A method for parallel scheduling of multi-rate co-simulation on multi-core platforms

Salah Eddine Saidi, Nicolas Pernet, Yves Sorel

To cite this version:

Salah Eddine Saidi, Nicolas Pernet, Yves Sorel. A method for parallel scheduling of multi-rate co-simulation on multi-core platforms. Oil & Gas Science and Technology - Revue d'IFP Energies nouvelles, 2019, 74, pp.49. 10.2516/ogst/2019009 . hal-02141707

HAL Id: hal-02141707

https://ifp.hal.science/hal-02141707

Submitted on 28 May 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Oil & Gas Science and Technology - Rev. IFP Energies nouvelles **74**, 49 (2019)
© S.E. Saidi et al., published by IFP Energies nouvelles, 2019
https://doi.org/10.2516/ogst/2019009

Available online at:
ogst.ifpenergiesnouvelles.fr

Numerical methods and HPC
A. Anciaux-Sedrakian and Q. H. Tran (Guest editors)

REGULAR ARTICLE OPEN ACCESS

A method for parallel scheduling of multi-rate co-simulation on multi-core platforms

Salah Eddine Saidi[1], Nicolas Pernet[1,*], and Yves Sorel[2]

[1] IFP Energies nouvelles, 1-4, avenue de Bois-Préau, 92852 Rueil-Malmaison Cedex, France
[2] Inria centre de Paris, 2 rue Simone Iff, 75012 Paris, France

Received: 24 February 2018 / Accepted: 11 February 2019

Abstract. The design of cyber-physical systems is a complex process and relies on the simulation of the system behavior before its deployment. Such is the case, for instance, of joint simulation of the different subsystems that constitute a hybrid automotive powertrain. Co-simulation allows system designers to simulate a whole system composed of a number of interconnected subsystem simulators. Traditionally, these subsystems are modeled by experts of different fields using different tools, and then integrated into one tool to perform simulation at the system-level. This results in complex and compute-intensive co-simulations and requires the parallelization of these co-simulations in order to accelerate their execution. The simulators composing a co-simulation are characterized by the rates of data exchange between the simulators, defined by the engineers who set the communication steps. The RCOSIM approach allows the parallelization on multi-core processors of co-simulations using the FMI standard. This approach is based on the exploitation of the co-simulation parallelism where dependent functions perform different computation tasks. In this paper, we extend RCOSIM to handle additional co-simulation requirements. First, we extend the co-simulation to multi-rate, *i.e.* where simulators are assigned different communication steps. We present graph transformation rules and an algorithm that allow the execution of each simulator at its respective rate while ensuring correct and efficient data exchange between simulators. Second, we present an approach based on acyclic orientation of mixed graphs for handling mutual exclusion constraints between functions that belong to the same simulator due to the non-thread-safe implementation of FMI. We propose an exact algorithm and a heuristic for performing the acyclic orientation. The final stage of the proposed approach consists in scheduling the co-simulation on a multi-core architecture. We propose an algorithm and a heuristic for computing a schedule which minimizes the total execution time of the co-simulation. We evaluate the performance of our proposed approach in terms of the obtained execution speed. By applying our approach on an industrial use case, we obtained a maximum speedup of 2.91 on four cores.

1 Introduction

The recent advancement in merging different technologies and engineering domains has led to the emergence of the field of Cyber-Physical Systems (CPS) [1]. In such systems, embedded computers interact with, and control physical processes. Because of the heterogeneity of the involved components (multi-physics, sensors, actuators, embedded computers), CPS may feature very complex architectures and designs, ever raising the need for time, cost, and effort-effective approaches for building robust and reliable systems. Numerical simulation has proven successful in responding to this need, and is therefore increasingly considered to be an indisputable step in design verification and validation. For complex CPS, experts of different engineering disciplines may be involved in the design process by developing models and simulators for different parts of the system. In a later stage, the developed simulators are coupled together in order to perform what is called simulation at the system level [2]. This approach is called co-simulation [3]. Each simulator is assigned an integration step, which in some cases is driven by the dynamics of the modeled system and the control objective, and exchanges data with the other simulators according to a communication step which can be equal or different than its integration step. Enabling co-simulation of heterogeneous models requires using adequate tools and methods. In this scope, the Functional Mock-up Interface (FMI) [4] was developed with the goal of facilitating the co-simulation and the exchange of subsystem models and simulators. FMI defines

* Corresponding author: nicolas.pernet@ifpen.fr

This is an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

a standardized interface that can be implemented by modeling tools in order to create models and simulators that can be connected with other FMI models and simulators.

Co-simulation is an alternative approach to monolithic simulation where a complex system is modeled as a whole. Here, by complex system, we refer to systems where the controlled physical process constitutes a multi-physics system and is modeled in the continuous-time domain using (hybrid) Ordinary Differential Equations (ODEs). Because they are aimed to be implemented on embedded computers, numerical laws that control the physical process are modeled in the discrete time domain. All of these features add to the complexity of the models and simulators. 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 and simulators. In co-simulation, the equations of each Functional Mock-up Unit (FMU) are integrated using a solver separately. FMUs exchange data by updating their inputs and outputs according to their communication steps [5]. For connected FMUs with different communication steps, the master algorithm can provide extrapolation techniques to produce the missing data of the continuous part of the simulated system.

Figure 1 shows an example of the evolution of time and data exchange between two FMUs. The horizontal arrows represent the simulated time of each FMU. The vertical double arrows represent data exchange between the FMUs. In general, for each FMU, simulation is performed integration step by integration step. We consider that integration step sizes of the FMUs may differ. Also, we consider that data exchange between FMUs only happens when the simulated time of both FMUs is equal. Consequently, a communication step is defined for every couple of simulators. In addition, we consider, that this communication step is a multiple of the integration steps of both. Co-simulation requires domain-specific information for each involved FMU. Such information, which is beyond the scope of FMI, can be provided by domain experts, for example by using an approach like the one proposed by Sirin *et al.* [6]. The communication steps of FMUs can be shared as part of this information. It is worth noting that setting communication step sizes might depend on the parameters set for the FMU, and the dynamics of the inputs.

Usually, assembling FMUs results in a heavy co-simulation, requiring high processing power. Increasing CPU performance through frequency scaling has reached some technological limits leading in a stagnation in the transistor count in processors. As a consequence, during the last decade, parallelism has been by far the main way for increasing the performance of processors [7].

In fact, the last years have witnessed a major shift among semiconductor manufacturers to building multi-core processors, *i.e.* integrating multiple processors into one chip allowing parallel processing on a single computer. Enabling parallel execution of heavy co-simulations on multi-core processors is keenly sought by the developers and the users

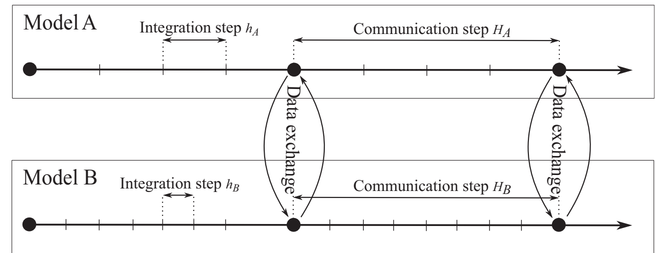


Fig. 1. Evolution of time and data exchange between two FMUs during co-simulation.

of simulation tools. However, fulfilling this objective is not trivial and appropriate parallelization techniques need to be applied on co-simulation simulators in order to accelerate their execution on multi-core processors.

In order to achieve (co-)simulation acceleration using multi-core execution, different approaches are possible and were already explored. From a user point of view, it is possible to modify the model design in order to prepare its multi-core execution, for example by using models that are developed using parallel computing libraries as in [8]. Using parallel solvers as in [9] is another alternative. In [10], authors present the DACCOSIM framework which allows multi-threaded distributed simulation of FMUs on multi-core processors as well as clusters of computers. However, the parallelization achieved through this approach is not automatic as the user has to define the distribution of the FMUs on the architecture. A more detailed discussion of related work is given in [11].

In this paper, we address the problem from a co-simulation tool provider point of view. In such a tool, the user connects different FMUs, embedding solvers or using solvers provided by the co-simulation tool. In this case, it is not possible to change the models, the solvers, nor the modeling tools.

Readers can refer to Figure 16 to figure out the main stages of the proposed solution.

The rest of the paper is organized as follows. Section 2 gives a background about the approach proposed in this paper. The dependence graph model, the graph transformation rules, algorithm and heuristic are presented in Section 3. Then Section 4 presents our multi-core scheduling proposal for co-simulation acceleration. In Section 5 we evaluate our proposed approach using both randomly generated graphs and an industrial use-case. Finally, we conclude in Section 7.

2 Background

The Refined CO-SIMulation (RCOSIM) [11] approach allows the parallelization on multi-core processors of co-simulations using the FMI standard. This approach is based on a representation of the co-simulation using a Directed Acyclic Graph (DAG). A scheduling heuristic is then used to accelerate the execution of the co-simulation.

We chose to build on the RCOSIM approach in order to achieve parallelization of FMI co-simulations for accelerated execution. Parallel execution of the functions of one

FMU is interesting since, as can be seen in Figure 3, it may not be possible to execute all the functions of one FMU and then all functions of another one. Concurrent execution is needed here which may lead to using synchronization mechanisms hurting performance. However, by exploring the search space of parallelization solutions, an efficient solution can be sought.

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

In this paper, we deal with the limitations of this approach that we highlighted in a recent work [12]. Although RCOSIM resulted in interesting co-simulation speedups, it has two limitations that have to be considered in the multi-core scheduling problem in order to obtain better performances. First, RCOSIM supports only mono-rate co-simulations, *i.e.* the different FMUs have to be assigned the same communication step size. Nevertheless, some co-simulation scenarios have groups of FMUs that are more tightly coupled than others. This leads to different communication step sizes between some FMUs of the same co-simulation. In particular, in a given co-simulation scenario, we may have small communication steps sizes to ensure accuracy between tightly coupled FMUs and large communication step sizes between loosely coupled FMUs to speed up the co-simulation. The use case we propose in Section 5 is an example of such co-simulation. Note that this work is restricted to fixed integration step sizes and communication step sizes that are greater or equal to the integration step sizes of the respective simulators. Second, the functions of an FMU may not be thread-safe, *i.e.* they cannot be executed in parallel as they may share some resource (*e.g.* variables) that might be corrupted if two operations try to use it at the same time. Consequently, if two or more operations belonging to the same FMU are executed on different cores, a mechanism that ensures these operations are executed in disjoint time intervals must be set up. It is worth noting that a co-simulation should be repeatable which would not be possible if mutual exclusion is not handled properly as this may lead to race conditions.

Previously, these mutual exclusion constraints were tackled in RCOSIM by executing the operations of the same FMU on the same core which restricts the exploitation of the parallelism.

We propose in this article a solution for each of the aforementioned limitations by making extensions to RCOSIM. In order to allow handling multi-rate co-simulations, we present a graph transformation algorithm that is applied to the initial constructed graph. Then, mutual exclusion constraints are represented by adding, in the graph, non oriented edges between operations belonging to the same FMU. In order assign directions to the added non oriented edges, we formulate the problem, propose a resolution using linear programming and finally present an acyclic orientation heuristic. Then we present a multi-core scheduling solution for such co-simulation graph. We first propose a resolution using linear programming before presenting a 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 are widely used in the context of offline multi-core scheduling.

Automatic parallelization [13] of computer programs embodies the adaptation of the potential parallelism inherent in the program to the effective parallelism that is provided by the hardware. Because computer programs are usually complex (multiple functions, nested function calls, control flow jumps, etc.), this process of adaptation requires the use of a model for abstracting the program to be parallelized. The aim of using such model is to identify which parts of the program can be executed in parallel by expressing some features of the program such as data dependence between different parts 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, *i.e.* it cannot be allocated to several computing resources.
- 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.

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, *i.e.* vertices that are not in precedence relation A which is asymmetric and transitive.

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

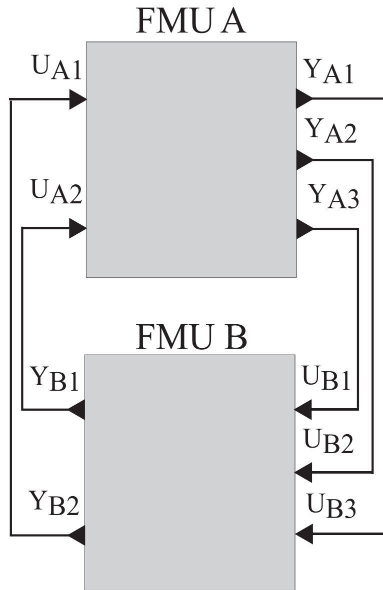


Fig. 2. Inter-FMU dependence specified by the user.

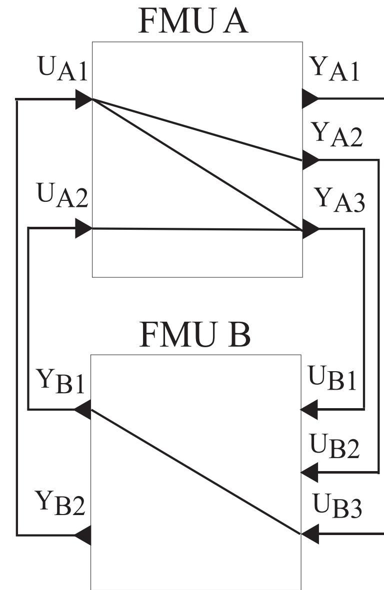


Fig. 3. Intra-FMU dependence provided by FMI.

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 article 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 [11].

2.1 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 2. Here, we consider only co-simulations where all FMUs are assigned identical communication step sizes. We refer to the graph which represents such co-simulation as *mono-rate* graph. The execution of each FMU is seen as computing a set of inputs, a set of outputs, and the state variables of the FMU. A computation of an input, output, or state variables is performed by FMU C function calls. 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 3, FMI allows the identification of Direct Feedthrough (*e.g.* Y_{B1}) and Non Direct Feedthrough (*e.g.* Y_{A1}) outputs of an FMU and other information depending on the version of the standard:

- FMI 1.0: Dependence between inputs and outputs are given. The computation of the state at a given simulation step k is considered necessary for the computation of every output 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 inputs 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 at a given simulation step depends on the state computation at the same step. Also, it is specified which input at a simulation step k needs to be computed before the computation of the state at the step $k+1$.

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

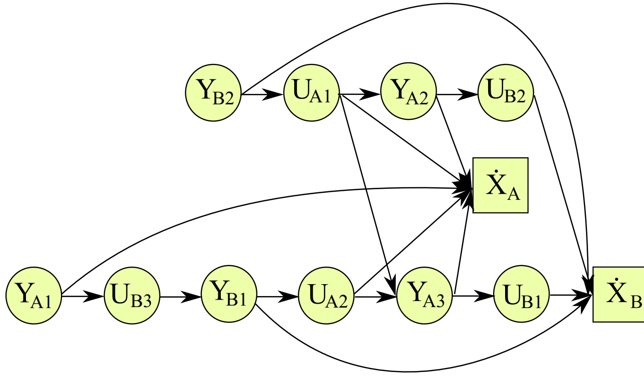


Fig. 4. Operation graph obtained from the FMUs of Figure 3.

An execution of the obtained graph corresponds to one simulation step. The operation graph corresponding to the FMUs of Figure 3 is shown in Figure 4. Note that we assume that no rollbacks are used and that the appropriate sequence of function calls is used as stated in the FMI standard.

2.2 Dependence graph attributes

The operation graph is used as input to the scheduling algorithm. In addition to the partial order defined by the graph, the scheduling algorithm uses a number of attributes to compute an efficient schedule of the operation graph. Many list scheduling algorithms are based on the Critical Path Method (CPM) [14] and use 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.* $T(o_i) \in \{\text{update}_{\text{input}}, \text{update}_{\text{output}}, \text{update}_{\text{state}}\}$. An operation o_i is characterized by its communication step $H(o_i)$ which is equal to the communication step of its FMU. 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). In the example shown by Figure 5, pred$(d) = \{b, c\}$. 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). In the example shown by Figure 5, ance $(d) = \{a, b, c\}$. 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). In the example shown by Figure 5, succ $(d) = \{b, c\}$. 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). In the example shown by Figure 5, desc$(a) = \{b, c, d\}$. 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. Let's consider that the operations shown in Figure 5 have the following execution times $C(a) = 2$, $C(b) = 2$, $C(c) = 1$, $C(d) = 4$. The earliest start time from start $S(o_i)$ and the earliest end time from start $E(o_i)$ are defined by equations (1) and (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 $E(o_i)$:

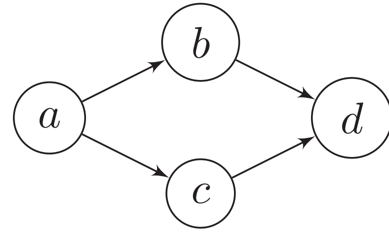


Fig. 5. Example of an operation graph.

$$S(o_i) = \begin{cases} 0, & \text{if pred}(o_i) = \emptyset \\ \max_{o_j \in \text{pred}(o_i)}(E(o_j)), & \text{otherwise,} \end{cases} \quad (1)$$

$$E(o_i) = S(o_i) + C(o_i). \quad (2)$$

In the example of Figure 5, we have: $S(a) = 0$, $S(b) = 2$, $S(c) = 2$, $S(d) = 4$ and $E(a) = 2$, $E(b) = 4$, $E(c) = 3$, $E(d) = 8$.

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 (3) and (4) respectively.

$$\overline{E}(o_i) = \begin{cases} 0, & \text{if succ}(o_i) = \emptyset \\ \max_{o_j \in \text{succ}(o_i)}(\overline{S}(o_j)), & \text{otherwise,} \end{cases} \quad (3)$$

$$\overline{S}(o_i) = \overline{E}(o_i) + C(o_i). \quad (4)$$

In the example of Figure 5, we have: $\overline{E}(a) = 6$, $\overline{E}(b) = 4$, $\overline{E}(c) = 4$, $\overline{E}(d) = 0$ and $\overline{S}(a) = 8$, $\overline{S}(b) = 6$, $\overline{S}(c) = 5$, $\overline{S}(d) = 4$.

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 (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:

$$R = \max_{o_i \in V_I}(E(o_i)). \quad (5)$$

In the example of Figure 5, we have: $R = \max\ (E(a), E(b), E(c), E(d)) = \max\ (2, 4, 3, 8) = 8$.

The flexibility $F(o_i)$ is defined by equation (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:

$$F(o_i) = R - S(o_i) - C(o_i) - \overline{E}(o_i). \quad (6)$$

In the example of Figure 5, we have $F(a) = 8{-}0{-}2{-}6 = 0$, $F(b) = 8{-}2{-}2{-}4 = 0$, $F(c) = 8{-}2{-}1{-}4 = 1$, $F(d) = 8{-}4{-}4{-}0 = 0$.

3 Extension of the dependence graph model for FMU co-simulation

This section describes our proposed extensions to the dependence graph model that is used for representing an FMU co-simulation. The transformations that it undergoes in order to represent multi-rate co-simulation and mutual exclusion constraints are presented.

3.1 Dependence graph of a multi-rate FMU co-simulation

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} $(V_{\mathrm{M}}, A_{\mathrm{M}})$. 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 to account for multi-rate scheduling problems [15].

We define the Hyper-Step (HS) as the Least Common Multiple (LCM) of the communication steps of all the operations: $\mathrm{HS} = \mathrm{LCM}\ (H(o_1), H(o_2),\ldots, H(o_n))$ where $n = |V_{\mathrm{I}}|$ is the number of operations in the initial graph. The Hyper-Step is the smallest interval of time for describing an infinitely 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 $r_i = \frac{\mathrm{HS}}{H(o_i)}$. 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 sth occurrence of o_i. The instant for which o_i^s is computed is denoted as $t(o_i, s) = H(o_i) \times s$. 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 $o_i, o_j \in V_{\mathrm{I}}$ connected by an arc $(o_i, o_j) \in A_{\mathrm{I}}$, then adding an arc (o_i^s, o_j^u) to A_{M}, depends on the instants $t(o_i, s)$ and $t(o_j, u)$ for which o_i^s and o_j^u are computed respectively. In other words, if $t(o_i, s)$, and $t(o_j, u)$ are the simulation steps associated with o_i^s and o_j^u respectively, then the inequality $t(o_i, s) \leq t(o_j, u)$ 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. Formally, o_j^u is connected with o_i^s such that $s = \max\ (0, 1,\ldots, r_{\mathrm{i}-1})$: $t(o_i, s) \leq t(o_j, u)$. 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 $(o_i, o_j) \in A_{\mathrm{I}}$ 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 $(o_i, o_j) \in A_{\mathrm{I}}$, 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:

$$u = \left\lceil s \times \frac{H(o_i)}{H(o_j)} \right\rceil. \quad (7)$$

We recall that for a slow to fast dependence, the master algorithm can perform extrapolation of the inputs of the receiving FMU. For a fast to slow dependence $(o_i, o_j) \in A_{\mathrm{I}}$, 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:

$$s = \left\lfloor u \times \frac{H(o_j)}{H(o_i)} \right\rfloor. \quad (8)$$

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 sth 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 6 shows the graph obtained by applying the multi-rate transformation algorithm on the graph of Figure 4. In this example $H_{\mathrm{B}} = 2 \times H_{\mathrm{A}}$, where H_{A} and H_{B} are the communication steps of FMUs A and B respectively.

Without any loss of generality, the superscript which denotes the number of the occurrence of an operation is not used in the remainder of this article for the sake of simplicity. Each occurrence of an operation o_i^s in the graph

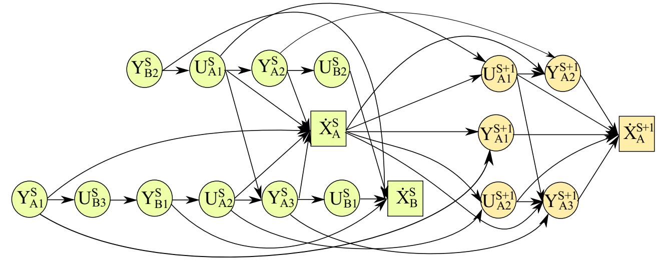


Fig. 6. Graph obtained by applying the multi-rate transformation algorithm on the graph of Figure 4.

$G(V, A)$ becomes an operation that is referred to using the notation o_j.

Algorithm 1: Multi-rate graph transformation algorithm.

Input : Initial operation operation graph $G(V, A)$;
Output: Transformed operation operation graph $G(V, A)$;

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

3.2 Dependence graph with mutual exclusion constraints

3.2.1 Motivation

The FMI standard states that *"FMI functions of one instance don't need to be thread safe"*. Consequently, an FMU could be implemented using global variables which introduce errors when calling the different functions of the FMU in parallel. 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 [12] we have shown that using synchronization objects such as mutexes or allocation constraints strongly reduce the obtained speedup. We consequently propose 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. The proposed method is based on modeling the mutual exclusion constraints in the operation graph of the co-simulation.

3.2.2 Acyclic orientation of mixed graphs

The operation graph model can be extended in order to represent scheduling problems that involve precedence constraints and also mutual exclusion constraints. This is commonly done using *mixed graphs*. A mixed graph $G(V, A, D)$ is a graph which contains a set A of directed arcs denoted (o_i, o_j): $0 \leq i, j < n$ and a set D of undirected edges denoted $[o_i, o_j]$: $0 \leq i, j < n$. In the scheduling literature, these graphs are known also as disjunctive graphs [16]. In addition to the arcs corresponding to the previously introduced precedence constraints, 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 undirected edge which is interpreted by assigning a direction to this edge. Cycles must not be introduced in the graph while assigning directions to edges, otherwise, the scheduling problem becomes infeasible. Since the final goal is to accelerate the execution of the co-simulation, 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 of a graph. 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 9 holds:

$$\forall [o_i, o_j] \in D, \alpha(o_i) \neq \alpha(o_j). \quad (9)$$

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 [17–20] links the length of the longest path of the graph, obtained by the orientation which minimizes this length, to vertex coloring of the graph. It states that the length of the longest path of a directed graph is at least $\chi(G)$. Thus, a minimum vertex coloring leads to an acyclic orientation that minimizes the length of the critical path of the resulting graph. Computing the chromatic number of a graph is NP-complete [21].

The acyclic orientation of a mixed graph can be obtained via vertex coloring also. However, vertex coloring of a mixed graph has to take into account both arcs and edges of the graph. More precisely, a vertex coloring of a mixed graph is a function $\alpha: V \rightarrow \{1, 2, \ldots, k\}$ such that inequalities 9 and 10 hold:

$$\forall (o_i, o_j) \in A, \alpha(o_i) < \alpha(o_j). \quad (10)$$

A coloring of a mixed graph $G(V, A, D)$ exists only if it is cycle-free [22], *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 [23–25]. The authors proposed efficient algorithms for the orientation of special types of mixed graphs and showed that, in the general case, the problem is NP-Hard.

3.2.3 Problem formulation

Let $G(V, A)$ be an operation graph of an FMU co-simulation constructed as described in previous sections. In order to represent mutual exclusion constraints between FMU operations, the initial operation graph $G(V, A)$ is transformed into a mixed graph by connecting each pair of mutually exclusive operations o_i, o_j by and edge $[o_i, o_j]$. The resulting mixed graph is denoted $G(V, A, D)$, where V is the set of operations, A is the set of arcs, and D is the set of edges. Once the mixed graph is constructed, directions have to be assigned to its edges in order to define an order of execution for mutually exclusive operations. The precedence and mutual exclusion relations represented by the mixed graph $G(V, A, D)$ are given by expressions (11) and (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:

$$\forall (o_i, o_j) \in A, E(o_i) \leq S(o_j), \quad (11)$$

$$\forall [o_i, o_j] \in D, (S(o_i), E(o_i)] \cap (S(o_j), E(o_j)] = \emptyset. \quad (12)$$

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 $(S(o_j), E(o_j)]$ in the graph $G(V, A)$ overlap (Eq. (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:

$$E(o_i) > S(o_j) \text{and} E(o_j) > S(o_i). \quad (13)$$

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 in 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 problem 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:**
  - $\forall (o_i, o_j) \in A, \alpha(o_i) < \alpha(o_j)$
  - $\forall [o_i, o_j] \in D, \alpha(o_i) \neq \alpha(o_j)$

3.2.4 Resolution using integer linear programming

Let $G(V, A, D)$ be a mixed graph constructed from 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.

Tables 1 and 2 summarize the variables and the constants that are used in the ILP formulation respectively.

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 (14) captures this constraint. Note that expression (14) indicates that the start time of operation o_j is greater or equal to the end time of each predecessor o_i. This is sufficient to express

Table 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 graph critical path

Table 2. Constants used in the ILP formulation of acyclic orientation problem.

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

$S(o_j) = \max_{o_i \in pred(o_j)}(E(o_i))$ since the formulated problem is a minimization problem.

$$\forall (o_i, o_j) \in A, S(o_j) \geq E(o_i) : \quad (14)$$

– *Mutual exclusion constraints:* We define the binary variable b_{ij} which is associated with the direction that is assigned to the edge $[o_i, o_j]$. The assignment of directions to edges is given by the function $\phi : \{[o_i, o_j] \in D\} \rightarrow \{(o_i, o_j), (o_j, o_i)\}$. 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.* color $\phi([o_i, o_j]) = (o_i, o_j)$ and to 0 otherwise. Note that b_{ij} is the complement of b_{ji}. For every pair of operations that are connected by and edge, we have to ensure that their time intervals are strictly disjoint, *i.e.* $\forall [o_i, o_j] \in D$, $(S(o_i), E(o_i)] \cap (S(o_j), E(o_j)] = \emptyset$. Expressions (15) and (16) capture this constraint where M is a large positive integer.

$$\forall [o_i, o_j] \in D, \;\; S(o_i) \geq E(o_j) - M \times (1 - b_{ij}) \quad (15)$$

$$\forall [o_i, o_j] \in D, \;\; S(o_j) \geq E(o_i) - M \times b_{ij} \quad (16)$$

– *Time intervals:* Expression (17) is used to compute the end time of each operation.

$$\forall o_i \in V, \;\; E(o_i) = S(o_i) + C(o_i) \quad (17)$$

– *Length of the critical path:* The critical path P is equal to the maximum of the end times of all the operations (expression (18)).

$$\forall o_i \in V, \;\; P \geq E(o_i) \quad (18)$$

The objective of this linear program is to minimize the length of the critical path of the operation graph (expression (19)):

$$\min(P). \quad (19)$$

3.2.5 Acyclic orientation Heuristic

While exact algorithms such as ILP give optimal results, they suffer from very long execution times not acceptable for the users. For many real world applications, ILP fails to produce the results within acceptable times. Heuristics are usually good alternatives. While the optimality of the solution cannot be guaranteed when using heuristics, they, in most cases, provide results of good quality, not too far from the optimal solution within acceptable execution times. We propose in this section a heuristic for the acyclic orientation of the mixed graph $G(V, A, D)$. A straightforward acyclic orientation can be obtained by sorting the operations in a non-decreasing order of their start times $S(o_i)$ and assigning directions to edges following this order, *i.e.* $\forall [o_i, o_j] \in D, \;\; S(o_i) \leq S(o_j), \;\; \phi([o_i, o_j]) = (o_i, o_j)$. This is a fast greedy acyclic orientation, however it can be improved as we 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:

$$V = \bigcup_{k=1}^{d} V_k, \forall o_i^p, o_j^q \in V_k, k \in \{0, 1, \ldots, d\}, \quad f_m(o_i^p) = f_m(o_j^q) \text{and}\, p = q. \quad (20)$$

It is known 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:

$$D = \bigcup_{k=1}^{d} D_k, \forall [o_i^p, o_j^q] \in D_k, k \in \{0, 1, \ldots, d\}, \quad f_m(o_i^p) = f_m(o_j^q) \text{and}\, p = q. \quad (21)$$

In view of the above, we define the set of subgraphs which constitute the graph $G(V, \emptyset, D) = \bigcup_{k=1}^{d} G(V_k, D_k)$. Theorem 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 $G(V, A, D)$.

Theorem 3.1. *An acyclic orientation of the mixed graph $G(V, A, D)$ can be obtained by finding an acyclic orientation for every subgraph G_k (V_k, D_k) following the*

non-decreasing order of the start times of the operations as described previously.

Proof. In order to prove this, we have to show that every edge in D is assigned a direction and that the resulting orientation does not lead to the creation of a cycle. We use a proof by contradiction to prove this statement. Since every edge $[o_i, o_j]$ belongs to one subset of edges D_k, finding acyclic orientations for all the subgraphs G_k (V_k) leads to assigning a direction to every edge in D. The existence of a cycle in the resulting graph means that there exists at least an edge $[o_i, o_j]$ 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 which contradicts the previous assertion and thus proves Theorem 3.1.

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

- Coloring α_l of operation o_i does not lead to assigning a direction to any conflict edge. In this case, no changes of the timing attributes occur.
- Coloring α_l of operation o_i leads to assigning a direction to at least one conflict edge $[o_i, o_j] \in D_k$. Without any loss of generality, suppose that the edge $[o_i, o_j]$ is transformed into the arc (o_i, o_j), 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 $o_{j'} \in \text{desc}(o_j)$ (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 $o_{i'} \in \text{ance}(o_i)$ (see Algorithm 3).

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 $[o_i, o_j] \in D$. Applying Theorem 3.1, the heuristic consists in finding vertex colorings of the subgraphs which constitute the graph $G(V, A, D)$ (see Algorithms 4 and 5). 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

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} ⊂ V;
Set α_l the last assignment of color made to an
  operation o_i ∈ 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) then
        S(o_h) ← E(o_t);
        E(o_h) ← S(o_h) + C(o_h);
        update(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 ;
```

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 already colored operations L_k for each subgraph $G(V_k, \emptyset, D_k)$. The operations of every list $o_i \in L_k$ are sorted in increasing order of their assigned colors. At each iteration, the heuristic selects among the operations not yet colored $o_i \in V$, the operation which has the earliest start time $S(o_i)$ to be assigned a color. Ties are broken by selecting the operation with the least flexibility. We call the operation to be colored at a given iteration, the *pending operation*. The heuristic checks in the order of L_k if the edges which connect the pending operation $o_i \in V_k$ with the operations $o_j \in L_k$ are conflict edges. If a conflict edge $[o_i, o_j] \in D_k : o_j \in L_k$ is detected, the pending operation is assigned the color $\alpha(o_j)$ and the colors assigned to all the already colored operations $o_{i'} \in L_k$ 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 using Algorithms 2 and 3. At this point, the increase of 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 $\alpha(o_i) = \alpha(o_i) + 1$. The increase in the critical path is evaluated again similarly. After repeating this process for all the edges $[o_i, o_j] \in D_k : o_j \in L_k$, 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 colors of all its predecessors which guarantees that no cycle is generated. The heuristic finally stops when all the operations have been assigned colors.

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

Input : Input Attributes of the mixed graph $G(V, A, D)$, partially colored subgraph $G_k(V_k, A_k, D_k)$;
Output: Output Update of the start and end times from end of a subset of operations $\{o_i\} \subset V$;
Set α_l the last assignment of color made to an operation $o_i \in V_k$;
Set $A_{k,l} = \{(o_t, o_h)\}$ the set of all arcs created from the orientations engendered by α_l;
foreach $(o_t, o_h) \in A_{k,l}$ **do**
  if $S(o_h) < E(o_t)$ *and* $S(o_t) < E(o_h)$ **then**
    if $\overline{E}(o_t) < \overline{S}(o_h)$ **then**
      $\overline{E}(o_t) \leftarrow \overline{S}(o_h)$;
      $\overline{S}(o_t) \leftarrow \overline{E}(o_t) + C(o_t)$;
      update(o_t);
    end
  end
end
Procedure update(o_t)
  if $pred(o_t) \neq \emptyset$ **then**
    foreach $o_{t'} \in pred(o_t)$ **do**
      if $\overline{E}(o_t^*) < \overline{S}(o_t)$ **then**
        $\overline{E}(o_{t'}) \leftarrow \overline{S}(o_t)$;
        $\overline{S}(o_{t'}) \leftarrow \overline{E}(o_{t'}) + C(o_{t'})$;
        update($o_{t'}$);
      end
    end
  end
  return ;

3.2.6 Complexity

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

4 Multi-core scheduling of FMU dependence graphs for co-simulation acceleration

This section 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 sections, it is scheduled on the multi-core platform with the goal of accelerating the execution of co-simulation.

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 hereafter 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.

Algorithm 4: Acyclic orientation heuristic.

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

Continued on next page

Continued

```
            if S(o_i) ≥ E(o_j) then
                α(o_i) ← α(o_j) + 1;
                evaluate(o_i, L_k);
            end
            else
                α(o_i) ← α(o_j);
                evaluate(o_i, L_k);
            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 : φ([o_i, o_i']) ←
                      (o_i, o_i');
                end
                else
                    Assign a direction to the edge
                      [o_i, o_i'] ∈ D_k : φ([o_i, o_i']) ←
                      (o_i', o_i);
                end
            end
            Update the timing attributes using
              Algorithms 2 and 3;
            Remove o_i from Ω; L_k ← L_k ∪ {o_i};
        end
    end
end
```

Algorithm 5: Evaluate procedure.

```
Procedure evaluate(o_i, L_k)
    foreach o_i' ∈ L_k : 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
    return ;
```

4.1 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)

4.2 Resolution using linear programming

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

4.2.1 Variables and constants

Tables 3 and 4 summarize respectively the variables and the constants that are used in the ILP formulation of the scheduling problem for co-simulation acceleration.

4.2.2 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 (22) gives the constraint that each operation has to be allocated to one and only one core:

$$\forall o_i \in V, \sum_{p_j \in P} x_{ij} = 1. \tag{22}$$

The end time of each operation o_i is computed using the expression (23):

$$\forall o_i \in V, E(o_i) = S(o_i) + C(o_i). \tag{23}$$

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 (24) and (25) capture this constraint. b_{ij} is a binary variable that is set to one if o_i is executed before o_j:

$$\begin{aligned} &\forall p \in P, \forall o_i \in V, \forall o_j \in V, (o_i, o_j), (o_j, o_i) \notin A_F, \\ &E(o_i) \le S(o_j) + M \times (3 - x_{ip} - x_{jp} - b_{ij}), \end{aligned} \tag{24}$$

$$\begin{aligned} &\forall p \in P, \forall o_i \in V, \forall o_j \in V, (o_i, o_j), (o_j, o_i) \notin A_F, \\ &E(o_j) \le S(o_i) + M \times (2 - x_{ip} - x_{jp} + b_{ij}). \end{aligned} \tag{25}$$

Table 3. Variables used in the ILP formulation of the acyclic orientation problem.

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

Table 4. Constants used in the ILP formulation of acyclic orientation problem.

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

The cost of synchronization is taken into account as follows. A synchronization cost is introduced in the computation of the start time of an operation o_j, if it has a predecessor 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, $\text{sync}_{ijp} = 1$ iff $\alpha(o_j) = p$ and $\alpha(o_i) \neq p$ and $S(o_j) = \max_{o_{j'} \in \text{succ}(o_i) \text{ and } \alpha(o_{j'})=p}(S(o_{j'}))$. Expressions (26) and (27) 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 (28) and (29) 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 (30) and (31):

$$\forall o_i \in V, \sum_{\forall p \in P, \forall o_j \in \text{pred}(o_i)} \text{sync}_{ijp} = V_{ip}, \tag{26}$$

$$\forall o_i \in V, \forall o_j \in \text{succ}(o_i), \text{sync}_{ijp} \leq x_{jp} : \forall o_i \in V, \tag{27}$$

$$\forall o_i \in V, \forall o_j \in \text{succ}(o_i), V_{ip} \geq x_{jp} - x_{ip} : \forall o_i \in V, \tag{28}$$

$$\forall o_i \in V, V_{ip} \leq \sum_{\forall o_j \in \text{succ}(o_i)} \left(x_{jp} - x_{ip}\right), \tag{29}$$

$$\forall o_i \in V, \forall o_j \in \text{succ}(o_i), Q_{ip} \leq S(o_j) + M \times \left(1 - x_{jp}\right), \tag{30}$$

$$\forall o_i \in V, \forall o_j \in \text{succ}(o_i), Q_{ip} \geq S(o_j) - M \times \left(1 - \text{sync}_{ijp}\right). \tag{31}$$

The start time of each operation is computed using expression (32). The synchronization cost is introduced taking into account the synchronizations with all predecessors of o_j that are allocated to different cores:

$$\forall o_j \in V, \forall o_i \in \text{pred}(o_j),$$
$$S(o_j) \geq \left[E(o_i) + \sum_{\forall p \in P, \forall o_{i'} \in \text{pred}(o_j)} \text{sync}_{ijp} \times \text{synCost}\right]. \tag{32}$$

The makespan is equal to the latest end time among the end times of all the operations as captured by expession (33):

$$\forall o_i \in V, \quad P \geq E(o_i). \tag{33}$$

4.2.3 Objective function

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

$$\min(P). \tag{34}$$

4.3 Multi-core scheduling Heuristic

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 minimize its makespan. Because of the number of fine-grained operations, and since the execution times and the dependence between the operations are known before runtime, it is more convenient to use an offline scheduling heuristic which has the advantage of introducing lower overhead than online scheduling heuristics. We use an offline scheduling heuristic similar to the one proposed in [26] 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 is detailed in Algorithm 6 and 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.

4.3.1 Complexity

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

Algorithm 6: Multi-core scheduling heuristic.

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

4.4 Code generation

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

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

synchronizes with the other threads. Hereafter, we refer to these threads as *schedule threads*.

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

5 Evaluation

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

5.1 Random generator of operation graphs

Due to the difficulty in acquiring enough industrial FMU co-simulation applications for assessing our approach, we had to use a random generator of FMU dependence graphs. The generator creates the graphs and characterizes them with attributes. We set the parameters of this generator, *e.g.* the number of FMUs and the number of operations based on our experience using industrial FMU co-simulation. Of course, the use of randomness for synthetic graphs generation restraints the possibility of considering exactly the same results for industrial application. Our evaluation nevertheless gives some indications about the achievable level of performance for the heuristics and the graph scale from which the exact algorithms may fail to produce result with an acceptable execution duration.

5.1.1 Random operation graph generation

The random generator that we have implemented consists in two stages. In other words, 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. The proposed generator is based on a technique of assignment of operations to levels. The level of an operation is the number of operations on the longest path from a source operation to this operation. The dependence graph can then be visualized on a grid of levels as depicted in Figure 7. The generator uses the following parameters:

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

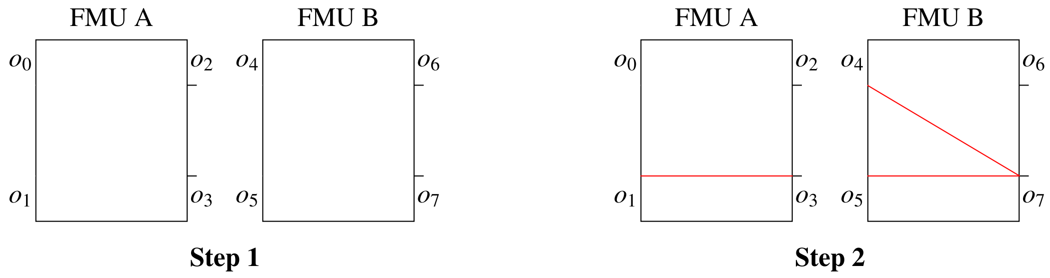
Note that parameters n and m are related. In other words, for a given size of a graph n, an adequate number of FMUs m has to be chosen.

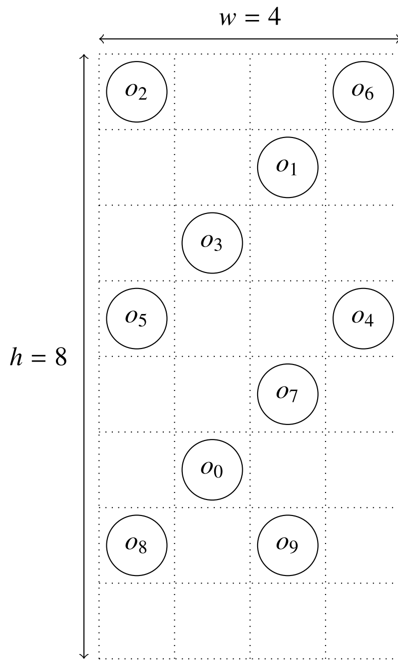
The generation of the dependence graph is performed as follows:

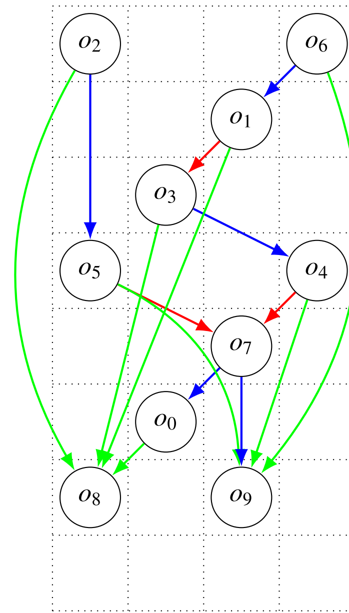
- **Input:** Size of the graph n, number of FMUs m, height of the graph h, and width of the graph w.
- **Step 1:** Randomly distribute the n operations across the m FMUs. Given the number of operations of each FMU, we randomly determine the number of its input operations and the number of its output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

1. Assign all NDF operations to level 0 of the grid.
2. Randomly assign remaining output operations to even levels (2, 4,…, $h-3$) of the grid.
3. Assign the input operations to the odd levels (1, 3,…, $h-4$) 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. Note that non oriented edges are







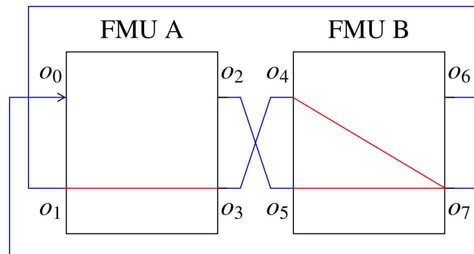


Fig. 7. Random generation of an operation graph.

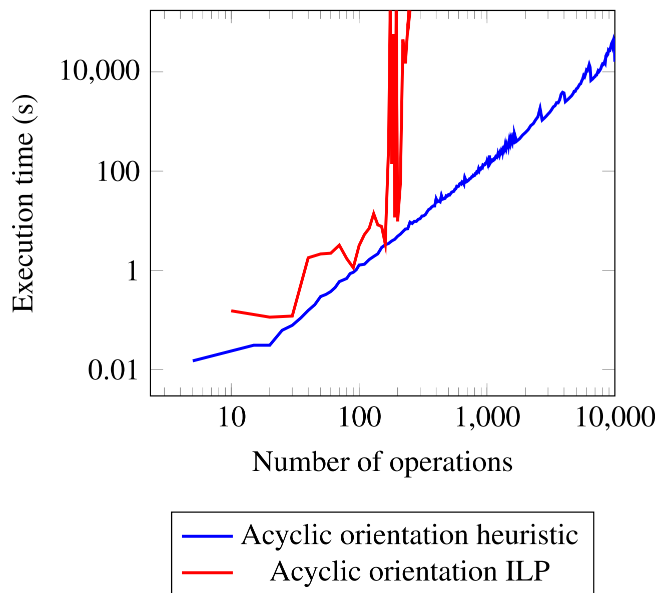


Fig. 8. Comparison of the execution times of the acyclic orientation algorithms.

automatically added between every pair of operations that belong to the same FMU and are mutually exclusive as described in Section 3.1.

Figure 7 illustrates the steps of our proposed random operation graph generator.

5.1.2 Random operation graph characterization

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

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

5.2 Performances of the Heuristics

We have carried out different tests in order to evaluate our proposed approach. For both the acyclic orientation and the scheduling, we compared the execution time of our proposed heuristic with the execution time of the ILP, and the value of the objective function of the heuristic with the value of the objective function of the ILP. For ILP resolution, we used three solvers: lpsolve [27], Gurobi [28], and CPLEX [29]. With lpsolve, we were only able to solve small instances of the scheduling problem. Gurobi was much more efficient but we obtained the best performance using CPLEX. Therefore, the results presented hereafter were obtained using CPLEX. Tests were performed on a desktop computer with a 6-core Intel Xeon processor running at 2.7 GH and 16GB RAM.

5.2.1 Execution time of the acyclic orientation algorithms

In order to compare the execution time of our acyclic orientation heuristic with the execution time of the acyclic orientation ILP, we have generated 200 random operation graphs of different sizes between 5 and 10 000. We considered 10 000 as the maximum size of the operation graph because it corresponds to the size of large industrial applications.

We executed the acyclic orientation heuristic and ILP on all of the generated random graphs and measured the elapsed time between the start and the end of the execution. For the ILP, the execution is stopped if the optimal solution is not found within two days. The obtained execution times are shown on a logarithmic scale in Figure 8. The acyclic orientation ILP cannot be resolved in practical times when the size of the operation graph exceeds 250. When the number of operations is less than 250 the ILP finds the optimal solution in reasonable times, except for two graphs. In addition, we observe that an increase in the graph size does not always result in an increase in the execution time. This can be explained by the fact that other factors impact the speed of resolution, *e.g.* number of conflict edges. Still, it is important to notice that the application of the acyclic orientation ILP is limited to relatively small graphs. On the other hand, the acyclic orientation heuristic produces results in practical execution times even for very large operation graphs (10 000).

5.2.2 Critical path length

We compared the values of the critical path length obtained using the acyclic orientation heuristic and ILP. Tests were performed using the same set of operation graphs described in the previous section. However, we consider only graphs for which the ILP was able to return the optimal solution within the resolution time limit that we set, *i.e.* two days. Thus, we applied our proposed heuristic and ILP on 12 operation graphs of sizes between 20 and 240 and saved the obtained length of the critical path. Results are depicted in Figure 9. For most of the operation graphs, our acyclic orientation heuristic produces a length of the critical path that is equal to the length of the critical path produced by the acyclic orientation ILP. The heuristic returns a longer length of the critical path for three graphs but the gap is very small remaining below 8%.

5.2.3 Execution time of the scheduling algorithms for co-simulation acceleration

Similarly to the acyclic orientation tests, we compared the execution time of the scheduling heuristic with the execution time of the scheduling ILP using 200 generated random operation graphs of different sizes between 5 and 10 000. We set a two day limit for the resolution of the ILP. Tests

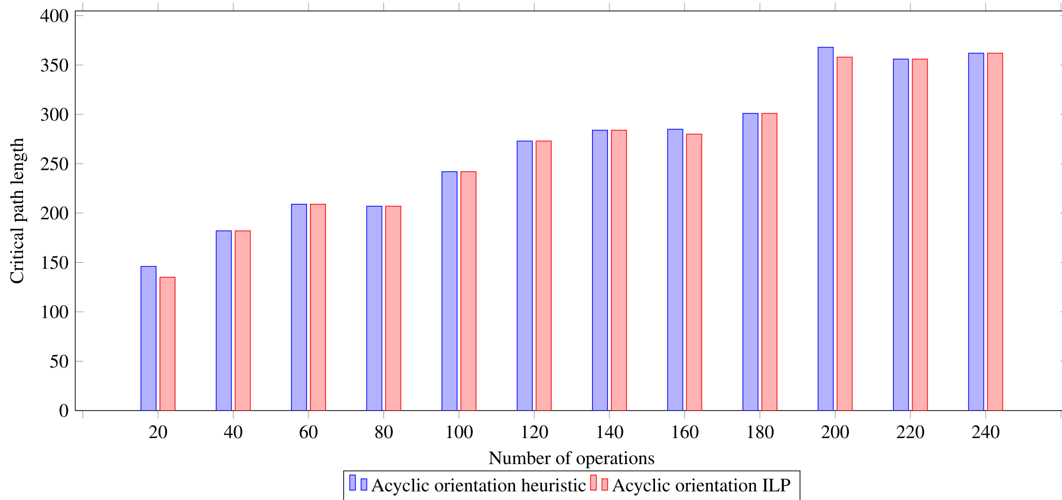


Fig. 9. Comparison of the critical path length.

were run for the scheduling problem with 2, 4, and 8 cores. Execution times were measured by fixing the number of cores and varying the number of operations (graph size). The results are depicted for two and eight cores in Figures 10 and 11 respectively. All results are plotted on a logarithmic scale. In these figures, we see that the execution time of the ILP resolution increases exponentially as the graph size increases, and only small instances are resolved within acceptable times. On the other hand, the acyclic orientation heuristic is very fast and produces results in very short times and even for very large graphs, the execution times remain within practical bounds.

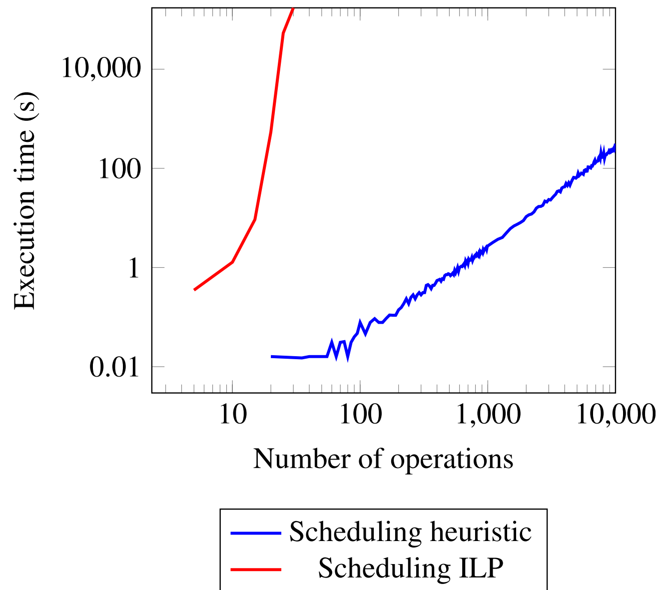


Fig. 10. Comparison of the scheduling execution time for two cores.

5.2.4 Makespan

We run tests to compare the value of the makespan obtained using the acyclic orientation heuristic and ILP. For these tests we have generated ten operation graphs of size $n = 15$. We have used graphs of size 15 because the ILP resolution returns the optimal solution in very short times which is not the case for large graphs. The graphs are different from each other because they are generated randomly which leads to different topologies and execution times of the operations. We run the scheduling heuristic and ILP on these graphs to obtain the values of the makespan. Results are shown in Figures 12 and 13 for two and eight cores respectively. Overall, the results show that the scheduling heuristic produces a makespan which is very close to the makespan produced by the scheduling ILP. The gap between the heuristic and the ILP result lies between 0% and 16%. We notice that the gap is smaller when eight cores are used than when two cores are used. In fact, when two cores are used the maximum gap is 16%, whereas when four or eight cores are used the maximum gap is 6%. This shows that the scheduling heuristic performs better when the effective parallelism is increased. It can be explained by the fact that the scheduling heuristic attempts more allocation possibilities which leads to a better exploitation of the potential parallelism.

5.3 Industrial use case

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

5.3.1 Use case description

Our use case consists in a Spark Ignition (SI) RENAULT F4RT engine co-simulation. It is a four-cylinder in line Port Fuel Injector (PFI) engine in which the engine displacement

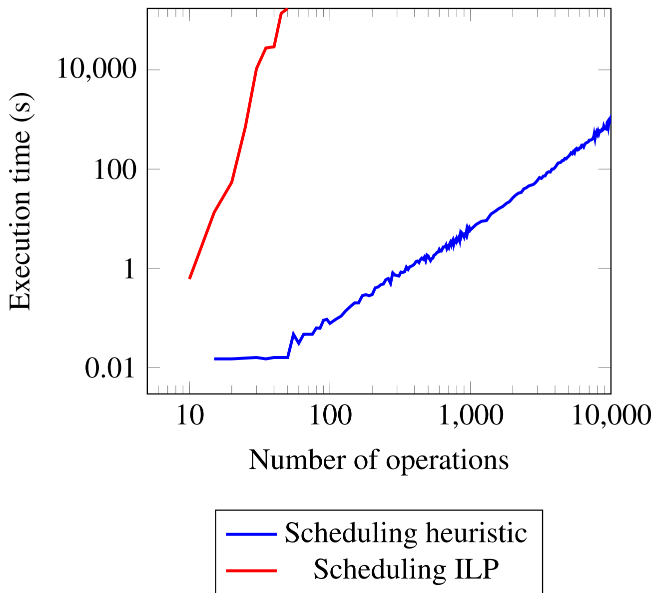


Fig. 11. Comparison of the scheduling execution time for eight cores.

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 (Fig. 14). This co-simulation is composed of six FMUs: an FMU of the airpath, four FMUs of the four cylinders, and one FMU of the controller. The engine model was developed using Modelica. The engine model was imported into xMOD using the FMI export features of the Dymola[1] tool. The operation graph of this use-case contains over 100 operations.

5.3.2 Test campaign

We based our tests on three different versions of RCOSIM. 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 µs. 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 µs. The communication step used for the airpath is 100 µs. 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.

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

5.3.3 Speedup

The speedup obtained using MUO-RCOSIM is compared with the speedups obtained using RCOSIM and MU-RCOSIM. The speedup was evaluated by running the co-simulation in xMOD. Execution times measurements were performed by getting the system time stamp at the beginning and at the end of the co-simulation. For a given run of the co-simulation, the speedup is computed by dividing the single-core co-simulation execution time of RCOSIM by the co-simulation execution time of this run on a fixed number of cores. Figure 15 sums up the results. The same speedup is obtained using MUO-RCOSIM and MU-RCOSIM even when only one 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 µs to 100 µs 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 two cores are used, and approximately 10% when four 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 eight 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 eight cores is less than the speedup with four cores for all the approaches. The best performance is obtained using five cores with slight improvement compared to using four cores.

5.4 Comparison of offline and online scheduling

In our work, we adopted an offline scheduling heuristic assuming it is more efficient than online scheduling since it introduces lower overhead. This choice was based on the fact that the grain size of the operation graph is small which makes it unsuitable for online scheduling which involves more decision overhead in runtime than offline scheduling. In fact, the decision overhead in runtime may become much more costly then the execution of the operations. Moreover, the different operations perform different tasks and have different execution times in contrast to applications that exhibit data parallelism which could be efficiently handled by online scheduling. In addition, the execution times of the operations and the dependence between them are known before the execution which allows the application of offline scheduling. In order to confirm this

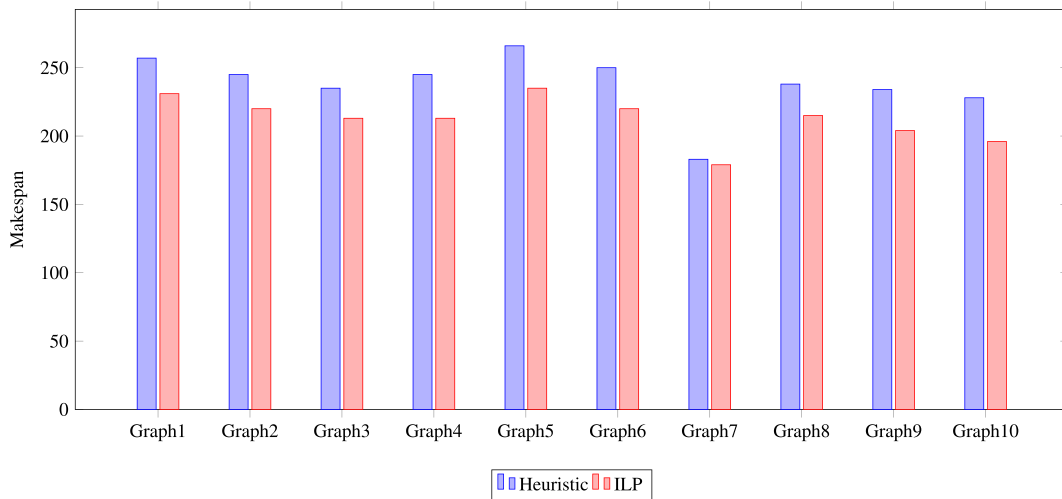


Fig. 12. Comparison of the makespan for two cores.

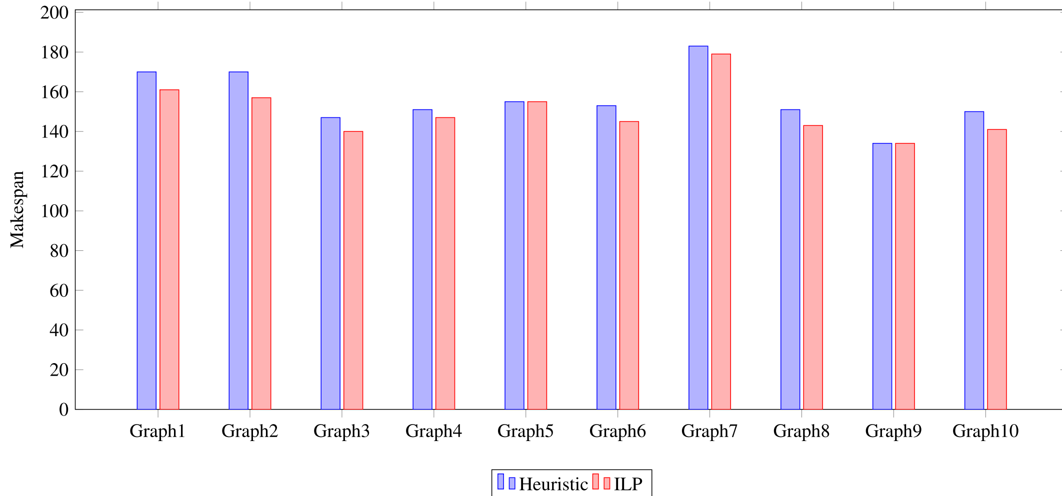


Fig. 13. Comparison of the makespan for eight cores.

assumption, we have compared our approach with a run-time scheduling approach, *i.e.* online scheduling. For this end, we have used Intel TBB library for the parallelization of the co-simulation. We performed several speedup tests and compared the results obtained using the two approaches.

5.4.1 Intel TBB flow graph

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

The user creates a graph, its nodes and then specifies dependence between them. The classes and functions of Intel TBB are highly parametrized to allow many possibilities of implementation.

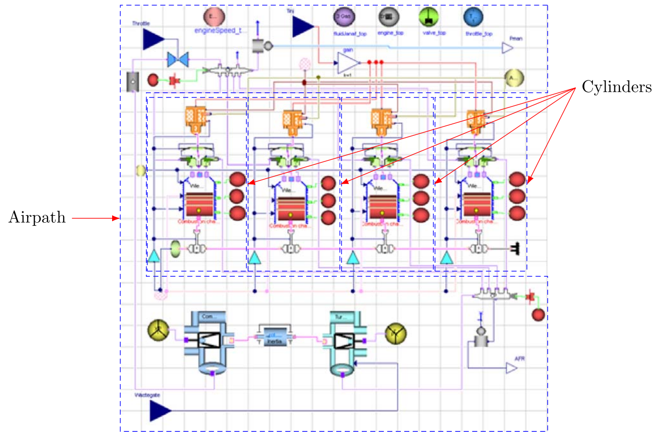


Fig. 14. Spark Ignition (SI) RENAULT F4RT engine model.

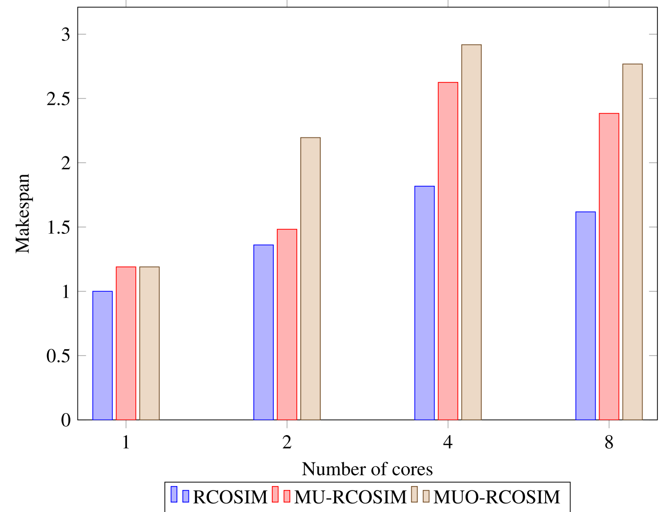


Fig. 15. Speedup results.

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

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

5.4.2 Scheduling in Intel TBB

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

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

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

5.4.3 Implementation

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

The creation of the dependence graph is done as follows: First, a graph is created and then nodes and edges are added to this graph. A node is created for each operation and added to an array that stores all the created nodes. The first node that is created is a source node which has no predecessor. This node becomes a predecessor of all the nodes that have no predecessor in the operation graph. The body of this node contains the initialization of the co-simulation. Then, for each operation in the operation graph, a function node is created. A function node can have multiple ports to be connected with multiple predecessors and successors. The body of each function node contains the FMU function calls of the corresponding operation. Finally, the edges that connect the nodes are added to the graph. All the created edges are of type continue message. Such edges are used to signal that the execution finished.

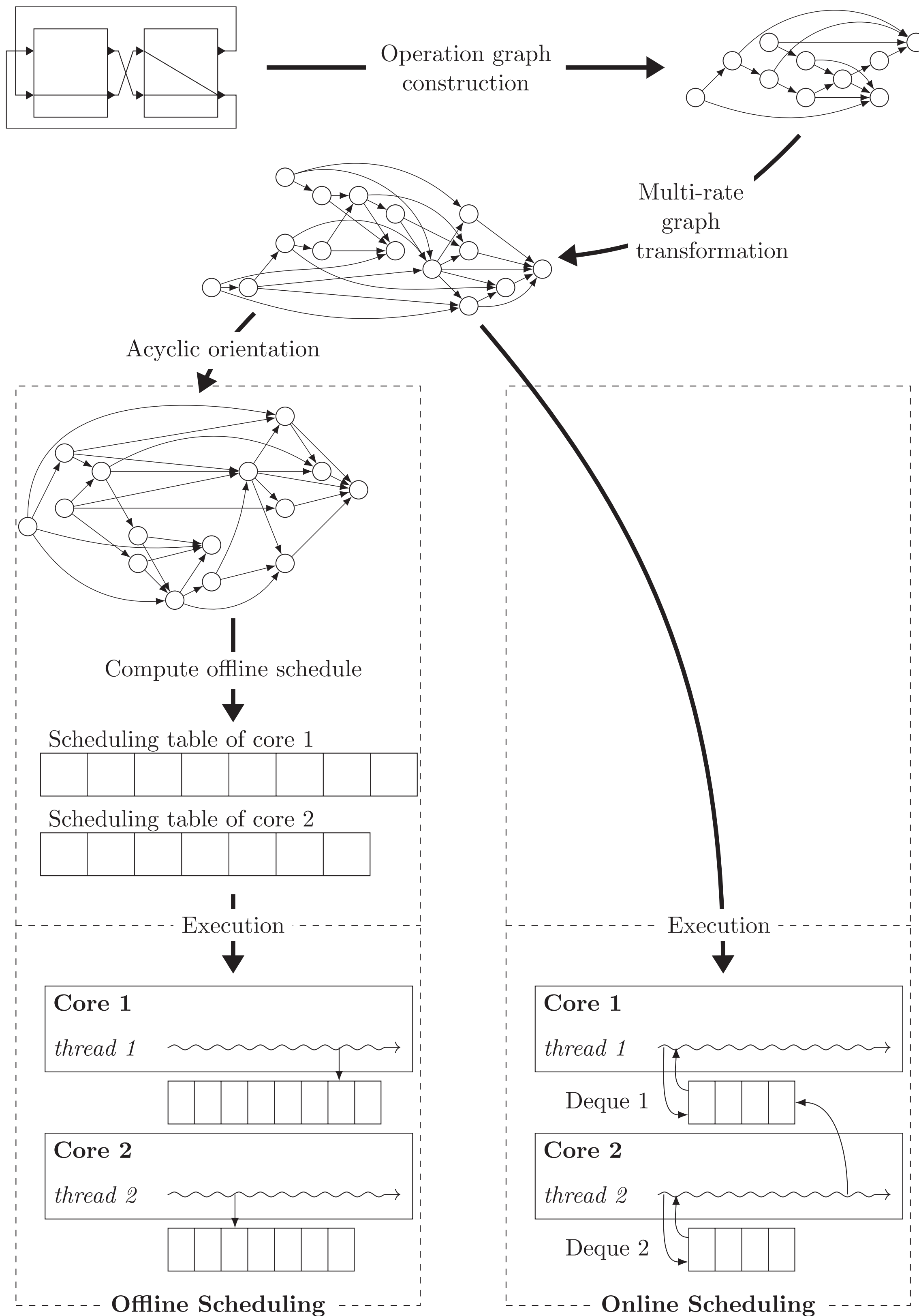


Fig. 16. Comparison of the different phases of the offline and online scheduling approaches.

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

5.4.4 Comparison

We implemented a parallelization approach of FMI co-simulations using Intel TBB for the purpose of comparing it with our proposed offline scheduling approach. We have measured the speedups obtained on different numbers of cores using both approaches. First of all, let's summarize the differences between the two approaches. Figure 16 illustrates the main steps of both approaches. As stated above, the two first two phases which consist in the construction of the operation graph and the graph transformation in the case of a multi-rate co-simulation are performed in the same way in both approaches. If online scheduling is used, the next step is execution. On the other hand, if offline scheduling is used, two more phases are performed before the execution. The acyclic orientation heuristic is applied on the operation graph to handle mutual exclusion constraints. After this, the offline scheduling heuristic is used to compute a schedule of the operations. During execution, in both the offline and online scheduling approaches, a thread is executed on each core. In the case of offline scheduling, each thread reads the schedule of the corresponding core and executes the operations in the order of this schedule, which does not change during execution. In the case of online scheduling, since no schedule is computed before execution, the runtime library distributes the operations across the threads during execution in such a way to balance the load. Each thread pushes the operations onto its deque from the top. It executes these operations by popping the operation on the top from its deque, or if its deque is empty, it steals work from another thread by popping an operation for the bottom of this victim thread. Mutual exclusion constraints are handled in online scheduling using lightweight mutex locks provided by the runtime library. These locks have lower cost than mutex locks provided by the OS.

We run the co-simulation of the industrial use case on an 8-core Intel core i7 processor running at 2.7 GH with 16GB RAM. The obtained speedup using offline scheduling raises 2.44 and is better than the one obtained using online scheduling (1.64) which confirms our assumption. The decision overhead of online scheduling is very costly compared to the execution times of operations which decreases the performance.

6 Discussion

In order to discuss the advantages and drawbacks of the proposed approach, we challenge here our methodology in terms of compliance with co-simulation scenario, application to industrial use case and optimality metrics. First, in Section 2 we explain that RCOSIM and our proposed extension is applied to co-simulation of connected models, where each model is imported as a FMU with its own fixed step solver. Since we aim at proposing solution to accelerate industrial co-simulation, we choose to focus on co-simulation solution where all models are imported into a unique tool. Indeed, optimizing a multi-software co-simulation, from different tool vendors, would not be possible without accessing the code of each simulation tool. On the opposite, the FMI standard meets a real success and most modeling and simulation tools offer now the possibility to export models as FMUs. By focusing on what could be done in the importing software, we keep a large set of possibilities as a real capability to experiment. Our industrial co-simulation goal also prevents us to propose solution including models modification. In large companies or when different parties provide different components of a system, numerous models are available. Each of them was built, validated and used by domain experts. At the co-simulation stage of the development process, the engineers are rarely able to understand all the models, and sometimes they do not have access to the initial model but only to the FMU.

In our approach we restricted to FMU with fixed step solver, mainly because beyond the co-simulation acceleration problem, the next step of our work is to perform the co-simulation under real-time constraints. This goal, which is out of scope of this article, leads us to restrict to co-simulation scenarios where computation duration can be evaluated and bounded. With variable step solvers, the number of computation phases is variable and consequently the computation duration over a simulation time interval is difficult to evaluate. That prevents to ensure that real-time constraints will be met. Nevertheless, if we discard our final objective, our solution could be applied to co-simulation of FMU with variable step solver. Indeed, if for each FMU i with a variable step solver, we can set a fixed macro step value H_i, all the previous described process can be applied. The macro-step parameter for a variable step solver forces a regular rate at which computation have to be done. But the solver remains free to cut the macro-step H_i in several steps, with variable step sizes. In our approach, all these intermediate computations over the macro-step would be considered as a unique operation.

Since we describe the connected FMU with a directed acyclic graph, it could prevent to apply our methodology for co-simulation scenarios containing algebraic loops (and consequently cycles in the corresponding graph). We propose to handle algebraic loops by inserting a delay function for breaking each cycle, then solving the corresponding

initialization problem to be able to parameter the output delivered by the delay functions at the first step.

One limitation of our approach when dealing with industrial use cases could be the size of the Hyper-Step (HS), the least common multiple of the communication step sizes. Indeed, in the case of a large number of FMUs with communication steps sharing few common factor, HS could be a large number. In our solution, it directly impacts the size of the dependence graph which has to be scheduled. In order to handle this limitation, we suggest, if the communication step cannot be harmonized, to split the co-simulation senario in several ones and apply RCOSIM principles on each part. The partition choice would be made for reducing the HS of each part. Thanks to our approach, it is easy to find how many cores are sufficient to reach the maximum speed up for each sub-scenario. Then the number of cores could be partitioned, one partition for each scenario. At execution, data exchange between the different co-simulation parts would have to be sequenced by adding synchronization mechanisms.

We do not deny that our solution have a cost in terms of computation time, even if the proposed heuristics offer an interesting trade-off between optimality and computation time. If a co-simulation execution takes few seconds on a monocore, this is obviously unnecessary to lose time to compute a distributed schedule before launching it. Our approach could be easily implemented to launch the co-simulation scenario on one core while, after profiling the operation cost (from the current monocore execution), another process on another core could compute the different steps of our solution. Then, if the co-simulation is not over, the execution could switch from the initial monocore execution to the one optimized for multicore. This efficient schedule could also be recorded to be used in the next launch of the same scenario.

Our evaluation methodology could rise some observations. With automatic graph generator, we choose to test our solution on a large amount of graph rather than a few number of industrial case studies. The goal is to compare our heuristics performance to an exact solution algorithm. One may argue that the optimality comparison suffers that the exact ILP algorithm is quickly unable to find a solution when the number of operations increase and consequently we do not prove that heuristics continue to give results close to the optimal solution when graphs become bigger. However, the reader should keep in mind that our goal is to accelerate the co-simulation. Optimality is not targeted since we show that optimal solution search is too costly in terms of time. The comparison is consequently more relevant with monocore co-simulation (given by the speedup factor) or with a totally on-line scheduling algorithm. The presented industrial use-case tends to show that our proposed approach offers a real speed-up compared to these two basic solutions. Of course, numerous parameters of the co-simulation scenario could increase the computation cost of our solution. First, the maximal number of operations in one FMU is a factor, leading to increase the time spent by the acyclic orientation heuristic. Second, as previously discussed, the HS length compared to each H_i directly grows the number of operations to schedule. Finally, scheduling heuristics are sensitive to the number of cores.

7 Conclusion

The complexity of cyber-physical systems is steadily increasing due to several factors. A lot of efforts is being made in industry as well as in academia in order to implement technologies and methods that respond to the requirements and challenges in the design of complex CPS. Co-simulation is increasingly being adopted as a system-level simulation approach in the context of CPS design thanks to its advantages over monolithic simulation. Strengths of co-simulation include easy upgrade, reuse, and exchange of models and simulators, improved computational performance compared to monolithic simulation, and allowing better intervention of experts at the subsystem level in multi-domain design projects.

In this article, we are interested in the rising requirements on the computational performance of FMI co-simulation. Precisely, we look for proposing solution to efficiently exploit multi-core processor, available on every computer, to accelerate desktop co-simulation. We build on the work that was previously developed at *IFP Energies nouvelles* and aim at improving the existing RCOSIM method.

We propose extensions to the operation graph model used in RCOSIM to represent the co-simulation. The first extension targets multi-rate co-simulation. We propose some rules for transforming a multi-rate operation graph into a mono-rate one in order to prepare its multi-core scheduling. Based on these rules, we propose an algorithm that performs this transformation. The second extension consists in transforming the operation graph in order to handle mutual exclusion constraints between operations. First, the operation graph is transformed into a mixed graph by adding (non oriented) edges between mutually exclusive operations. Then, an acyclic orientation is computed for the mixed graph by assigning a direction to each edge. We propose two algorithms to perform the acyclic orientation: an ILP-based exact algorithm and a heuristic. Then we propose a multi-core scheduling algorithms for co-simulation acceleration. For this we propose two multi-core scheduling algorithms. The first algorithm is an ILP-based exact algorithm and the second one is a list scheduling heuristic. The schedule is computed using either of these algorithms over the hyperstep. During execution, this schedule is executed repeatedly. Finally, we evaluate our proposed approach. First, we propose a random generator of operation graphs. We use this graph to generate a large number of synthetic operation graphs of different sizes and structures and with different attributes. We evaluate the performances of our proposed ILP-based exact algorithms and heuristics for the acyclic orientation and scheduling for co-simulation acceleration. The obtained results show the efficiency of our heuristics. While the proposed ILP algorithms give optimal results for small operation graphs they suffer from intractable execution times. Our proposed heuristics, on the other hand, give acceptable results within acceptable execution times. Last,

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

References

1. Lee E.A., Seshia S.A. (2016) *Introduction to embedded systems: A cyber-physical systems approach*, Mit Press, Cambridge, MA.
2. Van der Auweraer H., Anthonis J., De Bruyne S., Leuridan J. (2013) Virtual engineering at work: The challenges for designing mechatronic products, *Eng. Comput.* **29**, 3, 389–408.
3. Kübler R., Schiehlen W. (2000) Modular simulation in multibody system dynamics, *Multibody Syst. Dyn.* **4**, 2–3, 107–127.
4. Blochwitz T., Otter M., Arnold M., Bausch C., Clauß C., Elmqvist H., Junghanns A., Mauss J., Monteiro M., Neidhold T., Neumerkel D., Olsson H., Peetz J.V., Wolf S. (2011) The functional mockup interface for tool independent exchange of simulation models, in: *Proc. of the 8th International Modelica Conference*, March 20–22, Dresden, Germany.
5. Gomes C., Thule C., Broman D., Larsen P.G., Vangheluwe H. (2017) *Co-simulation: State of the art.* arXiv preprint arXiv:1702.00686.
6. Sirin G., Paredis C.J.J., Yannou B., Coatanéa E., Landel E. (2015) A model identity card to support simulation model development process in a collaborative multidisciplinary design environment, *IEEE Syst. J.* **9**, 4, 1151–1162.
7. Sutter H. (2005) The free lunch is over: A fundamental turn toward concurrency in software, *Dr. Dobb's J.* **30**, 3, 202–210.
8. Gebremedhin M., Hemmati Moghadam A., Fritzson F., Stavaker K. (2012) A data-parallel algorithmic Modelica extension for efficient execution on multi-core platforms, in: *Proc. of the 9th International Modelica Conference*, September 3–5, Munich, Germany.
9. Elmqvist H., Mattsson S.E., Olsson H. (2014) Parallel model execution on many cores, in: *Proc. of the 10th International Modelica Conference*, March 10–12, Lund, Sweden.
10. Galtier V., Vialle S., Dad C., Tavella J.P., Lam-Yee-Mui J.P., Plessis G. (2015) FMI-based distributed multi-simulation with DACCOSIM, in: *Proc. of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, April 12–15, Alexandria, Virginia, pp. 804–811.
11. Ben Khaled A., Ben Gaid M., Pernet N., Simon D. (2014) Fast multi-core co-simulation of cyber-physical systems: Application to internal combustion engines, *Simulat. Pract. Theory* **47**, 79–91.
12. Saidi S.E., Pernet N., Sorel Y., Ben Khaled A. (2016) Acceleration of FMU co-simulation on multi-core architectures, in: *Proc. of the First Japanese Modelica Conference*, May 23–24, Tokyo, Japan.
13. Darte A., Robert Y., Vivien F. (2012) *Scheduling and automatic Parallelization*, Springer Science Business Media, Berlin, Germany.
14. Kohler W.H. (1975) A preliminary evaluation of the critical path method for scheduling tasks on multiprocessor systems, *IEEE T. Comput.* **100**, 12, 1235–1238.
15. Ramamritham K. (1995) Allocation and scheduling of precedence-related periodic tasks, *IEEE T. Parall. Distr.* **6**, 4, 412–420.
16. Blażewicz J., Pesch E., Sterna M. (2000) The disjunctive graph machine representation of the job shop scheduling problem, *Eur. J. Operation. Res.* **127**, 2, 317–331.
17. Gallai T. (1968) On directed paths and circuits, in: *Theory of Graphs*, Academic Press, New York, NY, pp. 115–118.
18. Roy B. (1967) Nombre chromatique et plus longs chemins d'un graphe, *Revue française d'informatique et de recherche opérationnelle* **1**, 5, 129–132, in French.
19. Vitaver L.M. (1962) Determination of minimal coloring of vertices of a graph by means of Boolean powers of the incidence matrix, *Dokl. Akad. Nauk SSSR+* **147**, 758–759.
20. Hasse M., Reichel H. (1966) Zur algebraischen Begründung der Graphentheorie. III, *Math. Nachr.* **31**, 5–6, 335–345, in German.
21. Karp R.M. (1972) Reducibility among combinatorial problems, in: *Complexity of Computer Computations*, Springer, Boston, MA, pp. 85–103.
22. Ries B. (2007) Coloring some classes of mixed graphs, *Discrete Appl. Math.* **155**, 1, 1–6.
23. Andreev G.V., Sotskov Y.N., Werner F. (2000) Branch and bound method for mixed graph coloring and scheduling, in: *Proc. of the 16th International Conference on CAD/CAM, Robotics and Factories of the Future, CARS and FOF*, June, Port of Spain, Trinidad, West Indies, pp. 1–8.
24. Sotskov Y.N., Tanaev V.S., Werner F. (2002) Scheduling problems and mixed graph colorings, *Optimization* **51**, 3, 597–624.
25. Al-Anzi F.S., Sotskov Y.N., Allahverdi A., Andreev G. (2006) Using mixed graph coloring to minimize total completion time in job shop scheduling, *Appl. Math. Comput.* **182**, 2, 1137–1148.
26. Grandpierre T., Lavarenne C., Sorel Y. (1999) Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors, in: *Proc. of the 7th International Workshop on Hardware/Software Co-Design, CODES'99*, March 3, Rome, Italy.
27. Berkelaar M., Eikland K., Notebaert P. (2004) *lpsolve: Open source (mixed-integer) linear programming system*, Eindhoven University of Technology, Eindhoven, Netherlands.
28. Gurobi Optimization, LLC (2016) *Gurobi optimizer reference manual*, Beaverton, Oregon, USA.
29. IBM ILOG (2017) *CPLEX User's Manual*, IBM Corp. Armonk, New York, USA.