execution 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 dependencies 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 [**leung:2004**]. Makepsan is the time needed by a machine to process a set of tasks. We focus here on the makespan objective because our goal is to accelerate the execution of simulations and thus to minimize the needed execution time.

The sets of tasks are usually described by Directed Acyclic Graphs (DAGs) $G(V, E)$ where each task is represented by a vertex in V and precedence constraints are represented by edges in V. A vertex may have one or more incoming edges which connect it with its parents and one or more outgoing edges which connect it with its children. A task cannot start its execution unless all its parents have finished the execution. If a vertex has no parent it is called an entry

or source vertex. A vertex that has no child is called an exit or sink vertex. The vertices may be weighted by the execution times of the corresponding tasks (see Figure **??**).

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 dependencies 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 and distribution constraints if there are any and the timing parameters of the different functions, like their execution times and periods.

Having both the functional and non functional specifications, we can deduce the "potential" and the "effective" parallelisms. The potential parallelism is related to the functional specification. It is defined by the functions that are not dependent because they can be executed in parallel, for example 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. In fact, this falls in the scope of Amdahl's law because it describes how hardware parallelism limits the exhibition of the application parallelism.

A well-known algorithm in the literature to minimize the makespan of a graph with no transitive arcs is Hu's algorithm [**hu:1961**]. 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 [**coffman:1972**] 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:1979**] dealt with 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 [**fishburn:1985**]. 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. [**adam:1974**] presented a number of level-based algorithms for scheduling task DAGs among which: The 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. [**kasahara:1984**] proposed a similar algorithm with the improvement of breaking ties by selecting the vertex with the largest number of successors. [**shirazi:1990**] proposed two algorithms: the Heavy Node First algorithm is based on a local analysis of the vertices at each level and allocates the heaviest vertice first. The second algorithm, WL (Weighted Length), considers a global view of the DAG by taking into account the relationships among the nodes at different levels. [**kruatrachue:1987**] proposed the ISH algorithm. The main idea of ISH is to fill the "scehduling holes" which are the idle time slots as the schedule is being constructed. The MCP (Modified Critical Path) algorithm proposed by

[**wu:1990**] 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. [**hwang:1989**] Earliest Start Time algorithm 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 [**sih:1993**] 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 an exit 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. [**yang:1994**] presented 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 Genetic Algorithms (GA) [**hou:1994**, **wu:2004**, **omara:2010**].

2.2.6 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 realease 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 strict 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, 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:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{2.1}$$

$$\Delta = \sum_{i=1}^{n} \frac{C_i}{D_i} \tag{2.2}$$

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

- Fixed priorities
  - Rate Monotonic (RM): Tasks are assigned priorities inversely proportional to their periods. The schedulability test of RM is: $U \leq n(2^{\frac{1}{n}} - 1)$, $D_i = T_i$.
  - 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 scehdule 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-procesor 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 doesn't 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

In this section we breifly review some of the approaches in the literature on the parallelization of simulations. We present also some of the available simulation tools that support parallel simulations.

2.3.1 Methods

In order to achieve simulation acceleration using multicore execution, different approaches are possible and were already explored. Some approaches seek to parallelize the calculation methods, for example a multi-satge solver requires several computations within one integration step. It

is then possible to perform multiple computations in parallel within one step [**iserles:1990**]. Another approach consists in parallelizing operations on vectors for ODEs resolution like in the PVODE solver [**byrne:1999**] implemented using MPI. It is possible to modify the model design in order to prepare its multicore execution, for example by using marked functions as in [**Elmqvist2015**, **Gebremedhin2012**]. If providing OpenMP ready libraries is possible, the key feature for simulation acceleration is to provide techniques which offer speedup whatever the model is. Proposing parallel solvers or automatic parallel executions of model equations as in [**Elmqvist2014**] is also an efficient way. [**clauberg:2012**] presented an approach for the parallelization of simulations on shared memory multiprocessors using OpenMP to parallelize matrix operations. Transmission Line Modeling [**hui:1990**] is a method that allows the decoupling and the parallelization of models by representing them using transmissionline graphs. Decoupling points are chosen where variables change slowly because the models are considered as if they were connected by constants at these points. [**sjolund:2010**, **braun:2012**] are based on the TLM method. As shown in [**Benkhaled__A__2012__ECOSM**], splitting a model into several FMUs, by isolating discontinuities, may reduce the simulation time, even in the case of a monocore execution. [**BenKhaled201479**] presented the RCOSIM approach. 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 multicore schedule.

2.3.2 Tools

In this section, we present some of the available simulation tools that support parallel execution.

xMOD

xMOD[1] is the modeling and simulation software developed by IFP Energies nouvelles. It supportss FMI and provides a heterogeneous model integration environment for models built by different persons using different languages and tools. xMOD can execute models embedding different solvers at different step-times on multicore architectures enabling the acceleration of complex simulations execution. The parallelization approach of xMOD is based on scheduling graphs of tasks. The approach is detailed in chapter **??**. Figure **??** shows an example of co-simulation models connected in xMOD.

MATLAB Simulink

Simulink[2], developed by Mathworks is a graphical modeling environment. It is intended to simulate dynamical systems that can be described usisng block diagrams. Simulink allows parallel execution by allowing the partitioning of the models and mapping them to a multicore processor in such a way to balance the computational load. The user can configure the parallelization using a graphical interface.

Dymola

Dymola[3], developed by Dassault Systèmes AB, is a modeling and simulation environnement based on the open Modelica modeling language. Dymola can parallelize the computations of

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

model equations. It finds which equations can be be parallelized and automatically introduces OpenMP directives. The prallelization process is detailed in [**Elmqvist2014**].

Amesim

Amesim[4] is a modeling and simulation software developed by Siemens PLM Software. 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 multicore processors.

Hopsan

Hopsan[5] is a free multi-domain system simulation tool developed at the division of Fluid and Mechatronic Systems at Linköping university. The parallelization approach of Hopsan is based on the TLM method [**sjolund:2010**, **braun:2012**].

MBSim

MBSim[6] is a simulation environment for multibody systems developed at the Institute of Applied Mechanics of the Technische Universität München. It is able to run simulations on multicore processors by using OpenMP library to parallelize matrix operations [**clauberg:2012**].

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

3
Problem Position

4

Dependence Graph Models 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 the said 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 FMU Dependence Graph

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 dependency between different pieces of the code. 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. An 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 *head* vertex and v_j is called the *tail* vertex.

The dependence graph defines the partial order of execution of the tasks which describes the potential parallelism of the program. Figure **??** shows an example of a task dependence graph of size 5.

The co-simulation of FMUs lends itself to the dependence graph representation as shown hereafter. According to the FMI standard, the code of an FMU can be exported in the form of source code or as precompiled binaries. However, most FMU providers tend to adopt the latter option for proprietary reasons. We are thus interested in this case. The method for automatic parallelization of FMU co-simulation that we propose in this thesis is based on representing the co-simulation as a dependence graph. We present in the rest of this section how this graph is constructed and a set of attributes that characterize it. The graph construction and characterization method is part of the RCOSIM approach as presented in [**benkhaled:2014**].

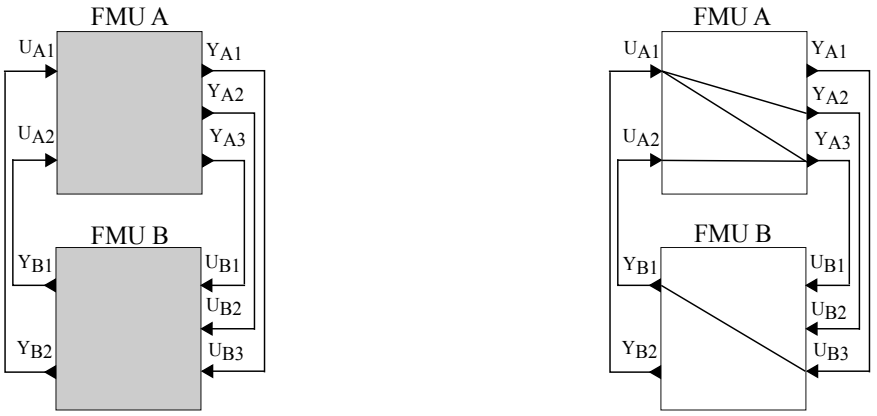
4.1.1 Construction of the Dependence Graph of FMU Co-Simulation

The entry point for the construction of an FMU dependence graph 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: Dependencies 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 dependencies. 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 an instant k need to be computed before the computation of the state at the step $k + 1$.

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



(a) Inter-FMU dependencies specified by the user **(b)** Intra-FMU dependencies provided by FMI

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

each vertex $o_i \in V_I$ represents one operation and each oriented edge $(o_i, o_j) \in A_I$, represents a precedence relation between two operations. The operation graph is built by exploring the relations between the FMUs and between the operations of the same FMU. A vertex is created for each operation and arcs are then added between vertices if a precedence dependence exists between the corresponding operations. If FMI 1.0 is used, which does not give information about the dependencies between the state variables computation and the input and output variables computations, we must add edges 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 as for all the outputs belonging to the same FMU. Running the co-simulation consists in executing the graph repeatedly. A new execution of the graph cannot be started unless the previous one was totally finished. The operation graph corresponding to the FMUs of Figure 4.1 is shown in Figure 4.2.

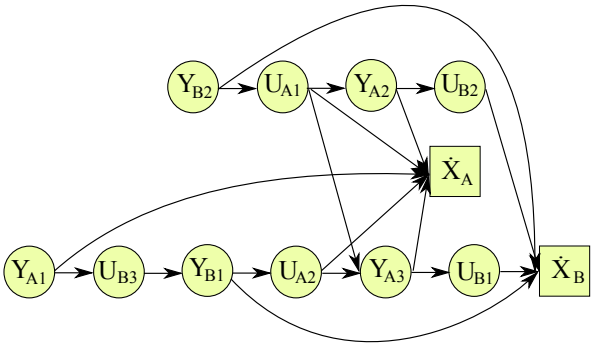


Figure 4.2: Operation graph obtained from the FMUs of Figure 4.1

4.1.2 Dependence Graph Attributes

The dependence 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 dependence 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 FMU dependence 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. $s_p(o_i) \in \{input, output, state\}$. 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 4.3 and 4.4 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 $\overline{E}(o_i)$ and the latest start time form end $\overline{S}(o_i)$ are defined by equations **??** and **??** respectively. $\overline{E}(o_i)$ defines the latest time by which the operation o_i must finish its execution so as not to increase the total execution time of the graph. For o_i to finish its execution no later than $\overline{E}(o_i)$, it has to start its execution at the latest at $\overline{S}(o_i)$.

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

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

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 the it. The length of the critical path of the graph R is defined by equation 4.5. The critical path is a very important characteristic of the dependence graph. It defines a lower bound on the execution time of the graph, i.e. in 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. The flexibility expresses a windows of time where the 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 Multi-rate FMU Dependence Graphs

The FMU dependence graph model presented in the previous section 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 some FMUs of one co-simulation are provided by different parties. It is very common that FMU providers design 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 FMU dependence graph model has to be extended in order to accommodate multi-rate data exchange between operations.

For a multi-rate co-simulation, the dependence graph that is constructed as described in the previous section is referred to as a multi-rate dependence graph. A way for making such dependence graph suitable for multi-core scheduling is to tranform it to a mono-rate graph. This section presents an algorithm that transforms the initial multi-rate operation graph $G(V, A)$ into a mono-rate 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 [**kermia:2009**, **ramamritham:1995**].

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. 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 dependencies: 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.

Algorithm 1: Multi-rate graph transformation algorithm

Input: Initial operation graph $G_I(V_I, A_I)$;
Output: Transformed operation graph $G_M(V_M, A_M)$;
Set $G_I(V_I, A_I)$ the initial multi-rate 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

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.

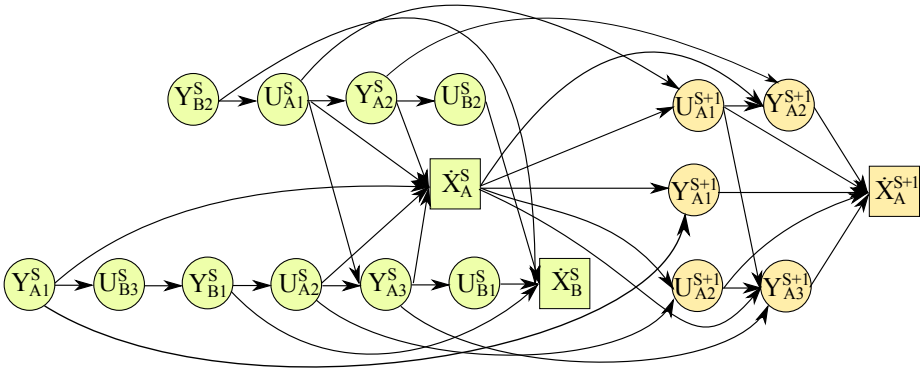


Figure 4.3: Graph obtained by applying the multi-rate transformation algorithm on the graph of Figure 4.2

4.3 Dependence Graph with Mutual Exclusion Constraints

4.3.1 Motivation

The FMI standard states that *"FMI functions of one instance don't need to be thread safe"*. Therefore, the 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.

In order to study the impact of these constraints, we have tested two mutual exclusion strategies and the performance obtained using each of them has been evaluated. The first one uses a dedicated mutex (system object that guarantees mutual exclusion) for each FMU: Every time an FMU function call is made at runtime, the associated mutex have to be acquired before the execution of the function code can be started. The second solution is explained in [**BenKhaled201479**] and consists in allocating the operations of a same FMU to the same core (constrained allocation). Thanks to SynDEx, it is easily possible to theoretically estimate the impact of using the constrained allocation in the multi-core scheduling heuristic. Results are given in Figure **??**. 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 5, but similar when 5 cores or more are available. When using less than 5 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 5 cores or more.

In order to compare the two mutual exclusion strategies, we implemented them in xMOD. Execution times measurements were performed by getting the system time stamp at the beginning of the simulation and after 30 seconds of the simulated time. As previously, we compare the speed-up by dividing the mono-core simulation execution time by the simulation execution time on a fixed number of cores. Figure 4.4 sums up the results, where unconstrained allocation corresponds to the use of mutex 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 in xMOD of the constrained allocation gives similar results in terms of speed-up improvement when increasing the number of cores until 5. 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 estimating any synchronization cost between cores. The real implementation in xMOD contains synchronization objects between operations to ensure the consistency of data dependencies which certainly have an important impact on the speed-up.

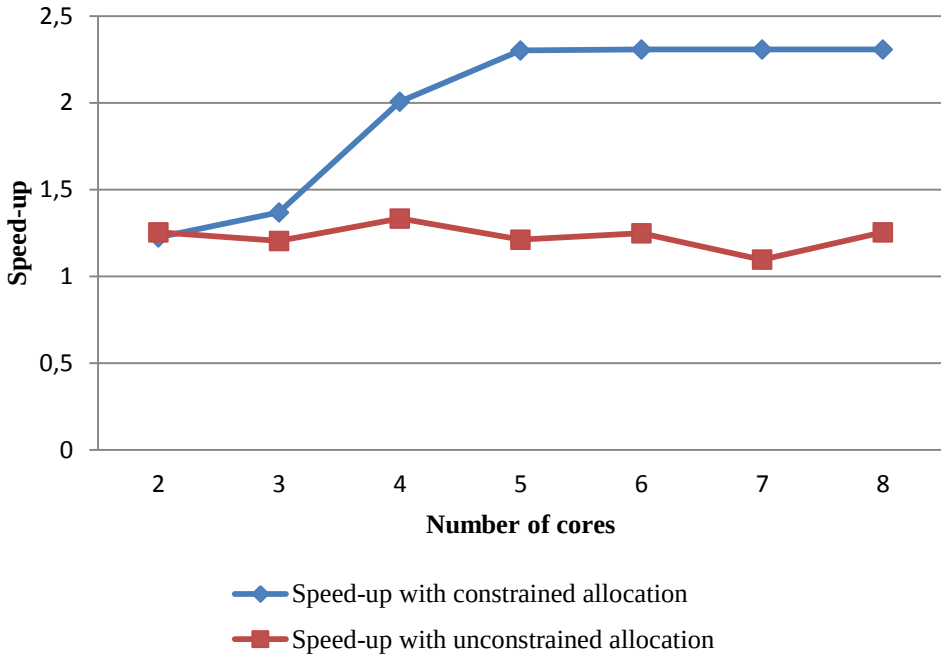


Figure 4.4: Measured speed-up.

4.3.2 Acyclic Orientation of Mixed Graphs

The restrictions introduced by employing the mutual exclusion techniques shown in the previous section 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. As such, the mutual exclusion constraints need to be modeled in the dependence graph of the co-simulation. This is possible using *mixed graphs*. A mixed graph is a graph which has both directed and undirected edges connecting pairs of its vertices. In the scheduling literature, these graphs are know as disjunctive graphs also. In order to distinguish between the two types of the edges, we conventionally refer to directed edges as arcs and to undirected edges as edges. 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. Operations that are connected by an edge 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 dependence 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 NP-hard and is equivalent to finding an optimal coloring of the initial graph. In its general form, i.e. when all edges of the graph are undirected, graph coloring consists in labeling the vertices of the graph with integers, called colors, such that adjacent vertices (connected by an edge) are labeled with different colors. The orientation can than be performed by orienting each edge from the vertex with the smaller color to the vertex with the greater color.

4.3.3 Problem Formulation

Mutual exclusion relations between operations of the same FMU are represented by edges in the mixed graph $G_X(V_X, A_X, D_X)$, where V_X is the set of vertices (operations), A_X the set of arcs, and D_X the set of edges. A mutual exclusion relation between the pair of vertices o_p and o_q is denoted by an edge $[o_p, o_q] \in D_X$ which means that o_p and o_q have to be executed in disjoint time intervals but in either order ($[o_p, o_q] = [o_q, o_p]$). Choosing one order or another does not impact the numerical results of the co-simulation since these operations do not have data dependencies. Still, we have to ensure mutual exclusion between them due to the non-thread-safe implementation of FMI. The goal here is to assign directions to the edges of D_X in order to define an order of execution between each pair of operations joined by an edge with the aim of minimizing the length of the critical path of the resulting graph, and subject to the constraint that no cycle is generated in the obtained graph. The output of the heuristic is a graph $G_F(V_F, A_F, \emptyset) = G_F(V_F, A_F)$. The problem of acyclic orientation of mixed graphs has been studied and shown to be NP-Hard in [**andreev:2000**, **sotskov:2002**, **al-anzi:2006**]. In these studies, only the case of operations with unit (or equal) execution times were considered and the acyclic orientation problem was interpreted as a graph coloring problem. We consider operations with different execution times and use an approach which takes into account that only operations belonging to the same FMU are joined by edges. The problem can be stated as an optimization formulation as follows:

Input	Mixed dependence graph of operations $G_X(V_X, A_X, D_X)$
Output	DAG $G_F(V_F, A_F)$
Find	Assignment of colors cr_i to vertices o_i, $\alpha : V \rightarrow \mathbb{N}$
Minimize	Critical path of graph $G_F(V_F, A_F) : P = max(E(o_i))_{o_i \in V}$
Subject to	No cycle generated in $G_F(V_F, A_F)$

4.3.4 Resolution using Linear Programming

We present here an ILP formulation for solving the problem of acyclic orientation of mixed graphs. We express the problem as a graph coloring problem where the operations can have different execution times. Tables 4.1 and 4.2 summarize the variables and the constants that are used respectively.

We define an integer variable $x(o_i)$, which represents the color that is assigned to operation o_i. The inequality 4.9 expresses the constraint that every operations must be assigned a color that is greater than the colors of all its predecessors.

$$x(o_i) > x(o_j) : \forall (o_j, o_i) \in A \tag{4.9}$$

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

Variable	Type	Description
$x(o_i)$	Integer	Color assigned to operation o_i

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

Constant	Type	Description

The binary variable y_{ij} is associated with the edge $[o_i, o_j]$ and indicates whether operation o_i is assigned a color greater than the color assigned to o_j or not, i.e. $y_{ij} = 1\ if x(o_i) > x(o_j), 0\ otherwise$. Note that y_{ij} is the complement of y_{ji}. For every edge $[o_i, o_j]\ in E$, the colors assigned to o_i and o_j must be different. Expressions 4.10 and 4.11 capture this constraint. M_1 is a big constant.

$$x(o_i) > x(o_j) - M_1 \times (1 - y_{ij}) + 1 : \forall [o_i, o_j] \in E \tag{4.10}$$

$$x(o_j) > x(o_i) - M_1 \times y_{ij} + 1 : \forall [o_i, o_j] \in E \tag{4.11}$$

Next, we need to express the computations of the timing parameters of the operations, i.e. start and end times. Recall that for a given edge $[o_i, o_j]$, if o_j is assigned a color greater than the color assigned to o_i, then the edge $[o_i, o_j]$ is assigned a direction form o_i to o_j. Expressions 4.12 and 4.13 are used to compute the start time of each operation and expression 4.14 is used to compute the end time of each operation.

$$S(o_i) \geq E(o_j) : \forall (o_j, o_i) \in A \tag{4.12}$$

$$S(o_i) \geq E(o_j) - M2 \times (1 - y_{ij}) : \forall [o_i, o_j] \in E \tag{4.13}$$

$$E(o_i) = S(o_i) + C(o_i) : \forall o_i \in V \tag{4.14}$$

The critical path P is computed using the inequality 4.15

$$P \geq E(o_i) : \forall o_i \in V \tag{4.15}$$

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

$$min(P) \tag{4.16}$$

4.3.5 Acyclic Orientation Heuristic

We describe in this section a heuristic for handling mutual exclusion constraints between operations belonging to the same FMU. Without loss of generality, the superscript which denotes the number of the occurrence of an operation is not used in this section for the sake of clarity. Each vertex of the graph $G_M(V_M, A_M)$ represents an operation that we refer to using the notation o_i.

This mixed graph is obtained from the graph $G_M(V_M, A_M)$ by adding edges between operations belonging to the same FMU. Note that $V_X = V_M$.

The orientation of the edges of D_X means applying a function $\alpha : \{[o_i, o_j] \in D_X\} \rightarrow \{(o_i, o_j), (o_j, o_i)\}$. Without loss of generality, consider that $\alpha([o_i, o_j]) = (o_i, o_j)$, then o_i is added to $\overline{\Gamma}(o_j)$ and o_j is added to $\Gamma(o_i)$. Consequently, the inequality $E(o_i) \leq S(o_j)$ holds and therefore o_i and o_j are executed in disjoint time intervals. The first step of the heuristic consists in building the mixed graph $G_X(V_X, A_X, D_X)$ by adding edges between operations belonging to the same FMU. The output of the heuristic is a topological ordering of the operations of each FMU. This can be interpreted for each FMU $\mathcal{M}$ as an assignment of integers $\phi : V_F \rightarrow \{1, 2, \ldots, m\}$ defining the order of execution for these operations, where m is the number of operations of the FMU $\mathcal{M}$. We call the integer assigned to an operation its rank.

Algorithm 2 details the acyclic orientation heuristic. After adding the edges to the graph, the heuristic iteratively assigns ranks to the operations. It keeps a list of already ordered operations l for each FMU $\mathcal{M}$ and at each main iteration it selects the operation which has the earliest start time $S(o_i)$ to be added to the list of its FMU. Ties are broken by selecting the operation with the least flexibility $F(o_i)$. We call the operation to be ordered at a given iteration, the pending operation. The the pending operation o_i is assigned the ranks $\tau \in \{x, x+1, \ldots, length(l)+1\}$ in ascending order, where $x = \max_{o_j \in \overline{\Gamma}(o_i)}(\phi(o_j)+1)$ $(1\ if\ \overline{\Gamma}(o_i) = \emptyset)$. For each rank τ, the assignment $\phi(o_i) = \tau$ is made and the ranks assigned to all the already ordered operations $o_{i'} \in l$ such that $\phi(o_{i'}) \geq \phi(o_i)$ are increased $\phi(o_{i'}) = \phi(o_{i'}) + 1$. Then, every edge $[o_i, o'_i] \in D_X : o'_i \in l$ is assigned a direction from the operation with the lower rank to the operation with the higher rank. At this point, the increase in R, the critical path of the graph, is evaluated. Next, the operations $o_{i'} \in l$ such that $\phi(o_{i'}) > \phi(o_i)$ are reassigned their previous ranks $\phi(o_{i'}) = \phi(o_{i'}) - 1$, and the pending operation is assigned the next rank $\phi(o_i) = \phi(o_i) + 1$. The increase in the critical path is evaluated again similarly. After repeating this for all the ranks $\tau \in \{x, x+1, \ldots, length(l)+1\}$, the pending operation is finally ordered at the rank which leads to the least increase in the critical path and edges $[o_i, o'_i] \in D_X : o'_i \in l$ are assigned directions as described before. The heuristic begins another iteration by selecting a new operation to be ordered. The heuristic assigns a rank to one operation at each iteration. Every operation is assigned a rank 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 ranks.

Complexity

Soundness

Completeness

Termination

4.4 Dependence Graph with Real-time Constraints

Algorithm 2: Acyclic orientation heuristic

Input: Operation graph $G_M(V_M, A_M)$;
Output: New operation graph $G_F(V_F, A_F)$;
Set $G_M(V_M, A_M)$ the operation graph, $G_X(V_X, A_X, D_X)$ the mixed graph, and $G_F(V_F, A_F)$ the final graph;
$V_X \leftarrow V_M$; $A_X \leftarrow A_M$; $D_X \leftarrow \emptyset$; $V_F \leftarrow V_M$; $A_F \leftarrow A_M$;
Set Ω the set of all the operations: $\Omega \leftarrow V_X$;
foreach *pair* o_i, o_j *such that* $M(o_i) = M(o_j)$ **do** Add the edge $[o_i, o_j]$ to D_X: $D_X \leftarrow D_X \cup \{[o_i, o_j]\}$;
while $\Omega \neq \emptyset$ **do**
  Select the operation $o_i \in \Omega$ whose start time $S(o_i) = \max_{o_j \in \Omega}(S(o_j))$. Break ties by selecting the operation with the least flexibility;
  Set $\mathcal{M} \leftarrow M(o_i)$ and l the list of the already ordered operations of FMU $\mathcal{M}$;
  Set $\sigma \leftarrow \infty$; (Initialize the increase in the critical path);
  Set $x = \max_{o_j \in \overline{\Gamma}(o_i)}(\phi(o_j) + 1)$ $(1\ if\ \overline{\Gamma}(o_i) = \emptyset)$;
  for $\tau = x$ *to* $length(l) + 1$ **do**
    $\phi(o_i) = \tau$;
    foreach $o_{i'} \in l$ *such that* $\phi(o_{i'}) \geq \phi(o_i)$ **do** $\phi(o_{i'}) \leftarrow \phi(o_{i'}) + 1$;
    foreach $o_{i'} \in l$ **do**
      if $\phi(o_{i'}) > \phi(o_i)$ **then** Assign a direction to the edge $[o_i, o_{i'}] \in D_X : \alpha([o_i, o_{i'}]) \leftarrow (o_i, o_{i'})$;
      else Assign a direction to the edge $[o_i, o_{i'}] \in D_X : \alpha([o_i, o_{i'}]) \leftarrow (o_{i'}, o_i)$;
    end
    Compute the new critical path and set σ' the increase in the critical path;
    if $\sigma' < \sigma$ **then** $rank \leftarrow \tau$, $\sigma \leftarrow \sigma'$;
    foreach $o_{i'} \in l$ *such that* $\phi(o_{i'}) > \phi(o_i)$ **do** Reassign $o_{i'}$ its previous rank: $\phi(o_{i'}) \leftarrow \phi(o_{i'}) - 1$;
  end
  $\phi(o_i) = rank$;
  foreach $o_{i'} \in l$ **do**
    if $\phi(o_{i'}) > \phi(o_i)$ **then** Assign a direction to the edge $[o_i, o_{i'}] \in D_X : \alpha([o_i, o_{i'}]) \leftarrow (o_i, o_{i'})$;
    else Assign a direction to the edge $[o_i, o_{i'}] \in D_X : \alpha([o_i, o_{i'}]) \leftarrow (o_{i'}, o_i)$;
  end
  Remove o_i from Ω;
end

5

Multi-core Scheduling of Dependence Graphs

Contents

This chapter presents methods for scheduling an FMU dependence graph on a multi-core architecture. Once the dependence 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 dependence graph with the goal of accelerating the execution of co-simulation. Second, we consider scheduling the dependence graph while satisfying real-time constraints.

5.1 Scheduling of FMU 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 a mapping of the dependence 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. The scheduling algorithm makes decisions on mapping 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	Dpenendence graph of operations $G_F(V_F, A_F)$
Output	Offline Schedule of operations on multi-core processor
Find	Mapping 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.

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

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

The cost of synchronization is taken into account according to the synchronization model described in section **??**. 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.

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

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

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

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

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

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

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.

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

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

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

Objective function

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

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

5.1.3 Multi-core Scheduling Heuristic

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

A variety of list multi-core scheduling heuristics exist in the literature and each heuristic may be suitable for some specific kinds of multi-core scheduling problems. We detail in this section a heuristic that we have chosen to apply on the 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 dependencies 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 [**grandpierre:1999**] 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 *rho* the cost function and call it the schedule pressure. It expresses the degree of criticality of scheduling an operation.

The heuristic considers the different timing parameters 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. 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 3: 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 ‰ ≠ ∅ 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 FMU dependency 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 dependencies 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 dependence graph can then be visualized on a grid of levels as depicted in figure **??**. The generator makes use 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 operation 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 dependencies. 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 lvl $lvl' < lvl-1$ 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

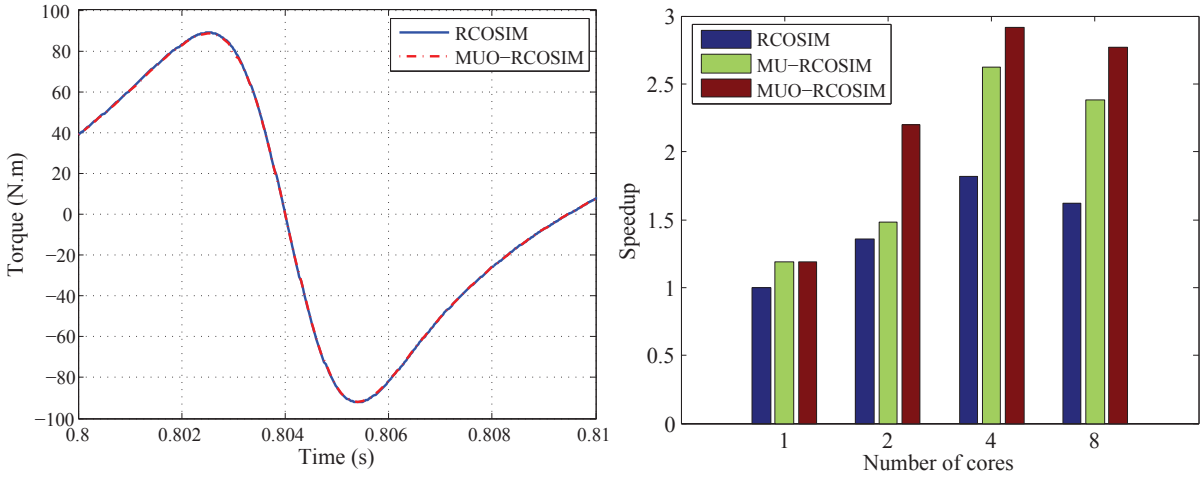
The proposed parallelization approach has been applied on an industrial use case. In this section, we give a description of the use case and then present the obtained results.

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 citebenkhaled:2014 to give a very good accuracy of the numerical results. Figure 6.1a 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 6.1b 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.



(a) Numerical results

(b) Speedup results

Figure 6.1: Speedup and numerical results of e-RCOSIM

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.

7 Conclusion and Future Work

Contents

7.1 Conclusion

7.2 Future Work