Initialization Algorithms for Coupled Dynamic Systems

Labinot Polisi

Master's thesis
2017:E53

LUND UNIVERSITY

Faculty of Engineering
Centre for Mathematical Sciences
Numerical Analysis

CENTRUM SCIENTIARUM MATHEMATICARUM

Abstract

In this master thesis the consistent initialization problem is studied and three different algorithms were developed regarding the subject area - a graph algorithm used for solving the initialization problem, a parallel algorithm to enable parallel computations when solving the initialization problem and lastly a genetic algorithm used as a preprocessing stage for parallelization.

The thesis is based on the Python package PyFMI, a high-level package developed by Modelon AB for working with models compliant with the FMI standard.

The algorithms were tested on test cases consisting of several synthetic examples as well as in a simulation of a real industrial physical model. The analysis based on these test cases showed that the graph algorithm outperformed previously algorithms in terms of optimization, a speedup was achieved when using the parallel algorithm and the genetic algorithm was able to further increase the speedup factor.

Acknowledgments

Firstly, I would like to express my gratitude to my supervisors Emil Fredriksson at Modelon AB and Claus Führer at Lund University, Department of Numerical Analysis for their guidance and support of this thesis.

I would also like to thank all the people on Modelon that have been contributing to this thesis, especially Christan Winther who has been very helpful through out the project.

Notation

x	global state vector
x_i	ith global state
$x^{[i]}$	state vector of model i
$x_j^{[i]}$	jth state from state vector of model i
u	global input vector
y	global output vector
$f(x, u)$	global derivative function
$g(x, u)$	global output function
$c(y)$	coupling function

Contents

Chapter 1

Introduction

When dealing with modeling of complex dynamic systems where the model components are provided from different suppliers, there is a need for a standardized model definition that can incorporate exchanges and coupling between model components in a satisfactory way. The Functional Mock-up Interface provides such a definition with support for both model exchange and the ability to perform collaborative simulation, co-simulation, of compound systems. Since the model content of each component is usually protected, it is not possible to make use of the standard selection of methods for integrating and simulating the model equations. Instead, the simulation is done in a distributed manner, orchestrated by a so called master algorithm. Here, each model component solves its own set of equations internally and the master algorithm's objective is to organize the system as a whole and step the simulation forward in time.

In order to begin the simulation procedure, the system model must be initialized in a consistent manner. This is achieved by solving a system of algebraic equations and the problem of solving these specific equations is known as the consistent initialization problem. Although methods for solving this problem already exist, there is more than one way of initializing a system in a consistent way. This opens up the possibility of different approaches which might turn out to be more beneficial than others from a computational perspective. This thesis presents an initialization algorithm and its implementation within the FMI-standard via the python package PyFMI. Moreover, in order to obtain further performance gains, two methods which make use of multiple computer processes in parallel are developed and implemented. The objective of this thesis is to improve the computational time when solving the initialization problem by the use of these three developed algorithms.

The report is structured as follows:

- Chapter 1 - Introduction
- Chapter 2 - A discussion of co-simulation is given along with its mathematical description and the consistent initialization problem is introduced and explained. Moreover, an overview of the Functional Mock-up Interface is given and lastly a brief introduction of the open-source python package PyFMI.
- Chapter 3 - The graph theoretical concepts that makes up the base theory behind the algorithms are presented. Moreover, solution methods to the initialization problem is explained.
- Chapter 4 - A conceptual evaluation reduction algorithm is explained along with its motivation. The algorithm is tested on a real industrial example and also synthetic examples.
- Chapter 5 - A brief introduction to parallel computation is given and a parallel algorithm is outlined and tested for solving the initialization problem.
- Chapter 6 - The basic components of genetic programming is explained along with the use of a genetic algorithm for priming a graph for parallelization. The algorithm is outlined and tested.
- Chapter 7 - The implementation of the three different algorithms within the python package PyFMI are explained.
- Chapter 8 - Presenting the results of the algorithms, which are tested on real industrial examples and synthetic examples.
- Chapter 9 - A discussion of the results is given. Lastly, the conclusions of this thesis is presented along with some remarks on future development.

Chapter 2

Co-simulation and Functional Mock-up Interface

In this chapter we will introduce the concept of co-simulation, its background and importance to modern simulation technology along with its mathematical description. Moreover, a brief description of the FMI-standard will be given, which is the chosen setting of co-simulation in this thesis.

2.1 Co-Simulation

Co-simulation is a well established simulation technique used for dynamic systems. These systems are in general composed of weakly coupled subsystems (cf. Figure 2.2) i.e. the internal dynamics of each subsystem is in general not known. Since complex dynamic systems often involve multi-domain physics, the co-simulation approach has an inherit advantage of being able to combine specialized simulation tools used for different fields and signals in contrast to the monolithic approach [4].

A dynamic system is represented by the differential algebraic equation (DAE)

$$\begin{aligned} \dot{x} &= f(t, x, u) \\ y &= g(t, x, u), \end{aligned} \tag{2.1}$$

where x is the state vector, y is the output vector, u is the input vector, t is an independent variable, g is the output function and f is the derivative function.

Example 2.1 *(Dynamic System). Schematic view of a single dynamic system represented as a modular block pictured in Figure 2.1.*

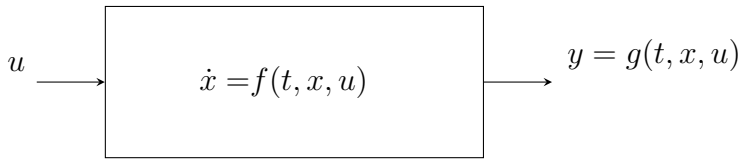


Figure 2.1: A single dynamic system.

2.2 Mathematical Description

Co-simulation is a simulation technique used for solving time-dependent problems on finite time intervals $[T_{start}, T_{end}]$ which have two or more connected subsystems. As mentioned in Section 2.1, the internal dynamics of each subsystem is in general not known and subsystems communicate with each other only through their inputs and outputs. Communication between the subsystems is restricted to a finite number of discrete time points T_n, $(0 < n < N)$, with $T_{start} = T_0 < T_1 < \ldots < T_N = T_{end}$, also known as *communication points*. This restriction of data exchange between the subsystems has the consequence that within each communication step, i.e., for $t \in (T_n, T_{n+1})$, all terms of the coupled problem that represent the coupling of subsystems have to be approximated by the use of extrapolation or interpolation techniques [4].

One distinguishes between a communication step T_n which is a step forward in time for the whole system and a *micro step* $t_{n,m}$, which is a step forward in time for a given subsystem. The two different types of steps are related such that $T_n < t_{n,0} < t_{n,1} \ldots t_{n,m} < T_{n+1}$ [2].

The subsystems are represented by modular blocks that are connected to other subsystems of the coupled problem by subsystem inputs and outputs cf. Figure 2.2. This modular method allows the subsystems of the coupled system to be solved separately with specific methods and micro step sizes for each subsystem. Such a method has an advantage over the monolithic approach regarding flexibility [4].

Example 2.2 *(Weakly Coupled Systems). A schematic description of a coupled system pictured in Figure 2.2 consisting of three models.*

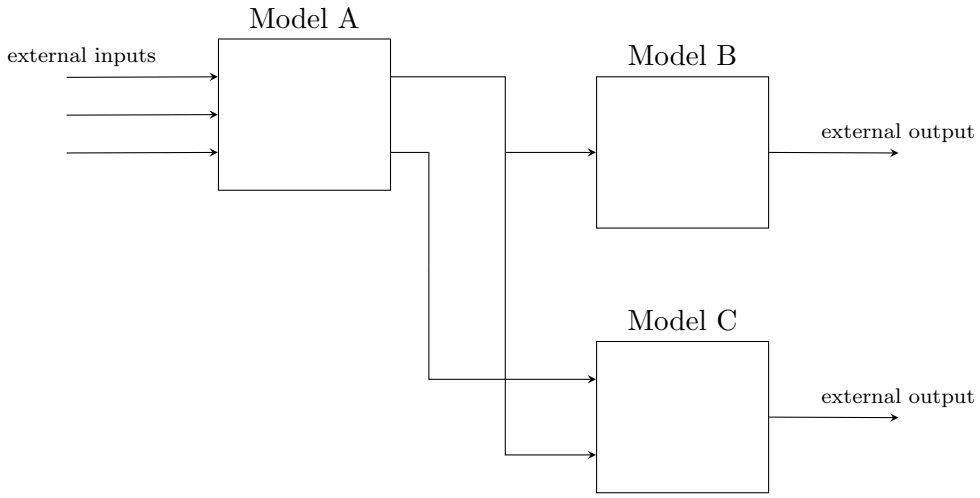


Figure 2.2: A system model consisting of three coupled models together with their connections.

In this thesis, we consider N coupled subsystems of the type described by Equation 2.1. Summarizing all the subsystems, we can express the DAE of the global system as

$$\dot{x} = \begin{bmatrix} f^{[1]}(t, x^{[1]}, u^{[1]}) \\ \vdots \\ f^{[N]}(t, x^{[N]}, u^{[N]}) \end{bmatrix} \tag{2.2}$$

$$y = \begin{bmatrix} g^{[1]}(t, x^{[1]}, u^{[1]}) \\ \vdots \\ g^{[N]}(t, x^{[N]}, u^{[N]}) \end{bmatrix} \tag{2.3}$$

$$u = \begin{bmatrix} c^{[1]}(t, x^{[1]}, u^{[1]}) \\ \vdots \\ c^{[N]}(t, x^{[N]}, u^{[N]}) \end{bmatrix} \tag{2.4}$$

$$x = \begin{bmatrix} x_1^{[1]} \\ \vdots \\ x_{j_1}^{[1]} \\ x_1^{[2]} \\ \vdots \\ x_{j_2}^{[2]} \\ \\ \vdots \\ \\ x_1^{[N]} \\ \vdots \\ x_{j_N}^{[N]} \end{bmatrix}, \quad y = \begin{bmatrix} y_1^{[1]} \\ \vdots \\ y_{k_1}^{[1]} \\ y_1^{[2]} \\ \vdots \\ y_{k_2}^{[2]} \\ \\ \vdots \\ \\ y_1^{[N]} \\ \vdots \\ y_{k_N}^{[N]} \end{bmatrix}, \quad u = \begin{bmatrix} u_1^{[1]} \\ \vdots \\ u_{l_1}^{[1]} \\ u_1^{[2]} \\ \vdots \\ u_{l_2}^{[2]} \\ \\ \vdots \\ \\ u_1^{[N]} \\ \vdots \\ u_{l_N}^{[N]} \end{bmatrix}, \tag{2.5}$$

where x is the global state vector, y is the global output vector, u is the global input vector, g is the global output function, f is the global derivative function and c is the global coupling function. The superscript specifies which subsystem is regarded and the subscript specifies the variable. For example, $x_1^{[2]}$ indicates the first state variable in the second subsystem. The same principle applies to the other DAE components [2].

Summarizing all components into vector form, we can express the global DAE in the more compact form,

$$\dot{x} = f(t, x, u), \tag{2.6a}$$
$$y = g(t, x, u), \tag{2.6b}$$
$$u = c(y). \tag{2.6c}$$

One distinguishes between an external input acting on the coupled system and an internal input which is determined by the coupling of the system. Consequently, the global input vector u consists of both external inputs and internal inputs.

A state-space linearized representation of Equation 2.6 can be formu-

lated as,

$$\dot{x} = Ax + Bu \tag{2.7a}$$
$$y = Cx + Du \tag{2.7b}$$
$$u = Ly \tag{2.7c}$$

with

$$A = \begin{bmatrix} A^{[1]} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & A^{[N]} \end{bmatrix}, \qquad B = \begin{bmatrix} B^{[1]} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & B^{[N]} \end{bmatrix}$$
$$C = \begin{bmatrix} C^{[1]} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & C^{[N]} \end{bmatrix}, \qquad D = \begin{bmatrix} D^{[1]} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & D^{[N]} \end{bmatrix}$$

where A is the state matrix, B is the input matrix, C is the output matrix, D is the feed-through matrix and L is the coupling matrix which maps the outputs y to the inputs u [2].

The simple case of a coupling is when we have a one-to-one coupling which is illustrated in Example 2.3. When the system coupling is more complicated, other methods may be required, cf. Section 3.2.

Example 2.3 *(Simple Coupling) Consider two coupled subsystems described by*

$$\dot{x}^{[1]} = -x^{[1]} + u^{[1]}, \qquad \dot{x}^{[2]} = -x^{[2]} + u^{[2]} \tag{2.8}$$
$$y^{[1]} = x^{[1]}, \qquad y^{[2]} = -u^{[2]} \tag{2.9}$$

and the coupling matrix

$$L = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

Then the coupling equations simply are

$$\begin{bmatrix} u^{[1]} \\ u^{[2]} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} y^{[1]} \\ y^{[2]} \end{bmatrix} \implies \begin{bmatrix} u^{[1]} \\ u^{[2]} \end{bmatrix} = \begin{bmatrix} y^{[2]} \\ y^{[1]} \end{bmatrix}.$$

Initialization

Simulating systems can in general be divided into three parts - initialization of the system, computation of the states for each time step and lastly, compiling the simulation results. In the FMI-standard these three parts correspond to the following respectively,

1. Instantiation and initialization phase: The FMUs of the system are unzipped, the XML descriptions parsed and loaded into memory along with the binaries and lastly the master algorithm sets their initial values and parameters. Mathematically, this corresponds to solving Equation 2.10.

2. Computation phase: The master algorithm advances the simulation forward in time. This is done by computing the states of the variables by calling the `fmiGet` and `fmiSet` functions and then calling the `fmiDoStep` function on each FMU [1]. This corresponds to integrating the state derivatives of the system, cf. Equation 2.6.

3. Termination phase: The simulation is completed and the results are made available.

Due to Equation 2.6b, 2.6c which can be interpreted as algebraic constraints in the DAE, the input u cannot be chosen arbitrarily but has to be consistent. Consider Example 2.3, with $x^{[1]}(t_0) = 1$, $u^{[1]} = 1$ and $u^{[2]} = -1$, we get

$$\begin{aligned} y^{[1]} &= 1 \\ y^{[2]} &= 1. \end{aligned}$$

From the coupling, we also have

$$\begin{aligned} u^{[1]} &= y^{[2]} \\ u^{[2]} &= y^{[1]}. \end{aligned}$$

We have an inconsistency since $u^{[2]} = -1 \neq 1 = y^{[1]}$.

Definition 2.1 *(The Consistent Initialization Problem) To start the simulation of a system described by Equation 2.6 from a consistent initial state, it is required to solve Equation 2.10,*

$$\begin{aligned} y &= g(T_0, x, u) \\ u &= c(y) \end{aligned} \tag{2.10}$$

for global input vector u and global output vector y with global state vector x fixed [2].

The coupling is assumed to be known and $c \in \mathcal{C}^0$. To find a sufficient condition on solving the algebraic equation for coupled systems with direct feed-through, we look at the state-space linearized representation, cf. Equation 2.7b, 2.7c. Eliminating the input u in Equation 2.7b, we get

$$y = Cx + DLy \implies y = (I - DL)^{-1}Cx.$$

This is solvable when $(I - DL)$ is non-singular [2].

Definition 2.2 *(Direct Feed-Through) ([2]). Let $y^{[i]} = g^{[i]}(t, x^{[i]}, u^{[i]})$ be the outputs from model i. If output $y_k^{[i]}$ depends on the input $u_l^{[i]}$ such that*

$$\frac{\partial g_k^{[i]}(t, x^{[i]}, u^{[i]})}{\partial u_l^{[i]}} \neq 0, \tag{2.11}$$

then model i has direct feed-through between variables $u_l^{[i]}$ and $y_k^{[i]}$.

2.3 FMI Overview

The *Functional Mock-up Interface* (FMI) is a standardized interface for system models which was developed under the European project MODELISAR and is now developed and maintained as a Modelica Association project. The FMI standard consists of two main types of protocols, *FMI for model exchange* and *FMI for co-simulation*. The main difference between these two is that if the system model being described is a continuous system, the modular block unit of which the whole system is built upon called *Functional Mock-up Unit* (FMU), are simulated using the tools in the imported environment. A co-simulation FMU on the other hand is packaged together with an internal solver [1]. Since this thesis main focus is on co-simulation, FMI for model exchange will not be further discussed. For a more in-depth read on FMI for Model exchange, cf. [1].

The interface defines the structure of the FMU as well as the state machine of the FMU, i.e. which operations are allowed at any given point and how the call sequence is defined. When modeling a system, the FMU:s are usually the sub-systems that make up the whole system but they can also be instances of other types, e.g. a coupling part of a simulation tool. Each FMU is packaged as a zip-file consisting of two main parts, which are the following,

- The C sources which contain the model equations and the run-time libraries used in the model. These are usually packaged together into binary form for the specific target machine.

- An XML-file that contains the definition of the variables according to the standard, such as the name, type, value reference etc. The XML-file also contains other information such as unit definitions, the model name and which tool was used to generate the FMU [1].

An overview of the data flow between the environment and an FMU is given in Figure 2.3. The red arrows correspond to the data that is provided to the FMU, e.g. initial values and input u^i for a given communication point. Conversely, the blue arrows correspond to the data that is provided from the FMU to the external environment e.g. other FMUs or a simulator [1].

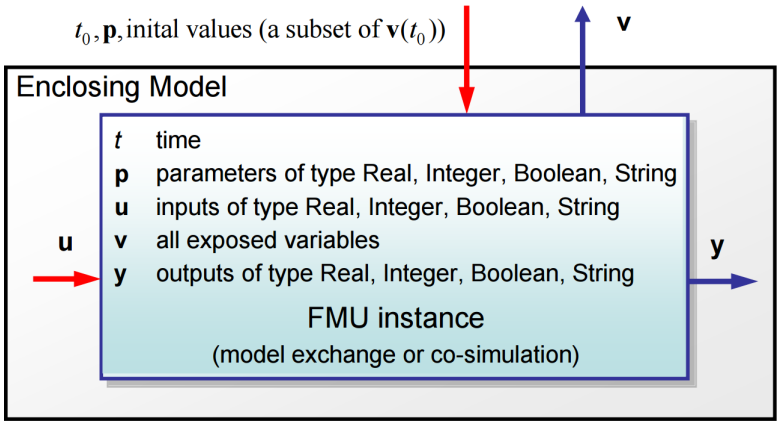


Figure 2.3: Data flow between the environment and an FMU. Blue arrows: Information provided by the FMU. Red arrows: Information provided to the FMU [1].

Features and Restrictions

In the FMI standard there are several key features and restrictions that have an impact on which types of algorithms that can be used. In this section, the most relevant features and restrictions to this thesis will be presented.

Feature 2.1 *(Dependency information) ([2]). Information about which inputs directly impact the outputs is available.*

Feature 2.2 *(Save/Get state) ([2]). There is support for serializing the internal state of an FMU.*

Restriction 2.1 *(FMU Serialization). There is not support for serializing the entire FMU.*

Restriction 2.1 has a very direct consequence on the problem that is considered in this thesis. More specific, the restriction has consequences on the design of parallel algorithms, which we will see later in Chapter 5, since there is no support to pass instances of FMUs between processes. Essentially, one has to resort to either lock each FMU to one process or load the same FMUs across multiple processes and synchronize the internal states with Feature 2.2. Both methods have a degrading effect on the performance of a parallel algorithm. This issue will be further discussed in Chapter 5.

2.4 PyFMI

JModelica.org is an extensible Modelica-based open source platform for optimization, simulation and analysis of complex dynamic systems. The main objective of the project is to create an industrially viable open source platform for optimization of Modelica models, while offering a flexible platform serving as a virtual lab for algorithm development and research. As such, JModelica.org provides a platform for technology transfer where industrially relevant problems can inspire new research and where state of the art algorithms can be propagated from academia into industrial use. JModelica.org is a result of research at the Department of Automatic Control, Lund University, and is now maintained and developed by Modelon AB in collaboration with academia.

PyFMI is a package for loading and interacting with FMUs, both for Model Exchange and Co-Simulation. PyFMI offers a Python interface for interacting with FMUs and enables for example loading of FMU models, setting of model parameters and evaluation of model equations. PyFMI is available as a stand-alone package or as part of the JModelica.org distribution. Using PyFMI together with the Python simulation package Assimulo adds industrial grade simulation capabilities of FMUs to Python.[1]

[1]Citation from the official web page of JModelica.org.

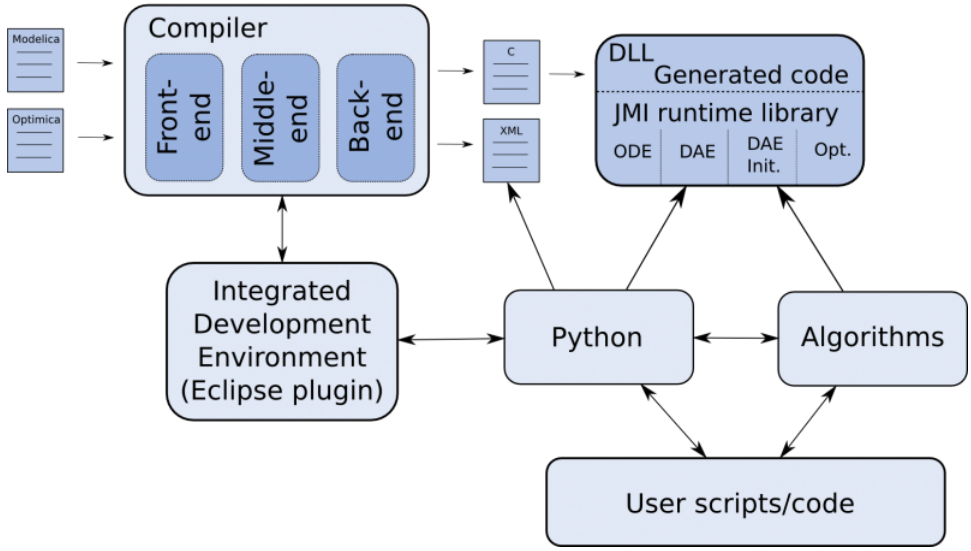


Figure 2.4: Overview of the JModelica.org platform.

Chapter 3

Graph Theoretical Tools for the Consistent Initialization Problem

In this chapter we introduce the graph theoretical concepts and tools necessary for understanding and solving the initialization problem, cf. Definition 2.1. The graph theoretical concepts introduced in this chapter are also used for the initialization algorithm presented in Chapter 4 as well as the parallel algorithm presented in Chapter 5.

In Section 2.2 the problem of initializing a system was presented where we saw the need to solve Equation 2.10. However, solving the algebraic equations explicitly is not alway required.

Consider the system illustrated in Example 3.1 with the coupling defined as

$$y_1^{[1]} = u_2^{[2]}, \qquad y_1^{[2]} = u_2^{[1]}. \tag{3.1}$$

A path which is free of cycles can be found by traversing the connections in the system, starting from the external inputs T_{env} and T_{ref} to the external output T which belongs to the model Plant. This means that the external output T can be computed by evaluating the models in sequence. When all values are set, the system has been initialized and the simulation can begin.

This is a convenient way of initializing the system in contrast to the other methods which makes use of non-linear solvers, e.g. Newton's method, and are thus faced with the problem of choosing a good initial guess [2]. This method of initialization will be discussed more in-depth in Section 3.3, for now the reader notes that it is possible to initialize a system by using

its structural dependency information. This information is assumed to be available due to Feature 2.1.

Example 3.1 *(Graph of Controlled Temperature). In this example we see a simple system with two coupled models, cf. Figure 3.1. The first model, Controller, outputs variable* On Switch *to true or false based on the inputs* T *and* T_{ref}*. The second model, Plant, takes two inputs,* T_{env} *and* On Switch*, and outputs the temperature* T*.*

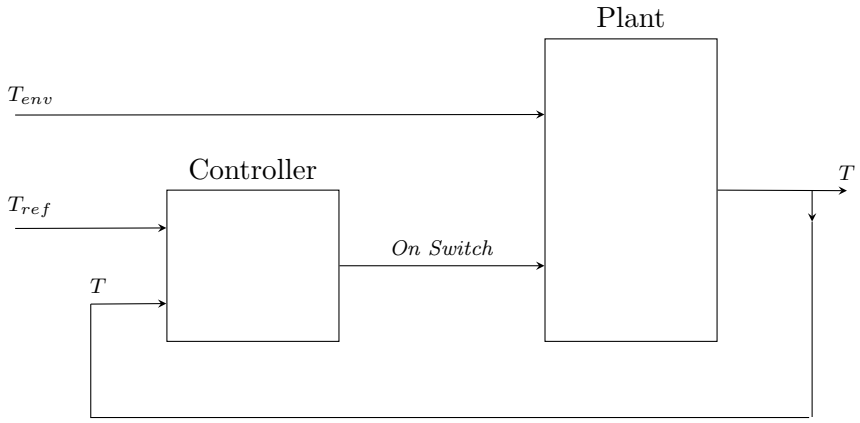


Figure 3.1: A system consisting of two coupled models together with their connections.

3.1 Directed Graphs

For compound systems there is usually a strict information flow that goes from one variable to another. Therefore a *directed graph* is best suited for studying how these variables are related to each other.

Definition 3.1 *(Directed Graph). A directed graph or digraph* $\mathcal{G}_D(\mathcal{V}, \mathcal{E})$ *is a set of vertices* $\mathcal{V}$ *and a set of edges* $\mathcal{E}$*, where the edges are ordered pairs of vertices of* $\mathcal{V}$*.*

Definition 3.2 *(Cycle). A cycle of a graph* $\mathcal{G}(\mathcal{V}, \mathcal{E})$ *is a subset of the edge set* $\mathcal{E}$ *with at least two edges that forms a path such that the first node of the path corresponds to the last.*

Example 3.2 *(Digraph with a Cycle). In this example we can see a digraph pictured in Figure 3.2 where the vertices c, d and e form a cycle.*

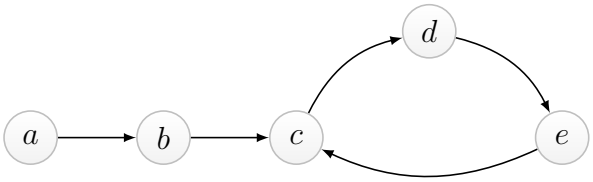


Figure 3.2: A digraph with a cycle.

Definition 3.3 *(Directed Acyclic Graph). A directed acyclic graph or DAG is a directed graph containing no directed cycles.*

Example 3.3 *(DAG). In this example we can see a digraph pictured in Figure 3.3 which does not contain any cycles.*

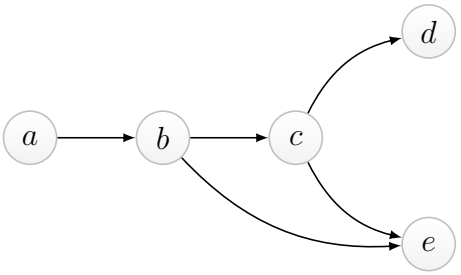


Figure 3.3: A digraph without cycles.

Definition 3.4 *(Precedence Constraint). Given a directed graph* $D = \mathcal{G}(\mathcal{V}, \mathcal{E})$ *and an ordering* L *of the vertices where* $L(\cdot)$ *gives the index of the vertex in* L*, we say that the condition*

$$(v, w) \in \mathcal{E} \implies L(v) < L(w),$$

is a precedence constraint on L*.*

As we will see later in Section 3.3, we will need to retrieve an ordering of the vertices of a DAG such that the precedence constraints are respected. Such an ordering is called a *topological sort* or *topological ordering*. There exists a number of algorithms that can construct a topological ordering of any DAG with a time complexity of $\mathcal{O}(|\mathcal{V}| + |\mathcal{E}|)$, essentially by traversing the graph such that each vertex is visited once and only once. In general, this ordering is not unique and by Lemma 3.1 we are guaranteed that there exists at least one topological ordering if the graph we are using is a DAG.

Definition 3.5 *(Topological Ordering). A topological ordering of a DAG $\mathcal{G}(\mathcal{V}, \mathcal{E})$ is a ordering of all its vertices such that if $(v, w) \in \mathcal{E}$, then v appears before w.*

Lemma 3.1 *(Topological Ordering and Directed Graphs). A digraph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ is acyclic if and only if there exists a topological ordering of its vertices.*

Example 3.4 *(Topological Ordering). A topological ordering of the DAG defined in the previous example in Figure 3.3. As we can see in Figure 3.4, the vertex ordering is $[a, b, c, e, d]$ and all precedence constraints are respected.*

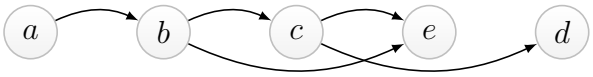


Figure 3.4: A topological ordering of 3.3 from Example 3.4. Note that all edges are directed from left to right.

3.2 Strongly Connected Components

Definition 3.6 *(Strongly Connected Graph) ([2]). Let $\mathcal{G}(\mathcal{V}, \mathcal{E})$ be a directed graph. If for every pair of vertices $v, w \in \mathcal{V}$ there is a directed path from v to w and w to v, then $\mathcal{G}(\mathcal{V}, \mathcal{E})$ is a strongly connected graph.*

Definition 3.7 *(Strongly Connected Component) ([2]). Let $\mathcal{G}(\mathcal{V}, \mathcal{E})$ be a directed graph and let $\mathcal{G}(\bar{\mathcal{V}}, \bar{\mathcal{E}})$ where $\bar{\mathcal{E}} = \{(v, w) \in \mathcal{E} \mid v, w \in \bar{\mathcal{V}}\}$ be a subgraph of $\mathcal{G}(\mathcal{V}, \mathcal{E})$. If $\mathcal{G}(\bar{\mathcal{V}}, \bar{\mathcal{E}})$ is a strongly connected graph, then $\mathcal{G}(\bar{\mathcal{V}}, \bar{\mathcal{E}})$ is said to be a strongly connected component of $\mathcal{G}(\mathcal{V}, \mathcal{E})$.*

A vertex is regarded as trivially strongly connected to itself. Every non-trivial SCC contains at least one cycle. This is evident when recalling Definition 3.2 which states that given a directed graph with at least two vertices, we have a cycle if there exists a path such that we can return to the vertex we started our path from. In the case of strong connectivity, we are interested in all the cycles of a given graph. The fact that every non-trivial SCC contains at least one cycle means that a digraph is acyclic if and only if it has no strongly connected subgraphs with more than one vertex.

Example 3.5 *(Strongly Connected Component). In this example we see the strongly connected components of a digraph marked in gray in Figure 3.5.*

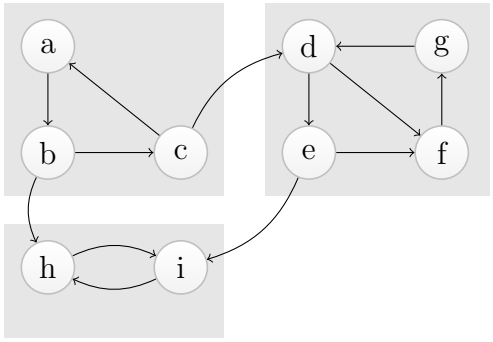


Figure 3.5: SCCs of the digraph from Example 3.5.

Algebraic loops

Component based modeling can sometimes lead to systems where no explicit evaluation sequence can be found to compute the outputs y_j. This happens when an input with direct feed-through is set by an output from the same model, either directly or by a feed-back path through other models which have direct feed-through. Such a loop is represented as a non-trivial SCC, i.e. a SCC with more than one element, in the associated graph of the coupled system, cf. Example 3.6 [2].

Example 3.6 *(Algebraic Loop). A system with an algebraic loop.*

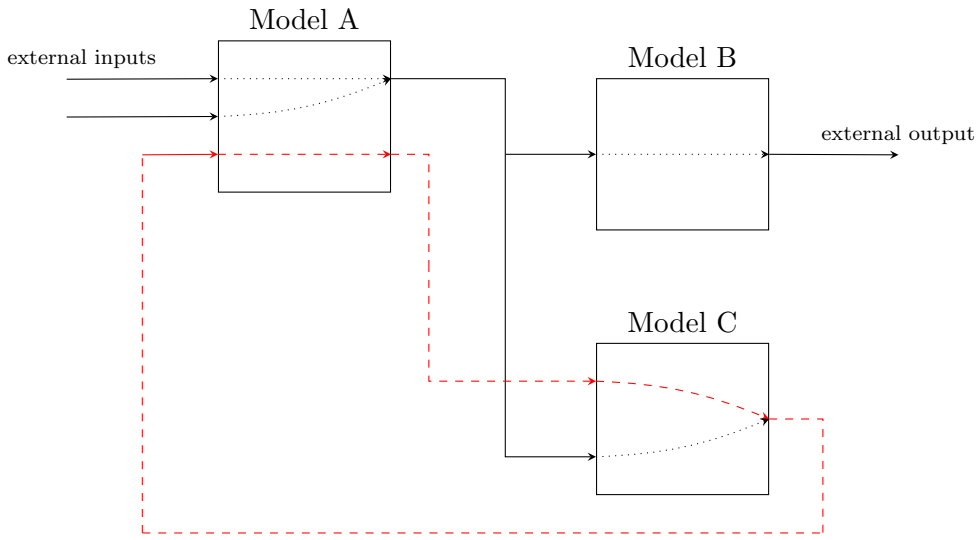


Figure 3.6: A system model with an algebraic loop marked in dashed lines.

It is necessary to solve the arising algebraic equations of an algebraic loop whenever they are present in a system. This is in general a non-

trivial problem. In PyFMI's master algorithm, this is done by grouping the variables that constitutes a loop into an SCC and solve them simultaneously [2]. More specific, this means that we rewrite Equation 2.10 as

$$y - g(t, x, c(y)) = 0$$

and use the left-hand side as a residual function to calculate the output vector y with a non-linear solver. The outputs are then used to set the inputs as in the simple case, cf. Example 2.3.

Tarjan's Strongly Connected Components Algorithm

As mentioned in Section 3.1, there exist several algorithms that can construct a topological ordering of any DAG with a time complexity of $\mathcal{O}(|\mathcal{V}| + |\mathcal{E}|)$. One such algorithm is the one proposed by Robert Tarjan which is used in this thesis, cf. Algorithm 1 [8].

Algorithm 1 Tarjan's strongly connected components algorithm [2]

Require: A directed graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$.

lowlink $\leftarrow \{v : \textit{not numbered} \mid \forall v \in \mathcal{V}\}$
number $\leftarrow \{v : \textit{not numbered} \mid \forall v \in \mathcal{V}\}$
$i \leftarrow 0$
for $v \in \mathcal{V}$ **do**
    if v *not numbered* **then**
        STRONGCONNECT(v)
    end if
end for
procedure STRONGCONNECT(x)
    lowlink$(x) \leftarrow i$
    number$(x) \leftarrow i$
    stack.append.(x)
    $i \leftarrow i + 1$
    for $w|(x, w) \in \mathcal{E}$ **do**
        if w *not numbered* **then**
            STRONGCONNECT(w)
            lowlink(x) = min(lowlink(x),lowlink(w))
        else if number(w) < number(x) **and**$w \in$ stack **then**
            lowlink(x) = min(lowlink(x),lowlink(w))
        end if
    end for
    if number(x) = lowlink(x) **then**
        create new strongly connected component.
        while stack **and**number(last in stack) $\geq$ number(x) **do**
            add last in stack to the component and remove from stack
        end while
    end if
end procedure

3.3 Initialization with a Structural Approach

Solving the initialization problem, Definition 2.1, with a structural approach is done by finding an evaluation order of the inputs u_j's. The evaluation order can either be an explicit sequence of input-outputs, in which case the algebraic equations can be solved by a forward evaluation, or we have at least one an algebraic loop in the system in which case the relevant algebraic

equations needs to be solved together e.g. with a non-linear solver, cf. Section 3.2 [2].

This is done by transforming the system into a directed graph. The directed graph $\mathcal{G}(\mathcal{E}, \mathcal{V})$ is defined such that each output $y_k^{[i]} \in y$ and input $u_l^{[i]} \in u$ correspond respectively to a vertex in $\mathcal{G}$. An edge is added between two vertices $y_k^{[i]}$ and $u_l^{[i]}$ whenever there exists direct feed-through from $u_l^{[i]}$ to $y_k^{[i]}$ according to Definition 2.2. Further, an edge is added between y_k and u_l if the variables are coupled, i.e. if

$$\frac{\partial c_l(y)}{\partial y_k} \neq 0 \quad [2]. \tag{3.2}$$

Constructing the graph with the vertices and edges as defined above, we are both able to analyze the graph for structural loops, i.e. non-trivial SCCs which correspond to algebraic loops in the system and we are also able to retrieve a topological ordering of the graph which corresponds to a proper evaluation order of the inputs. Identifying structural loops and determining a topological ordering are the two main goals when solving the initialization problem with a graph-theoretical approach. After identifying which components of the graph constitute structural loops, the components of that specific loop are then grouped together into an SCC [2]. This transforms the digraph into a DAG where the vertices are instead the SCCs and thus enables the use of Lemma 3.1.

As mentioned in the introduction, the main purpose of this thesis is to find the *optimal solution of the initialization problem*, as given by Definition 3.8. Depending on which factors are considered, the means of achieving this will change as we shall see in Chapter 5 and 6.

Definition 3.8 *(Optimal Solution of the Initialization Problem). The optimal solution of the initialization problem, Equation 2.10, is the method which initializes the system with minimal execution time.*

Example 3.7 *(Initializing Coupled System [2]). Consider a system where we have two different models with the global state vector and output function defined as*

$$\dot{x}^{[1]} = x^{[1]} + u^{[1]}, \qquad \dot{x}^{[2]} = x^{[2]} + u^{[2]} \tag{3.3}$$

$$y^{[1]} = \frac{x^{[1]}}{2}, \qquad y^{[2]} = \frac{x^{[2]}}{2} + u^{[2]} \tag{3.4}$$

with $x^{[1]}(t_0) = 2$, $x^{[2]}(t_0) = -4$ *and the coupling function defined as*

$$u^{[2]} = y^{[1]}, \quad u^{[1]} = y^{[2]}. \tag{3.5}$$

Since we have values for the initial states $x(t_0)$, we can get the value of $y^{[1]}$ with the output function in Equation 3.4. Via the coupling in Equation 3.5, we then get the value to set for input $u^{[2]}$. Next, we get the value of $y^{[2]}$ with the output function and lastly set $u^{[1]}$ via the coupling.

This results in the following initialization sequence.

1. *Get $y^{[1]}$:* $y^{[1]} = 1$
2. *Set $u^{[2]}$:* $u^{[2]} = 1$
3. *Get $y^{[2]}$:* $y^{[1]} = -1$
4. *Set $u^{[1]}$:* $u^{[1]} = -1$

Chapter 4

Reduction of Model Function Evaluations

In this chapter we will present an algorithm for reducing the number of *model function evaluations* during the initialization. The algorithm is then demonstrated on a synthetic example as well as on a real industrial system.

Definition 4.1 *(Model Function Evaluation). Given a model with model number i, an evaluation of the internal dynamics of the model at a time instance t_n corresponds to solving the DAE*

$$\begin{aligned} \dot{x}^{[i]} &= f^{[i]}(t_n, x^{[i]}, u^{[i]}) \\ y^{[i]} &= g^{[i]}(t_n, x^{[i]}, u^{[i]}). \end{aligned} \tag{4.1}$$

Since the evaluation order of a system model is a topological ordering, the evaluation order is in general not unique. This property is important since it allows for a reordering of the evaluation order which is more beneficial from a computational standpoint without interfering with the initialization and simulation results. The underlying reason for the potential of making performance gains by considering the evaluation order is that if any input $u_l^{[i]}$ has been set between retrieving outputs $y_k^{[i]}$, an internal evaluation of the dynamics in the subsystem is triggered, cf. Definition 4.1. This evaluation is assumed to be computationally expensive and thus by reducing the number of such evaluations, we also reduce the computational cost of the whole initialization procedure.

Assumption 4.1 *If any input $u_l^{[i]}$ has been set between retrieving outputs $y_k^{[i]}$, an internal evaluation of the dynamics in a co-simulation FMU is triggered [2].*

Assumption 4.2 *Evaluation of a subsystem's dynamics is expensive [2].*

Example 4.1 *(Non-unique Evaluation Order). Consider a system with three coupled models with the input-output relations defined as*

$$y_1^{[1]} = u_1^{[1]} + u_2^{[1]} \tag{4.2}$$
$$y_2^{[1]} = u_3^{[1]} \tag{4.3}$$
$$y_1^{[2]} = u_1^{[2]} \tag{4.4}$$
$$y_1^{[3]} = u_1^{[3]} + u_2^{[3]} \tag{4.5}$$

and the coupling defined as

$$u_1^{[2]} = y_1^{[1]} \tag{4.6}$$
$$u_1^{[3]} = y_2^{[1]} \tag{4.7}$$
$$u_2^{[3]} = y_1^{[1]}. \tag{4.8}$$

The associated graph of the system shown in Figure 4.1 does not have an unique evaluation order. Two different examples are given below:

Sequence 1:

$$[u_2^{[1]}, u_1^{[1]}, y_1^{[1]}, u_3^{[1]}, y_2^{[1]}, u_1^{[2]}, y_1^{[2]}, u_2^{[3]}, u_1^{[3]}, y_1^{[3]}].$$

Sequence 2:

$$[u_2^{[1]}, u_1^{[1]}, y_1^{[1]}, u_3^{[1]}, y_2^{[1]}, u_2^{[3]}, u_1^{[3]}, y_1^{[3]}, u_1^{[2]}, y_1^{[2]}].$$

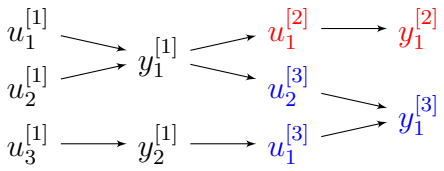


Figure 4.1: Graph of the initialization problem in Example 4.2.

The problem of minimizing the model function evaluations can be seen as a combinatorial optimization problem, since the objective is to find a partition of the graph which will yield a minimal or near-minimal number of SCCs as motivated by Assumption 4.2. Given Definition 3.8, the optimal or near-optimal solution is achieved by minimizing the number of model

function evaluations, i.e. group together as many of $y^{[i]}$'s as possible into a SCC while respecting the precedence order constraints.

An algorithm for reducing the number of model function evaluations has been given previously, shown in Algorithm 2. The basic outline of the algorithm is to first group all output vertices from a model that are not included in a feed-through term and then group all output nodes from a model that are connected to inputs which are not included in a feed-through term. After this, an evaluation sequence is computed along with the identification of structural loops by deploying Algorithm 1. After these steps, a procedure of modifying the evaluation sequence begins. This is done by considering each SCC in the sequence which are single outputs and check if it is possible to move it so that it is evaluated earlier without violating any precedence constraints.

Algorithm 2 Computing a reduced initial evaluation order

Require: A directed graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$.
Require: Information about which nodes belong to same model, M.
Require: Information about which nodes are outputs.
1: {Group all outputs, from a model, that are not included in a feed-through term.}
2: {Group all outputs, from a model, that are connected to inputs which are not included in a feed-through term.}
3: $\mathcal{F} = \textbf{Tarjan}(\mathcal{G})$ {Compute the strongly connected components}
4: $i \leftarrow 0$
5: **while** $i < \dim(\mathcal{F})$ **do**
6: $f_i \in \mathcal{F}$
7: $b \leftarrow 0$
8: **if** $\dim(f_i) = 1$ **and**$v \in f_i | v$ output **then**
9: **for** $j \in \{0, \ldots, i-1\}$ **and** $e_j \in \mathcal{F}$ **do**
10: **if** $\dim(e_j) = 1$ **and** $w \in e_j | w$ output, $w, v \in M^{[k]}$ **and** v not a child of w **then**
11: $f_i \leftarrow \{f_i, e_i\}$ {Update the ith item in $\mathcal{F}$, by joining f_i and e_i into one}
12: $b \leftarrow 1$ {Do not update the counter}
13: **break**
14: **end if**
15: **end for**
16: **end if**
17: **if** $b = 0$ **then**
18: $i \leftarrow i + 1$ {Increment counter}
19: **end if**
20: $\mathcal{G} \leftarrow \mathcal{G}/f_i$ {Group nodes in a strongly connected component}
21: **end while**

In this thesis, an alternative algorithm for reducing the number of model function evaluations has been developed. The goal of the algorithm is to compute an evaluation order, group as many outputs which belong to the same model as possible and detect algebraic loops. An outline for this alternative algorithm is as follows. We construct a digraph as described previously and modify the graph such that all vertices that are outputs, belong to the same model and are on the same *precedence order level* are grouped together. More formally, this means we form a collection of sets $\mathcal{L}$ such that

$$\begin{aligned} \mathcal{L}_0 &= \{v \in \mathcal{V} \mid \forall u \in \mathcal{V},\ (u,v) \notin \mathcal{E}\} \\ \mathcal{L}_{i+1} &= \{v \in \mathcal{V} \mid \forall u \in \mathcal{V},\ (u,v) \in \mathcal{E} \implies u \in \bigcup_{k=0}^{i} \mathcal{L}_k\} \end{aligned} \tag{4.9}$$

so that all elements in a set $\mathcal{L}_i \in \mathcal{L}$ are said to belong to the precedence order level i [5] [7]. For each $\mathcal{L}_i$ we then check if there exists outputs belonging to the same model, in which case we group these together into an SCC. These vertices can be grouped together without creating any algebraic loop due to how the sets were constructed. This first step will potentially decrease the number of possible groupings from one SCC to another and thus simplify the graph. This will be useful for the next step of the algorithm.

To further reduce the model function evaluations we now consider which possible groupings exist from one SCC to another under the condition that each one of these potential groups do not form an algebraic loop on its own. In general, if there exists several possible groupings, it is not the case that we can trivially group them all, since one grouping will affect the precedence constraints of the graph and thus another possible group might not be feasible afterwards, i.e. grouping all potential groups might introduce algebraic loops. Because of this fact, the groupings are done in an iterative manner where we calculate all the possible groups, pick one of these possible groups according to some heuristic and perform the merge and then start over again by calculate the possible groups in this newly transformed graph. This iterative procedure continues until there are no more possible groupings left to do.

The heuristic which is used to determine which possible group to actualize was found experimentally. The basic principle is, given two or more possible groups we first search for which y_j belongs to the highest precedence level order, i.e. the set $\mathcal{L}_i$ with the highest index, and fixate it. Afterwards, we consider all the other SCCs which can be potentially grouped together with our fixated y_j. We choose the SCC which has the highest precedence level order, y_i. The SCCs that are grouped together into one SCC in this iteration are thus (y_j, y_i). The proposed algorithm is shown in Algorithm 3.

Example 4.2 *(Outputs and direct feed-through). In this example we consider the initialization problem represented by the graph in Figure 4.2. There are two connected models where model 2 has multiple outputs. From the figure we note that the outputs from model 2 are connected to inputs which are not included in a feed-through term* ($u^{[3]}$)*. Thus, the outputs from model 2*

can be grouped, without creating an algebraic loop. Furthermore, this results in a minimal number of model function evaluations. The resulting graph is shown in Figure 4.3.

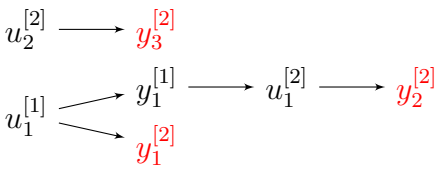


Figure 4.2: Graph of the initialization problem in Example 4.2.

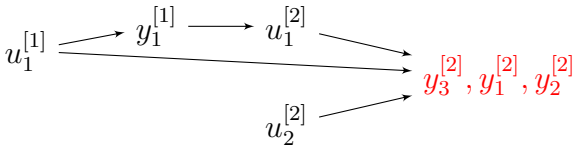


Figure 4.3: Graph of the solution of the initialization problem from Example 4.2 where all the outputs from the second model, $y^{[2]}$, have been joined.

Algorithm 3 Alternative algorithm for computing a reduced initial evaluation order

Require: A directed graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$.
Require: Information about which nodes belong to same model, M.
Require: Information about which nodes are outputs.

1: {Group all outputs, from a model, that are on the same precedence order level.}
2: $\mathcal{F} = \textbf{Tarjan}(\mathcal{G})$ {Compute the strongly connected components}
3: **while** $\dim(\mathcal{S}) \neq \emptyset$ **do**
4: $\quad \mathcal{S} \leftarrow \text{COMPUTE\_POSSIBLE\_GROUPINGS}(\mathcal{G}_i)$
5: $\quad f_i \leftarrow \text{HIGHEST\_LEVEL}(\mathcal{S})$
6: $\quad \mathcal{G} \leftarrow \mathcal{G}/f_i$ {Group nodes in a strongly connected component}
7: **end while**

Algorithm 3 Computing a further reduced initial evaluation order (continued)

procedure COMPUTE_POSSIBLE_GROUPINGS($\mathcal{G}(\mathcal{V}, \mathcal{E})$)
    $\mathcal{P} \leftarrow \emptyset$
    for $i \in \dim(\mathcal{V})$ **do**
        $N \leftarrow \emptyset$
        $f_i \leftarrow \mathcal{V}_i$
        if $v \in f_i | v$ output **then**
            for $j \in \{0, \ldots, i-1\}$ **and** $e_j \in \mathcal{V}$ **do**
                if $w \in e_j | w$ output, $w, v \in M^{[k]}$ **and** v, w not on the same path **then**
                    $N_i \leftarrow \{v, w\}$ {Update the ith item in N, by joining v and w into one}
                end if
            end for
        end if
        if $\dim(N) > 0$ **then**
            $\mathcal{P}_i \leftarrow N_i$
        end if
    end for
    return $\mathcal{P}$
end procedure

4.1 Random Graph Generator

In order to construct a large enough collection of examples, a random graph generator was implemented. The purpose of the generator was at first to simplify the search of graph instances where Algorithm 2 did not partition the graph optimally. For the purpose of obtaining the optimal solution, the generator was used in conjunction with an exhaustive search method.

The general idea behind the generator is to create a uniform spanning tree with the use of a random walk and then randomly select nodes to create the desired amount of edges. The algorithm starts by picking a random vertex from the graph which is then followed by a random walk. For each vertex which has not been encountered before, we form a new edge from the previous vertex to the current and save it. When we have performed a random walk on all vertices, we proceed by adding new random edges until we have reached the desired number. This whole procedure is then repeated for the desired number of times to produce the corresponding number of disconnected components in the graph.

Definition 4.2 *(Spanning Tree). A spanning tree $\mathcal{T}_\mathcal{G}$ of a graph $\mathcal{G}(\mathcal{E}, \mathcal{V})$ is a connected graph with no cycles that includes every vertex of $\mathcal{G}(\mathcal{E}, \mathcal{V})$ and every edge in $\mathcal{T}_\mathcal{G}$ belongs to $\mathcal{G}(\mathcal{E}, \mathcal{V})$.*

Definition 4.3 *(Uniform Spanning Tree). A spanning tree chosen randomly from among all possible spanning trees with equal probability is called a uniform spanning tree.*

4.2 Exhaustive Search Method

An exhaustive search, also known as a brute-force search, is a method for solving a problem by systematically searching for a solution in the entire search space. This is done by simply evaluating each candidate solution and check if the solution satisfies the problem constraints. A property to the exhaustive search method is that given that a solution exists, we will always obtain it by the end of the procedure. When the problem to be solved is an optimization problem, one is usually forced to check all feasible solutions to guarantee that the solution obtained is indeed optimal. This method is for practical reasons not feasible as a solution to the problem of reducing the model function evaluations. Instead, this is strictly used as a reference when evaluating the performance of the heuristic algorithms. In this thesis, the problem is to find a heuristic algorithm that provides a near-optimal solution within a reasonable execution time.

4.3 Case Studies

Synthetic Test Case

In this example, six models with feed-through are connected. A graph of the couplings is shown in Figure 4.4. The example is intended to illustrate Algorithm 3 where the number of model function evaluations is reduced.

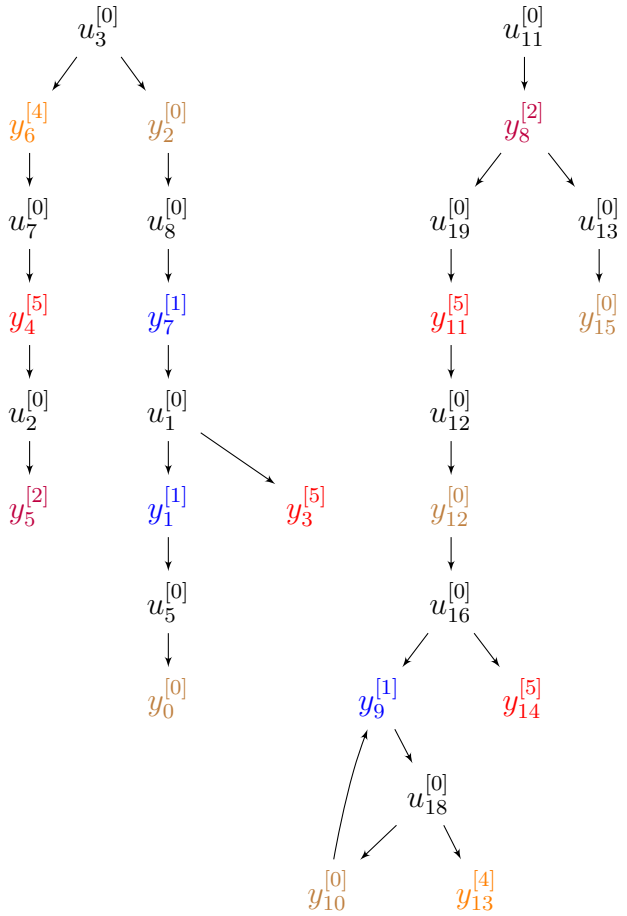


Figure 4.4: Graph showing six coupled models where as many as possible of the outputs for each model should be joined.

The first step in the algorithm is to group outputs which belong to the same model and the same precedence order level, where the order levels are computed according to Equation 4.9. In this case the only group of outputs that satisfy these conditions are the outputs $y_4^{[5]}$ and $y_{11}^{[5]}$ which both belong to the fourth level, cf. Figure 4.5.

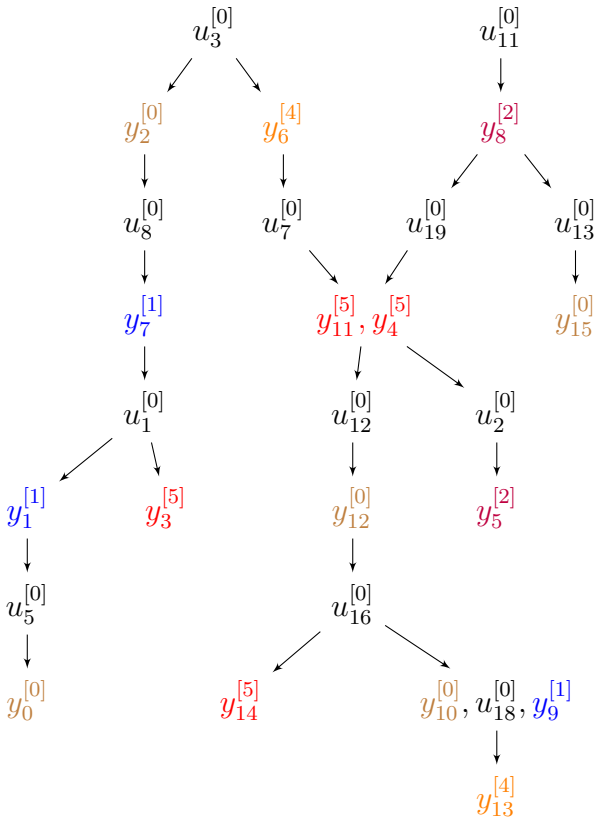


Figure 4.5: Result after the first steps of Algorithm 3 where vertices of the same precedence level order has been joined and a cycle has been grouped.

The next step in Algorithm 3 is to compute a first evaluation order and also to identify any cycles in the graph and when present, group them together. This is done by applying Algorithm 1 on the graph. In this case, the vertices $y_{10}^{[0]}$, $u_{18}^{[0]}$ and $y_9^{[1]}$ form a cycle and are thus grouped together, cf. Figure 4.5.

After these two steps, a final iterative procedure follows which terminates when all possible groupings have been exhausted. The first step in the procedure is to identify the output which satisfies the conditions of having the highest level and at least one other possible output to be grouped with. The output with the highest level in this case is $y_{13}^{[4]}$, but since grouping this output with any of the outputs which belong to the same model would yield in a cycle, this output does not satisfy our second condition for being selected.

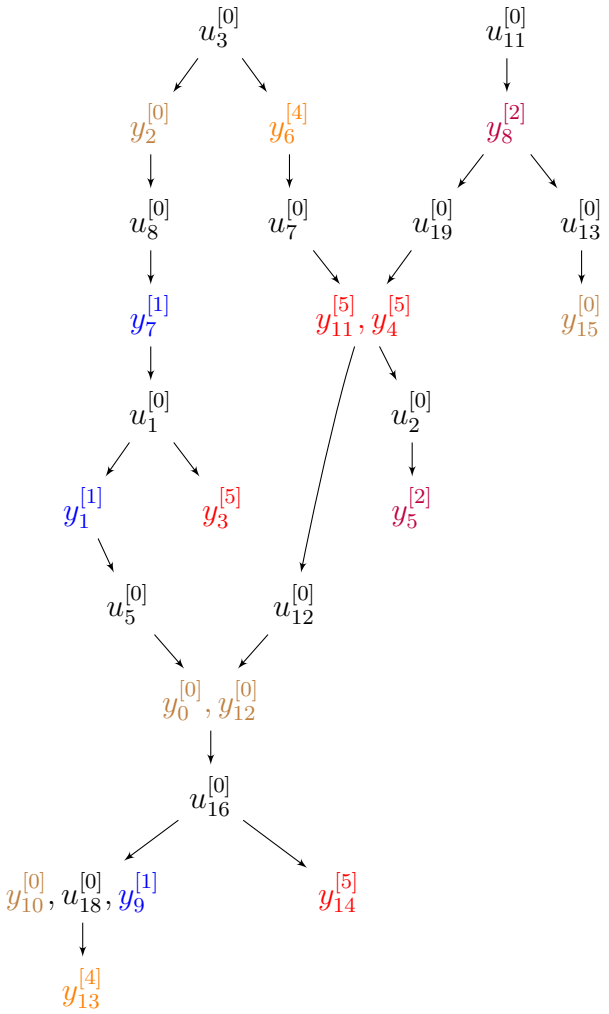


Figure 4.6: Result after the first iteration of the while-loop in Algorithm 3 where the vertices with the highest precedence order has been joined.

The output which satisfies both our conditions is $y_0^{[0]}$. The next step is to choose which other output it should be grouped with as there are two options, $y_0^{[12]}$ and $y_0^{[15]}$. Since $y_0^{[12]}$ has a higher level than $y_0^{[15]}$, this will be our pick in this iteration, cf. Figure 4.6.

After completing the iterative procedure, we are done with Algorithm 3. The final result is shown in Figure 4.7.

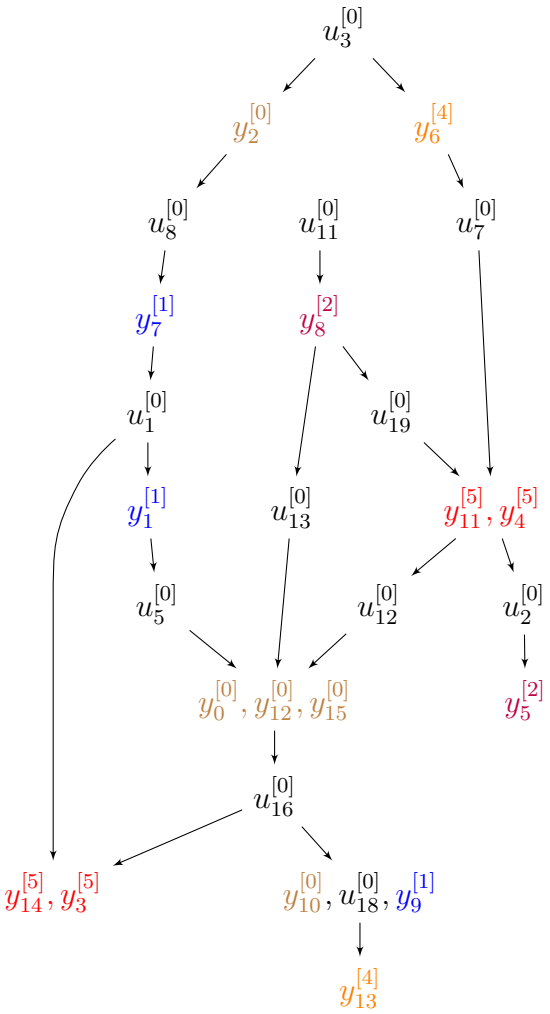


Figure 4.7: Resulting graph after Algorithm 3 has been executed. As many as possible of the outputs from the different models has been joined.

Race Car

In this example, the model describes a race car which has been provided from the commercial Vehicle Dynamics Library supplied by Modelon AB, cf. Figure 4.8. The race car is modeled together with a driver that tries to maneuver the vehicle in a figure eight shaped path while increasing the velocity. Simulations of this kind provides useful information of the dynamic response of the vehicle. This information can then in turn be used to optimize the lap time by calibrating the vehicle design [2].

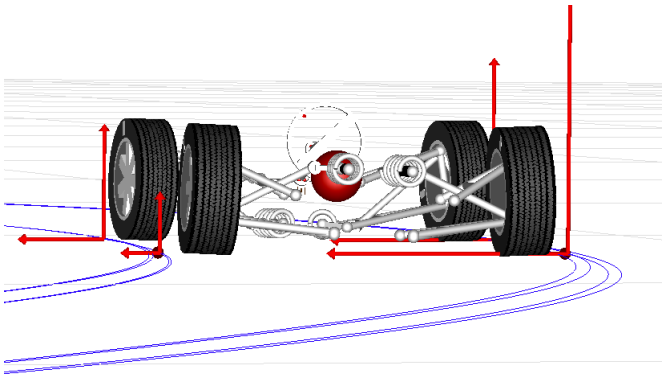


Figure 4.8: Visualization of the race car from Section 4.3 [2] [3] © Modelon

The coupled system consists of five models, the chassis and the four wheels, which are coupled together by 172 connections, cf. Figure 4.9. Additionally there is direct feed-through in the wheels. Considering the outputs of the system which are direct feed-through, we end up with one output for the chassis and the torques t_i and forces $f_i, i = 1 : 4$ gives us 6 outputs for each wheel in total. The reason the torques and forces are vector valued is due to the fact that the couplings are spatial, i.e. we have an output for each spatial dimension. This gives us that the minimal number of model function evaluations is 5 and the maximum is 25 [2] [3].

By deploying Algorithm 3 on the coupled system, we find that there are no none-trivial SSCs and thus no algebraic loops. Executing Algorithm 3 on the coupled system gives the optimal evaluation order [2].

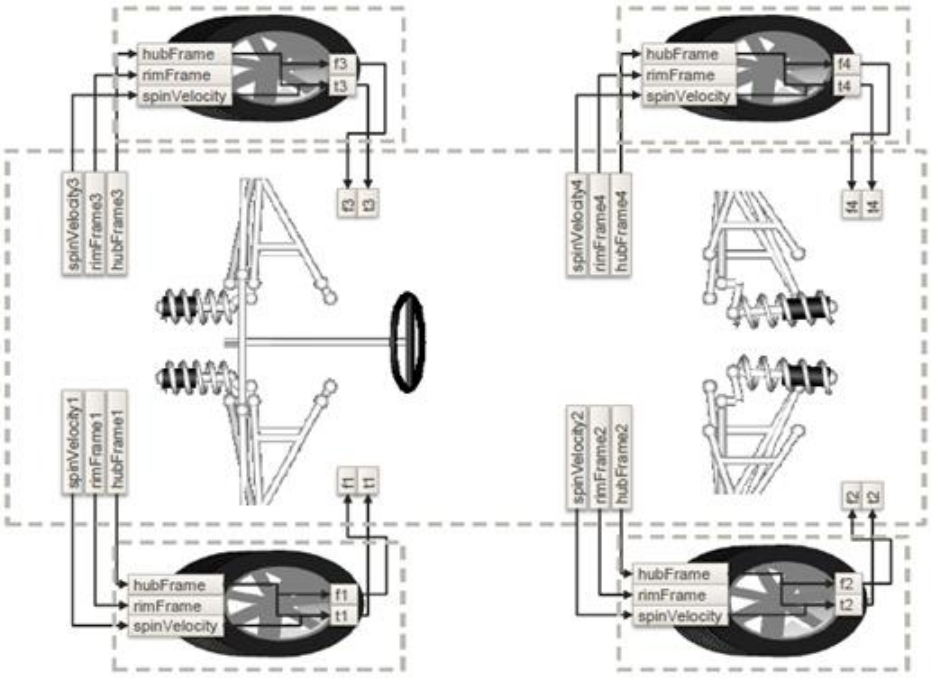


Figure 4.9: Overview of the couplings between the wheels and chassis of the race car from Section 4.3. Shown in the figure is the direct feed-through in the wheels between the hubFrame and spinVelocity with t[1-4] and f[1-4]. Note that the connections are vector valued [2] [3]. © Modelon

Chapter 5

Parallelization of Model Function Evaluations

In the previous chapters we have seen a new algorithm for reducing model function evaluations, cf. Algorithm 3. The motivation behind reducing model function evaluations lies in the assumption that computing the internal dynamics of a component is computationally expensive and by reducing the number of such evaluations, the computation time is lowered. With the ongoing trend of parallel computing in the computer and software industry, multiple computer processor (CPU) cores are common enough in standard consumer computers that designing software for parallel computing can often be a reasonable method of achieving lower computation time. This however depends on a number of things such as the size of the fraction of a program that can be parallelized and if the penalty of spawning new processes is sufficiently small to justify a parallel execution instead of a sequential one. Since this thesis ultimately revolves around speeding up the initialization phase, a parallel method of initializing coupled systems is considered.

This chapter introduces the general concepts of parallel programming, both the static case of load balancing as well as the dynamic case. Moreover, the model of the target parallel system is specified.

5.1 Basics of Parallel Computing

Concepts and Terminology

Parallel computing is the execution of a program using multiple CPUs concurrently instead of using one processor exclusively, i.e. a *parallel program*.

In order to use multiple CPUs and orchestrate the execution in a desired way, a parallel algorithm is needed. A parallel algorithm consists of a collection of discrete section of computational work that is executed by a processor. We call such a section of computational work a *task*. After executing the parallel algorithm, the final result should be a composition of the output from the tasks. Graphs are a common way of formulating an abstract representation of a parallel program, more specific *task graphs* are used for this purpose.

Definition 5.1 *(Graph Model) ([7]). A program consists of two kinds of activity - computation and communication. The computation is associated with the vertices of a graph and the communication with its edges. A task can range from an atomic operation to compound statements such as loops, basic blocks and sequences of these. All instructions or operations of one task are executed in sequential order, i.e. there is no parallelism within a task. A vertex is at any time involved in either computation or communication.*

Definition 5.2 *(Task Graph) ([7]). A task graph is a directed acyclic graph $\mathcal{G}_T(\mathcal{V}, \mathcal{E}, w, c)$ representing a program P according to the graph model of Definition 5.1. The vertices in $\mathcal{V}$ represent the tasks of P and the edges in $\mathcal{E}$ the communications between the tasks. An edge $(v, w) \in \mathcal{E}$ from vertex v to w , $v, w \in \mathcal{V}$, represents the communication from vertex v to vertex w. The non-negative weight $w(v)$, $v \in \mathcal{V}$ represents its computation cost and the non-negative weight $c((v, w))$, $(v, w) \in \mathcal{E}$ represents its communication cost.*

The process of designing a parallel algorithm is in general non-trivial and there are several components that need to be taking in consideration, e.g. hardware architecture, problem decomposition, scalability and overhead costs. In this thesis, only the necessary parts are introduced. For a more thorough explanation on parallel computing, cf. [6] [7]. The process of designing a parallel algorithm can in general be divided into three main parts:

- problem decomposition,
- granularity,
- mapping.

These parts will be given a more detailed explanation further into this chapter.

One of the most straightforward ways of evaluating the performance of a parallel algorithm is in terms of *speedup*.

Definition 5.3 *(Speedup) ([6]). Let n be the size of the input for a given program and p the number of processors which will be used in parallel. We say that $T(n, 1)$ is the run-time of the fastest known sequential algorithm and $T(n, p)$ the run-time of the parallel algorithm executed on p processors for the input size n. The speedup is then defined as*

$$S(n, p) = \frac{T(n, 1)}{T(n, p)}. \tag{5.1}$$

Ideally, one would like to have the speedup factor to equal the number of processors, i.e. $S(n, p) = p$, although this is rarely achieved in practice. We call this *perfect speedup* [6].

A way of estimating the potential speedup of a parallel program is by using *Amdahl's law* which can be formulated as,

$$S_A(n, p) = \frac{1}{(1 - r) + \frac{r}{p}}, \tag{5.2}$$

where S_A is the theoretical speedup of the execution of the whole task, r is the fraction of the code that can be parallelized and p is the number of processors used with a fixed input size n. [6].

Example 5.1 *(Amdahl's law). This example is intended to illustrate Amdahl's law by showing how the speedup is affected by the number of processors and the size of the parallel portion of the program, cf. Table 5.1.*

p	$r = 0.50$	$r = 0.90$	$r = 0.95$	$r = 0.99$
10	1.82	5.26	6.89	9.17
100	1.98	9.17	16.80	50.25
1000	1.99	9.91	19.62	90.99
10000	1.99	9.91	19.96	99.02
100000	1.99	9.99	19.99	99.90

Table 5.1: The theoretical speedup given from Amdahl's law for different r and p.

Problem Decomposition

One of the first steps into designing a parallel algorithm is to divide the computational problem into smaller subproblems which can be computed

in parallel, i.e. tasks. This is known as the process of *problem decomposition* or *partitioning*. There are many ways of decomposing a problem where some of the most common and well-known methods include *functional decomposition* and *data decomposition*.

Functional Decomposition

In the functional decomposition approach to decomposition, the initial focus is on the computations performed on the data rather than on the data itself. The aim is to find a way of separating these computations into parts, called functions, which can can be distributed to multiple processors for simultaneous execution. The next step is to analyze what data is needed to execute these functions. If the data requirements are disjoint, i.e. the functions perform computation on different data, the partition is done. If on the other hand there is a significant overlap on the data requirements for the functions, one needs to take special considerations to avoid replication of data [6].

Data Decomposition

When considering data decomposition, the focus is to first decompose the data associated with a problem. The data is divided into a number of equal parts where each part is typically associated with a certain type of operation. The decomposed data may be the input to the program, the output computed by the program, or intermediate values. Data decomposition is the most common approach to decomposing a problem for parallel computations and is typically used for problems which are associated with large amounts of data [6].

Granularity of Computation

In parallel computing, *granularity* of a task is a qualitative measure of the amount of the computational work which is performed by that task. The problem decomposition has an affect on the granularity as it defines the tasks. We say that a problem decomposition which yields many small tasks has fine granularity and the computation is fine-grained. Likewise in the opposite case, when the problem decomposition yields a small set of larger tasks, we say that the granularity is coarse and the computation is coarse-grained [6].

A concept related to granularity is that of *degree of concurrency* which is the number of tasks that can be executed simultaneously. Evaluating the

degree of concurrency only on account of the number of tasks involved in the parallel program is approximate. More accurate estimates are yielded by considering the granularity of the program, in which case we are interested in the *average degree of concurrency*. As the name indicates, this is the average number of tasks that can be executed simultaneously during the entire execution. Informally, the average degree of concurrency tells us how many operations, on average, can be carried out simultaneously in every step of computation [6].

In parallel computing, the *critical path* is the path which corresponds to the longest series of sequential operations in a parallel computation and thus also the theoretical minimal completion time of the program, cf. Definition 5.4. For this reason, minimizing this path is an important aspect when designing a parallel algorithm.

Definition 5.4 *(Critical Path Length). Given a task graph $\mathcal{G}_T(\mathcal{E}, \mathcal{V}, w)$, the critical path length $C_\mathcal{G}$ is the path which is associated with the largest sum of the vertex weights $w_i \in w$.*

Definition 5.5 *(Average Degree of Concurrency). Given a task graph $\mathcal{G}_T(\mathcal{E}, \mathcal{V}, w)$, the average degree of concurrency d_{avg} is defined as*

$$d_{avg} = \frac{\sum_{i=0}^{N} w_i}{C_\mathcal{G}}, \quad N = |\mathcal{V}|.$$

This corresponds to the average number of tasks that can be executed in parallel.

Example 5.2 *(Degree of Concurrency in Task Graph). In this example, the graph shown in Figure 5.1 has a critical path length $C_\mathcal{G} = 6+4+1 = 11$ and a weight sum $\sum_{i=0}^{N} w_i = 2+4+3+6+5+4+1 = 25$. The average degree of concurrency is thus $d_{avg} = \frac{25}{11}$.*

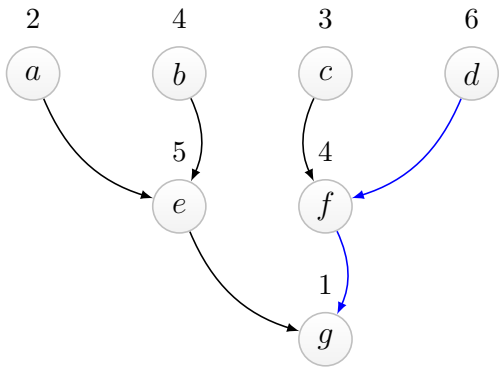


Figure 5.1: A directed weighted graph with its critical path colored in blue.

In general, a high degree of concurrency is related to a fine-grained decomposition. One could get the impression that a problem should be decomposed to be as fine-grained as possible in order to optimize the speedup. However, taking communication costs into consideration shows that the matter is more complicated. These costs commonly grow with the number of communicating tasks, which means that the degree of concurrency and granularity needs a thoughtful analysis in order to determine a proper balance between the two [6].

Definition 5.6 *(Computation and Communication Costs) ([7]). Let* $\mathcal{G}_T(\mathcal{V}, \mathcal{E}, w, c)$ *be a task graph representing a program* P *according to Definition 5.1. The vertex* $v \in \mathcal{V}$ *has an associated non-negative weight* w_v *which represents the computation cost of* v*. Similarly, the edge* $(v, w) \in \mathcal{E}$ *has an associated non-negative weight* $w_{(v,w)}$ *which represents communication cost of* (v, w)*. These costs typically correspond to the time a computation or communication takes on the specified target system.*

In the target parallel system considered in this thesis, computation costs are assumed to be uniform and communication costs are assumed to be negligible, cf. Assumption 5.1.

Assumption 5.1 *Computation costs, i.e. model function evaluations, are assumed to be uniform for all models and communication costs are assumed to be negligible.*

Assumption 5.2 *The number of available slave processors are assumed to be relative few, i.e. under 8. This motivates a coarse grained decomposition.*

Mapping

The next step of designing a parallel algorithm is to determine a systematic way of assigning tasks to processors involved in the parallel program. As for any parallel program, the goal is to maximize the speedup which is done by minimizing the execution time of the parallel program. A crucial aspect in order to achieve this is to have a task-to-processor mapping such that the time in which processors are not doing any computational work is minimized. This is also known as *load balancing* and the concept can roughly be divided into two paradigms, *static balancing* and *dynamic balancing* [6].

Static Balancing

Static load balancing is done before the execution of the parallel program. In order to determine how to assign the tasks to specific processors, several factors need to be taken in consideration. Such factors include the computational cost of each task, identification of data dependencies and frequency and latency for inter-process communication among others. These factors are in general difficult to measure and finding the optimum assignment is an NP-hard problem. The assignment is usually done with fast-acting heuristic algorithms which give a near-optimal assignment if designed properly [6] [7].

Dynamic Balancing

In contrast to the static case, in dynamic load balancing tasks are assigned to processors during the execution of a parallel program. If important information of hardware architecture and the software is dynamic or unknown, dynamic load balancing is the preferred strategy since the load is more uniformly distributed among the processors in this setting compared to a static strategy. There are two main methods of dynamic load balancing, centralized and decentralized. The latter will not be explained in this thesis [6].

In the centralized method the tasks are stored in a central data structure called work pool. By using a master processor, the tasks can be assigned to the slave processors. The idea is that whenever a slave processor completes execution of a task, it sends a request to the master processor to label the task as complete and then assign the next available task to the slave processor at hand. The centralized load balancing method is a suitable choice of method when there are few available slave processors and the granularity is coarse. In the opposite case, i.e. when we have many slave processors combined with fine-grained tasks, the master processor needs to handle a large number of requests which has a negative effect on performance [6].

5.2 Outline for Parallel Algorithm

Given Assumption 5.1 and 5.2, we have a rather coarse-grained decomposition where the communication costs are negligible. A parallel algorithm following a centralized method of dynamic mapping, i.e. a Master-Slave-pattern, is considered in this thesis.

Restriction 2.1 puts a difficult constraint on the parallel design since we are not allowed to send and retrieve FMUs between processes, i.e. each

FMU instance is locked to a given process. Although the actual initialization process would benefit greatly with a work-around by loading the same FMU models into several processes, it is not clear that this solution yields a better performance when also taking instantiation into account. Loading an FMU into memory is in general expensive relative to the initialization time and it is not desirable to repeat this process several times. The parallel algorithm considered in this thesis locks each model to a given process.

The basic outline of the algorithm that solves the initialization problem in parallel is as follows. The algorithm takes a task graph $\mathcal{G}_T$ as input where each vertex represent some task that we want to execute. In this particular setting, the tasks, i.e. the vertices of $\mathcal{G}_T$ will either be to set an input u_j or to compute an output y_i of a given model, where the latter entails a model function evaluation, cf. Definition 4.1.

A number of processes are spawned and assigned an available task to be executed. More precisely, the task is chosen such that there are no predecessors that have not already finished execution so that the dependency constraints are respected. In this stage, the algorithm tries to execute as many tasks as possible. If all processes have a task assigned or if there is no available task to run while respecting the dependency constraints, the algorithm enters an inner while-loop until a task has been finished. The inner while-loop checks if any process has sent a message to the master declaring that it has finished executing a task, in which case the task is removed from the task graph $\mathcal{G}_T$ and the master tries to assign a new task to the newly available process. This procedure is then repeated until all tasks have been executed. The proposed algorithm is shown in Algorithm 4.

Algorithm 4 Parallel algorithm

Require: A directed graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$.
Require: Information about which nodes belong to same model, M.
Require: Information about which nodes are outputs.

```
1: while dim(finished_tasks) ≠ dim(V) do
2:     variable ← None
3:     model ← None
4:     no_tasks ← False
5:     for v ∈ T do
6:         if variable.dependencies = ∅ then
7:             variable ← v
8:             model ← model_map[v]
9:         end if
10:    end for
11:    if variable ≠ None then
12:        slave ← worker_map[model]
13:        if ¬ slave.busy then
14:            SET_OR_GET_VARIABLE(variable)
15:            running_tasks.append(task_name)
16:        else:
17:            no_tasks ← True
18:        end if
19:    else
20:        no_tasks ← True
21:    end if
22:    if no_tasks or dim(running_tasks) = dim(worker_list) then
23:        task_done ← False
24:        while ¬ task_done do
25:            for slave ∈ worker_list do
26:                if slave.busy and HAS_PENDING_MESSAGE(slave) then
27:                    message, data ← GET_MESSAGE(slave)
28:                    if message = "FINISHED" then
29:                        variable ← data
30:                        finished_tasks.append(variable)
31:                        running_tasks.remove(variable)
32:                        slave.busy ← False
33:                        task_done ← True
34:                    end if
35:                end if
36:            end for
37:        end while
38:    end if
39: end while
```

5.3 Case Studies

Synthetic Test Case

In this example, seven models with feed-through are connected and we consider multiple processors for solving the initialization problem by deploying Algorithm 4. A graph of the coupling is shown in Figure 5.2. Since we have seven models, the number of processors required for the initialization is eight due to Restriction 2.1, i.e. seven slave processors and one master processor.

The actual computations involved during the initialization are replaced with a sleep-function in order emulate the computational task. The function is defined as follows.

- Vertices which are inputs are assumed to take 0.01 seconds.
- Vertices which are outputs are assumed to take 1.0 seconds.
- Vertices which correspond to algebraic loops are assumed to take 5 seconds.

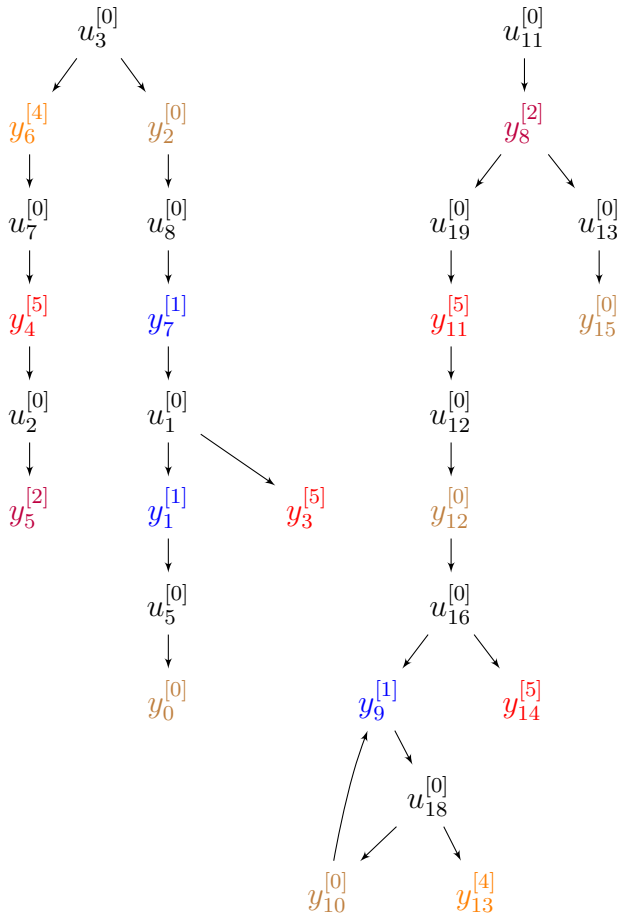


Figure 5.2: Graph showing seven coupled models.

Comparing the elapsed initialization time for both the sequential case and the parallel case we see that for the sequential case, the initialization takes about 20 seconds while for the parallel case it takes about 6.7 seconds, which gives a speedup factor of approximately 3. The evaluation order was computed with Algorithm 1 and thus the number of model function evaluations was not reduced.

Race Car

Revisiting the race car from Section 4.3 we now consider multiple processors for solving the initialization problem. As in the synthetic test case in Section 6.3, the system is initialized by assigning a process for each model and one for the master, i.e. six processors in this case. Comparing the global output vector y and global input vector u after initialization for both the

sequential case and the parallel case, we notice the results are identical. This means that the precedence constraints were respected and the system was initialized properly.

Comparing the elapsed initialization time for both the sequential case and the parallel case we see that for the sequential case, the initialization takes about 0.26 second while for the parallel case it takes about 0.056 seconds, which gives a speedup factor of approximately 4. In addition, the evaluation order was computed with Algorithm 3, which gives the minimal number of model function evaluations.

The system was also initialized without reducing the number of model function evaluations, i.e. by only using Algorithm 1 to compute the evaluation order. In this setting the sequential case takes about 1.0 seconds while the parallel case takes about 0.26 seconds and thus again we have a speedup factor of approximately 4.

Chapter 6

Priming for Parallelization with a Genetic Algorithm

When considering multiple processors working in parallel for solving the initialization problem, one should also take into account the average degree of concurrency of the graph, cf. Definition 5.4. This is a consequence of the fact that grouping output vertices affects the precedence constraints in the corresponding task graph and by extension, the graphs inherent capacity to make use of multiple processors. In essence, the objective of the method presented in this chapter is to achieve a better solution according to Definition 3.8 by combining the strategy of reducing model function evaluations as in Chapter 4 together with the concept of parallel initialization as in Chapter 5. The problem is then to find the correct balance between reducing models while still maintaining a high average degree of concurrency. We call this notion *priming for initialization*. In terms of combinatorial optimization, this results in an additional constraint on the set of candidate solutions.

For solving this problem, a *genetic algorithm* (GA) is developed and used. The general basic components and the problem specific components of the GA, along with the use of simple scheduling heuristics are explained in this chapter.

6.1 List Scheduling Heuristics

The dominant heuristic technique encountered in scheduling algorithms is the so-called list scheduling. In its simplest form, the first part of list scheduling sorts the vertices of the task graph to be scheduled according to a priority scheme, while respecting the precedence constraints, i.e. the

resulting vertex list is in topological ordering. In the second part, each vertex of the list is successively scheduled to a processor chosen for the vertex. The chosen processor is the one that allows the earliest start time of the vertex. This method is called *start minimization time*. Algorithm 5 outlines a simple form of list scheduling with the use of start minimization time [7].

Algorithm 5 Simple List Scheduling algorithm [7]

Require: A directed graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$.
1: {Sort vertices $n \in \mathcal{V}$ into list L, according to priority scheme and precedence constraints}
2: **for** $n \in \dim(L)$ **do**
3: SCHEDULE_NODE(n)
4: **end for**
5: **procedure** SCHEDULE_VERTEX(n)
Require: n is a free vertex
6: $t_{min} \leftarrow \infty$
7: $p_{min} \leftarrow$ `none`
8: **for** $p \in P$ **do**
9: **if** $t_{min} > max\{\text{TDR}(n,p), \text{TF}(p)\}$ **then**
10: $t_{min} \leftarrow max\{\text{TDR}(n,p), \text{TF}(p)\}$
11: $p_{min} \leftarrow p$
12: **end if**
13: **end for**
14: $\text{TS}(n) \leftarrow t_{min}$
15: $proc(n) \leftarrow p_{min}$
16: **end procedure**

Example 6.1 *(List Scheduling). In this example, we apply Algorithm 5 on the directed weighted graph shown in Figure 6.1. The resulting schedule is shown in Figure 6.2 and we get a completion time of 24.*

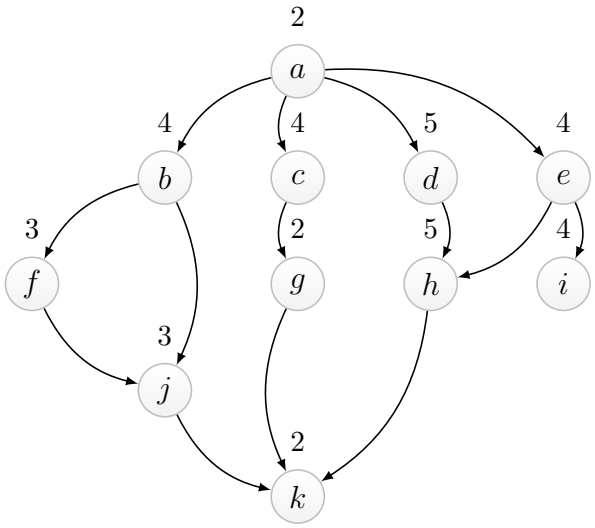


Figure 6.1: A directed weighted graph.

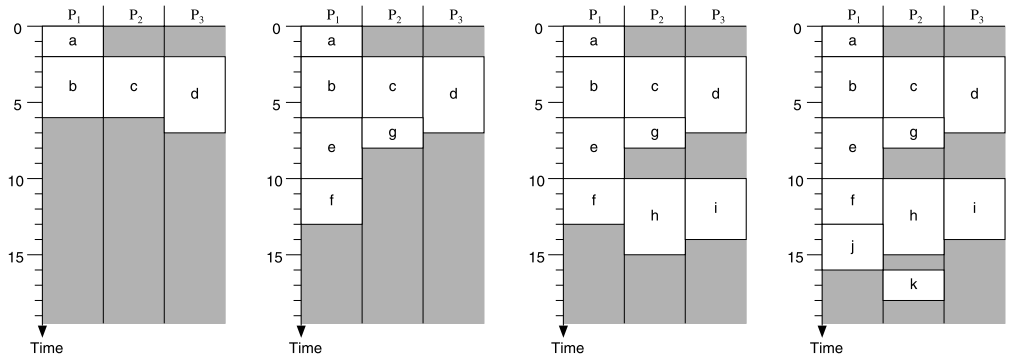


Figure 6.2: Example of simple list scheduling using the graph from Figure 6.1: (a) to (c) are snapshots of partial schedules; (d) shows the final schedule. The node order is [a,b,c,d,e,f,g,i,h,j,k].

6.2 Basic Concepts of Genetic Programming

Complex problems are sometimes solved with the use of randomized search methods like genetic algorithms, swarm algorithms, simulated annealing etc. The problem of priming a graph such that it is better suited for parallel computing is a complicated task since one needs to develop a good strategy for balancing the trade-off between reducing model function evaluations and maintaining a high average degree of concurrency in the graph structure.

Moreover, models can vary in complexity and size which in turn affects the trade off, i.e. such a strategy also needs to be flexible to be able to handle different types of systems models. In this thesis, the problem of priming a graph was solved by the use of a GA method.

A GA is a search algorithm that is based on the principles of evolution and natural selection. A set of candidate solutions called the *population*, creates new generations by two operators, *crossover* and *mutation*. The method is constructed such that the better candidate solutions in the population are more likely to be selected and thus transmitting their inheritance to the next generation [7]. This section studies how a GA can be applied to the priming problem. As the area of genetic algorithms is very broad, only the necessary concepts are introduced. For a thorough introduction on genetic programming, cf. [7].

Algorithm 6 outlines a simple GA and the fundamental components are described in the following.

- Chromosome: The generated candidate solutions to the problem are called chromosomes. These are typically encoded as a problem specific binary number or list [7]. In this setting, the chromosomes are encoded by traversing the graph and storing all pairwise possible vertex groupings in a list `all_groupings`. A chromosome will then consist of a binary list where the elements of the list, called genes, indicate if two SCCs on the corresponding index in `all_groupings` should group to form one combined SCC or not, cf. Example 6.2. In essence, each chromosome is a variation of the original initialization graph where the possible groupings of the vertices are explored, cf. Algorithm 2 and 3 which tries to group as many SCCs as possible.

- Population: A collection of chromosomes which make up all current chromosomes is called a population. The initial population is generated randomly whereas the subsequent generations are generated from the GA [7]. In this setting, the population will consist of several chromosomes of the type described above.

- Evaluation: The evaluation stage is required in order to determine the quality of each chromosome which is done with a fitness function [7]. The fitness function in this setting is to construct a graph according to the chromosome i.e. which vertices should form SCCs and then compute an approximation, i.e. a schedule, for how long it would take to initialize the system with a structural approach by using the current graph representation of the system on a given number

of processors. This time length is then the assigned fitness score. The motivation behind this is that we are interested in finding a graph which best favors parallelism for a given number of processors. The fitness of a chromosome is directly related to the length of the associated schedule. Computing a schedule and its length is a fast way of approximating the time it would take to solve the problem. The schedule and its length is calculated by deploying Algorithm 5.

- Selection: Selection is the stage of the GA in which the chromosomes are chosen from the population based on their fitness score.

- Crossover: Crossover is one of two operators used in the GA to vary the programming of the chromosomes. The operator functions by combining two existing chromosomes to form a new [7]. In this setting, we take two chromosomes and simply split them in two respectively at some randomly chosen index and then merge the sublists from the different chromosomes to form two new chromosomes.

- Mutation: The mutation operator acts on a chromosome by randomly changing a small portion of the encoding. The purpose of the mutation operator is avoid convergence at a local point by slightly perturbing the population [7]. In this setting, the mutation simply takes a random chromosome and then swaps a random gene to the opposite value, i.e. from 1 to 0 or 0 to 1.

After a specified number of iterations has been completed, the GA terminates and returns the chromosome with the best fitness.

Algorithm 6 Priming GA

Require: A directed graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$.
Require: Population size N_{pop}
Require: Generation size N_{gen}
1: POPULATE() {Create initial population}
2: EVALUATION() {Calculate fitness of initial population}
3: **for** $i \in$ RANGE(N_{gen}) **do**
4: SELECTION()
5: CROSSOVER()
6: MUTATION()
7: EVALUATION()
8: **end for**

Algorithm 6 Priming GA (continued)

```
9:  procedure POPULATE
10:     all_groupings ← COMPUTE_POSSIBLE_GROUPINGS(𝒢)
11:     for i ∈ RANGE(N_pop) do
12:         population[i] ← CREATE_RANDOM_BINARY_LIST() {Create a random binary list with same length as possible_groupings}
13:     end for
14: end procedure
```

Algorithm 6 Priming GA (continued)

```
15: procedure EVALUATION
16:     for chromosome ∈ population do
17:         𝒢*(𝒱, ℰ) ← 𝒢(𝒱, ℰ)
18:         to_group ← ∅
19:         for gene ∈ chromosome do
20:             if gene = 1 then
21:                 scc ← CREATE_SCC(gene) {Create new strongly connected component}
22:                 to_group.append(scc)
23:             end if
24:             𝒢*(𝒱, ℰ) ← CONSTRUCT_GRAPH(to_group) {Create new digraph with specific SCCs}
25:         end for
26:         score[chromosome] ← dim(LIST_SCHEDULE(𝒢*))
27:     end for
28: end procedure
```

Example 6.2 *(Chomosome Encoding). In this example we have a graph pictured in Figure 6.3. Traversing the graph to get all possible SCC groupings, we form the list* `all_groupings` $= [(y_1^{[2]}, y_2^{[2]}), (y_1^{[1]}, y_2^{[1]}), (y_2^{[1]}, y_3^{[1]})]$. *A chromosome of the graph can then be constructed by either grouping these vertices into a SCC or not depending on the value of the chromosome gene. This is indicated by a 1 or 0, e.g. the chromosome* $c_1 = [1, 1, 0]$ *means that* $(y_1^{[2]}, y_2^{[2]})$ *and* $(y_1^{[1]}, y_2^{[1]})$ *should merge while* $(y_2^{[1]}, y_3^{[1]})$ *should not.*

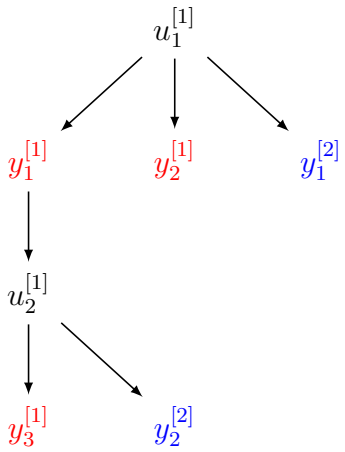


Figure 6.3: A graph of a coupled system with two models.

6.3 Case Studies

Synthetic Test Case

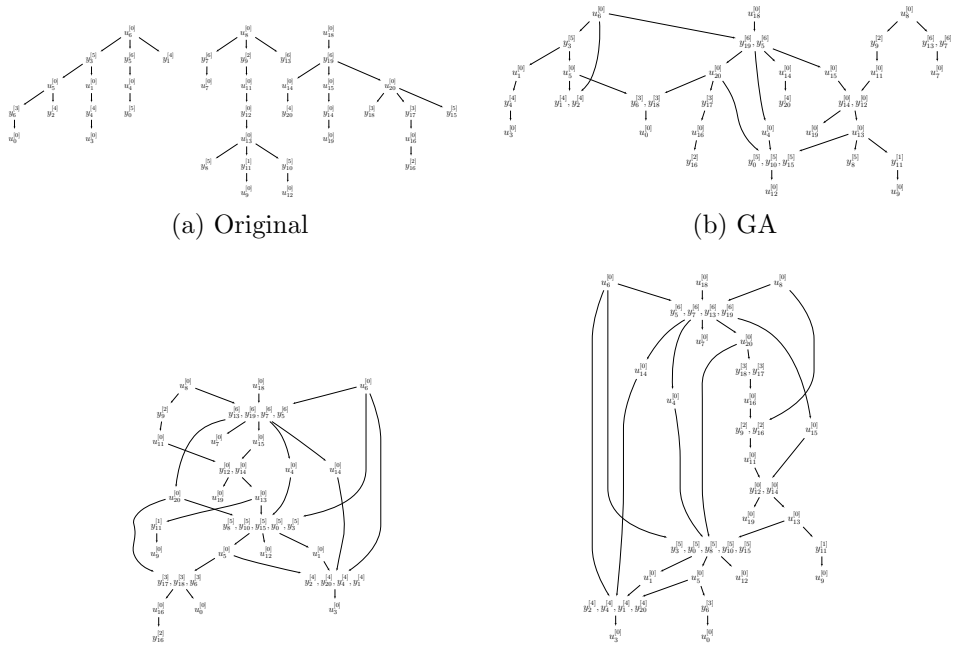
In this example, seven models with feed-through are connected where we consider multiple processors and the structure of the graph for solving the initialization problem by deploying Algorithm 6 and Algorithm 4.

The actual computations involved during the initialization are replaced with a sleep-function in order to emulate the computational time of the FMI-functions `fmi2SetXXX` and `fmi2GetXXX`. The function is defined as follows.

- Vertices which are inputs are assumed to take 0.1 seconds.
- Vertices which are outputs are assumed to take 2.5 seconds.
- Vertices which correspond to algebraic loops are assumed to take 5 seconds.

Algorithm	Reduced Evalutations	Completion Time [s]	Speedup
Algorithm 1	n/a	21.9	2.5
Algorithm 2	13	16.87	3.2
Algorithm 3	13	14.5	3.7
GA	7	13.78	3.9

Table 6.1: Completion time using list scheduling with seven slave processors and one master processor and neglecting communication costs.



(a) Original (b) GA

(c) Alg. 3 (d) Alg. 2

Figure 6.4: Structural comparison of different transformations of the same graph.

In Figure 6.4 we can see the structural differences depending on which algorithm that was used to reduce the number of model function evaluations. In Table 6.1 we notice that the GA outperforms the other algorithms even though it reduced fewer model function evaluations. An important point to be made here is that depending on the assumptions made about the evaluation costs, the GA should adapt accordingly.

Chapter 7

Algorithm Implementation

In this chapter, a more detailed description of the algorithm implementations within the open-source project PyFMI is given. The algorithms have previously been outlined in pseudo code, cf. Algorithm 3, 4 and 6, in Chapter 4 5 and 6 respectively, but without any implementation details such as a description of the data structures, classes and the integration within PyFMI. Although such a description is given here, the algorithms are just one part of chain of operations. Therefore, only the parts essential to the algorithms are presented here.

7.1 Model Reduction Algorithm

Classes and Data Structures

The two main classes from PyFMI used for this thesis are the following,

- `Master` - This class contains the master algorithm that is responsible for orchestrating the computations of the subsystem and sets up the coupled system for all stages of the simulation.
- `Graph` - This class contains the graph representation of the system and methods to compute the SCCs along with an evaluation order. The algorithm for reducing model function evaluations, i.e. Algorithm 2, is also a part of this class as method `compute_evaluation_order`.

The `Master` instantiates an instance of the `Graph` class by first constructing the necessary information and then pass the information as an input argument. The graph is represented by a list of tuple pairs, where the first element of a tuple is the source vertex and the second element is the destination vertex. Adding Algorithm 3 to PyFMI required only to add a new

method to the `Graph` class called `compute_evaluation_order` and then also adding the possibility to call this method form the `Master` class.

7.2 Parallel Algorithm

The parallel method for initialization of coupled systems was implemented within a master-slave pattern, where the master is represented by a class `MasterParallel` and the slave is represented by a class `Slave`.

Serialization

An important aspect for understanding the implementation of the parallel algorithm is the concept of *serializing*, also known as *pickling*. Serializing is the process of converting structured data into a byte stream such that it can be stored and reconstructed later in the same or another computer environment. The inverse of this process, *deserializing* or *unpickling*, is the process of converting a byte stream into structured data. The built-in serialization module for Python is called `Pickle` [9]. Since `Pickle` does not have support for serialization of FMUs, a work-around had to be implemented. The work-around involved loading the FMUs into specific processes such that each process is locked to a specific FMU.

Python Multiprocessing Module

The parallel framework that was used for this thesis is the native multiprocessing module for Python. The module is a package that supports spawning processes and effectively side-stepping the *Global Interpreter Lock* by using subprocesses. The processes are spawned by the use of the `Process` class where an instance of the class will represent an activity that is run in the newly spawned process [9].

The multiprocessing module also offers two methods of communication between processes, the `Queue` class and `Pipe` class. In this thesis, the communication between master and slave is handled by the `Pipe` class. The motivation behind this lies in the simplicity of the class which enables for fast communication and also due to the fact that the extra features of the `Queue` class are redundant [9].

The `Pipe()` function returns a pair of `connection` objects connected by a pipe which by default is duplex. The two `connection` objects returned by `Pipe()` represent the two ends of the pipe. Each `connection` object

has methods to send and receive messages under the condition that the messages are represented by pickable objects [9].

Classes and Data Structures

The implementation consists of 3 main parts,

- `MasterParallel` - This class contains the Master parallel algorithm which orchestrates the parallel computations by communicating with the slaves and delegating tasks.
- `Slave` - This class is an abstract representation of an instantiated FMU along with other methods and attributes that are important.
- `Run` - This is not a class but an helper function called by `Slave` via the `Process` class. The new processes execute `Run` and communicate with the main process by sending piped messages back and forth to `Slave`, which in turn sends the message to the `MasterParallel`.

The slave acts as a wrapper for a FMU and stores an instance of the Python `Process` class by spawning a new process. The newly created process will load a FMU and communicate with the main process in which the slave is located, e.g. typing `slave.get_real()` will trigger a message to the subprocess to call the corresponding FMI-function and then send the value back to the slave. The master's responsibility is then to orchestrate the triggering of such function calls and ensure the slaves execute the tasks according to the dependency constraints, cf. Figure 7.1.

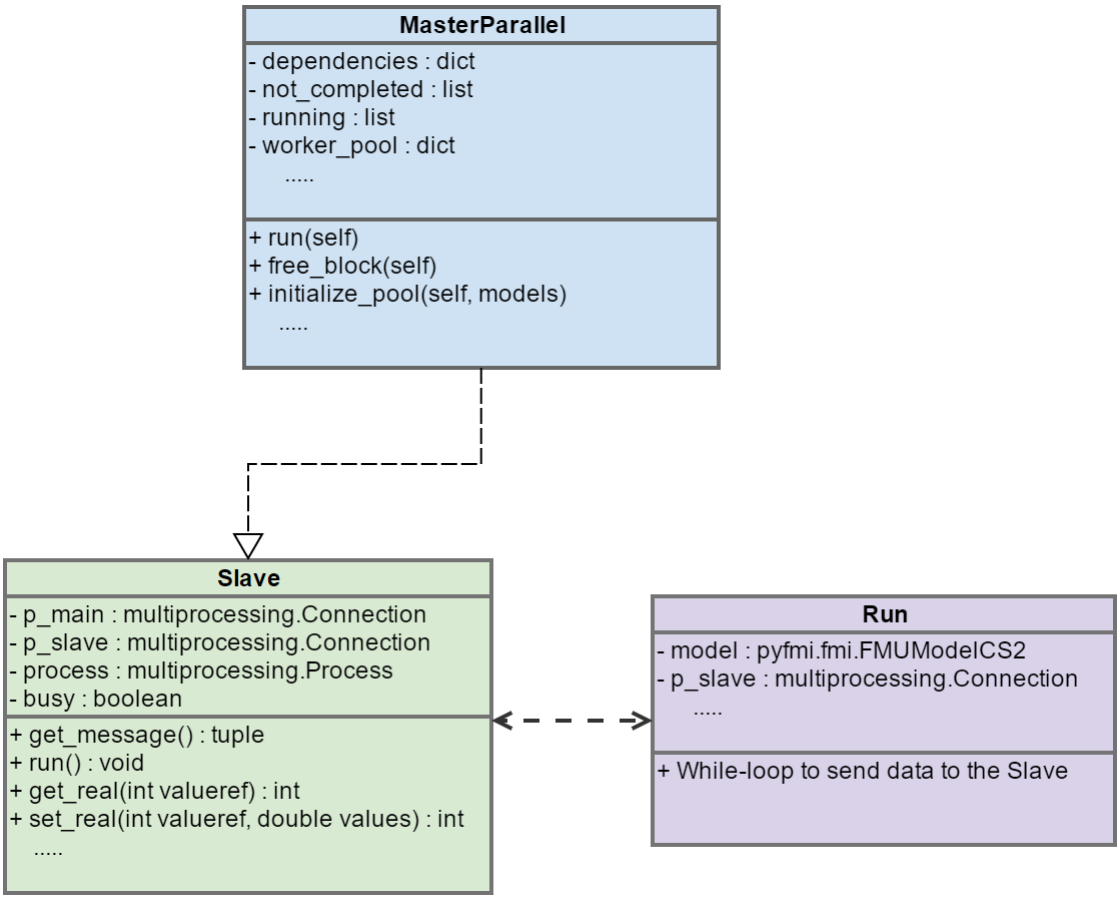


Figure 7.1: UML diagram of the Python classes used for parallel model function evaluations.

7.3 Genetic Algorithm

Classes and Data Structures

The implementation consists of 2 main parts,

- `GeneticAlgorithm` - This class contains the GA and the components of the GA as methods.
- `ListScheduler` - This class is used in the evaluation of the GA by constructing the schedule of a specific chromosome and evaluating the length of the schedule.

Chapter 8

Results and Benchmark

The main purpose of this thesis has been developing different algorithms which manages to speedup the initialization. We have already presented some performance results in the qualitative test cases in Section 4.3, 5.3 and 6.3. In this chapter, the performance of the algorithms are instead tested on several aspects by generating a large number of random graphs, cf. Section 4.1. The results are compared with Algorithm 1 and 2 and in some instances there is also a comparison with the optimal solution which has been computed with an exhaustive search method, cf. Section 4.2.

An important remark regarding Algorithm 2 is that due to how the graphs are generated, the algorithm will be unfairly penalized and produce results which are misleading in some test cases. To counter this problem, a small correction has been added by running the procedure `compute_possible_groupings` from Algorithm 3 in Chapter 4. By doing so, any potential reduction which has been left out is also reduced. The algorithm including the correction is called Algorithm 2.1.

8.1 Model Reduction Algorithm

In this section, the performance and results of Algorithm 3 are presented. The main objective of Algorithm 3 is to speed up the initialization by reducing the number of model function evaluations. The algorithm is tested on both how well it reduces the number of model function evaluation as well as how the execution time scales up for larger graphs. The graphs that are used to test the algorithm are randomly generated where a number of parameters can be specified to control the generation.

Reducing Model Function Evaluations

In Table 8.1 and 8.2 we compare the different algorithms with each other on how well they reduce the number of model function evaluations where the optimal solutions have been computed with an exhaustive search method for reference. The number of graphs that were generated was $n = 100$ for both tables but with different parameter settings in the graph generation. In the second column we see the total number of model function evaluations that are in excess compared to the optimal. In the third column we see how many times the algorithm did not find an optimal solution.

Algorithm	No. Missed Model Evals	No. Missed Optimal Solutions
Alg. 2	21 of 274	18 of 100
Alg. 2.1	10 of 274	8 of 100
Alg. 3	2 of 274	2 of 100

Table 8.1: Performance of reducing number of model function evaluations with the parameter settings: number of disconnected components was set to 1, the number of inputs was set to 10, the number of outputs was set to 10 and the number of models was set to 5.

Algorithm	No. Missed Model Evals	No. Missed Optimal Solutions
Alg. 2	39 of 570	35 of 100
Alg. 2.1	24 of 570	20 of 100
Alg. 3	6 of 570	6 of 100

Table 8.2: Performance of reducing number of model function evaluations with the parameter settings: number of disconnected components was set to 2, the number of inputs was set to 12, the number of outputs was set to 16 and the number of models was set to 5.

Execution Time

In Figure 8.1 and 8.2 we compare the execution time of Algorithm 2 and 3 on graphs of increasing size, i.e. an increased number of vertices where the number of edges are kept proportional to the number of vertices. The number of outputs are set equal to the number of inputs. For a given graph size we take the average execution time out of three randomly generated instances, increase the size of the graph with a factor of 2 and repeat the procedure.

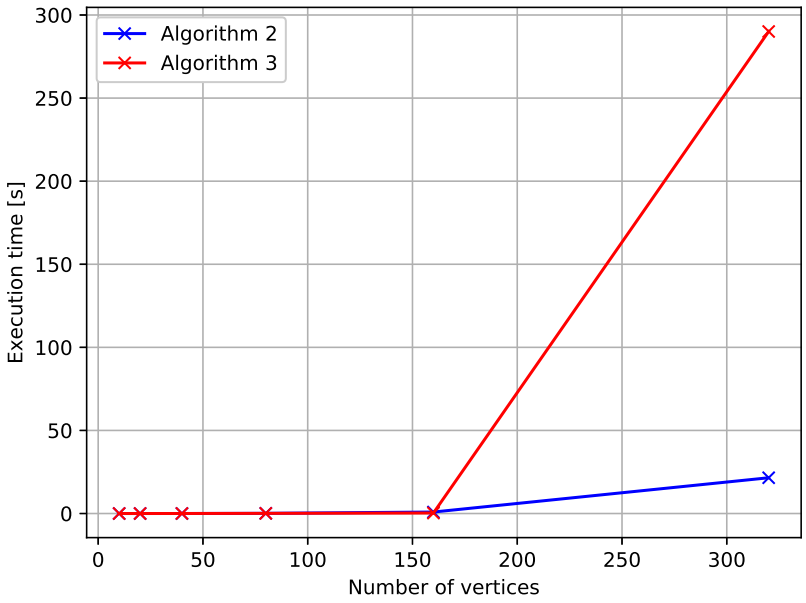


Figure 8.1: Execution times of Algorithm 2 and 3 for different graph sizes. The number of models is set to 5 and the number of disconnected components is set to 2.

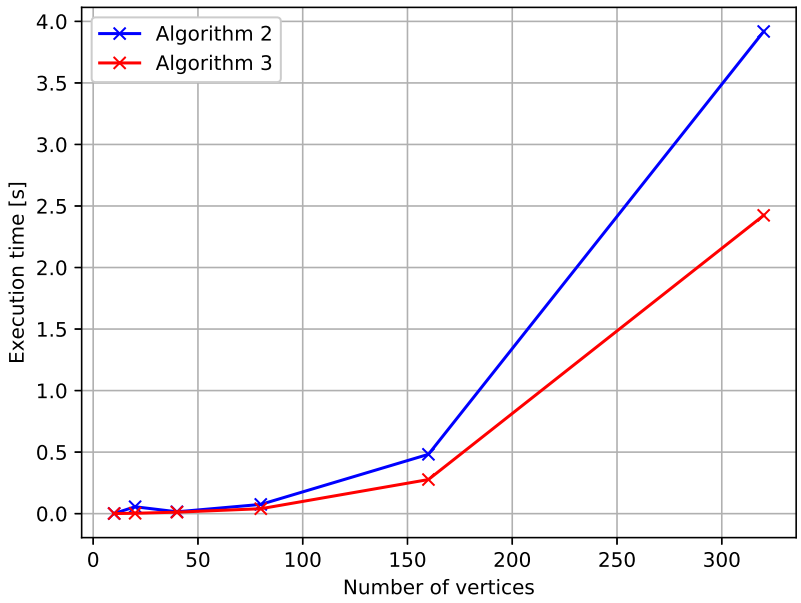


Figure 8.2: Execution times of Algorithm 2 and 3 for different graph sizes. The number of models is set to 5 and the number of disconnected components is increased with a factor of 2.

8.2 Parallel Algorithm

In this section, the performance and results of Algorithm 4 are presented. The main objective of Algorithm 4 is to speed up the initialization by making use of multiple processors to solve the initialization problem in parallel. The algorithm is tested on both instantiation of FMUs and initialization.

As in Section 8.1, the graphs that are used to test the algorithm are randomly generated where a number of parameters can be specified to control the generation. Since the graphs that are used for testing the initialization do not have corresponding real FMUs, a sleep-function is used to emulate the computational time of the FMI-functions `fmi2SetXXX` and `fmi2GetXXX`. The function is defined as follows:

- Vertices which are inputs are assumed to take 0.1 seconds.
- Vertices which are outputs are assumed to take 2.5 seconds.
- Vertices which correspond to algebraic loops are assumed to take 5 seconds.

Instantiation

In Table 8.3 we compare the time it takes to instantiate different systems in parallel and sequentially. Three different systems are used for this test, Race Car from Section 4.3, Controlled Temperature from Section 3.1 and 2 instances of the chassis model from Race Car. The number of processors working in parallel is equal to the number of FMUs for each system.

System	No. of FMUs	Instantiation [s]	Speedup
Controlled Temp.	2	0.8	0.7
Race Car	5	3.5	0.96
Race Car Chassis x 2	2	3.3	1.6

Table 8.3: Instantiation time for different systems.

Initialization

In Table 8.4, 8.5, 8.6 and 8.7 we compare the time it takes for the initialization. The number of graphs that were generated was $n = 100$ and the number of models was set to equal the number of slave processors for all four tables.

The graphs for Table 8.4, 8.5 and 8.6 were generated with the following the parameter settings: number of disconnected components was set to 3, the number of inputs was set to 18, the number of outputs was set to 24.

In Table 8.7 the parameter settings were: number of disconnected components was set to 3, the number of inputs was set to 18, the number of outputs was set to 24 and the number of models was set to equal the number of slave processors for each case.

The first column specifies if the graph was first prepared by merging vertices which belong to the same model, i.e. reducing the number of model function evaluations by deploying Algorithm 3. In the second column we see the average completion time for the initialization. In the third column we see the average speedup compared to the worst case, i.e. the sequential completion time for the graph where the number of model function evaluations has not been reduced.

Evaluations were reduced	Avg. Completion Time [s]	Avg. Speedup
No	35.6	1.4
Yes	13.7	1.3

Table 8.4: Average completion time of parallel initialization with 2 slave processors and 1 master.

Evaluations were reduced	Avg. Completion Time [s]	Avg. Speedup
No	26.0	1.9
Yes	14.8	3.3

Table 8.5: Average completion time of parallel initialization with 4 slave processors and 1 master.

Evaluations were reduced	Avg. Completion Time [s]	Avg. Speedup
No	18.9	2.6
Yes	15.2	3.2

Table 8.6: Average completion time of parallel initialization with 7 slave processors and 1 master.

Evaluations were reduced	Avg. Completion Time [s]	Avg. Speedup
No	42.8	2.3
Yes	21.3	4.6

Table 8.7: Average completion time of parallel initialization with 6 slave processors and 1 master.

Simulation

In order to ensure that the initialization is carried out properly, the system Race Car from Section 4.3 was initialized by first deploying Algorithm 3 to reduce the number of model function evaluations and then initialized in parallel by deploying Algorithm 4 The solution is compared with a reference solution which was computed from initializing the system using a sequential method.

Both solutions were computed using a relative and absolute tolerance set to 10^{-8}, step size $H = 0.005$ and an simulation time $t_f = 25.0$ s. The result of the simulations are shown in Figure 8.3.

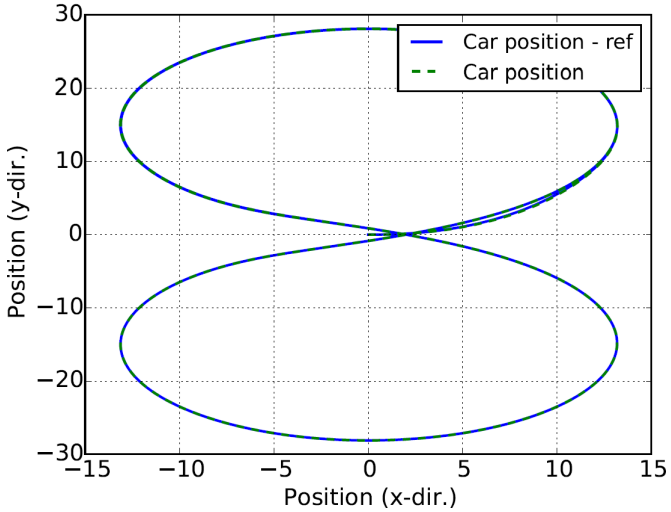


Figure 8.3: Simulation result for Race Car with $t_f = 25$ [s], a tolerance of 10^{-8} and step size H = 0.005.

8.3 Genetic Algorithm

In this section, the performance and results of Algorithm 6 are presented. The main objective of Algorithm 6 is to speed up the initialization by making use of both multiple processors as well as taking the graph structure in consideration.

In Table 8.8, 8.9 and 8.10 we compare the time it takes for the initialization. The number of graphs that were generated was $n = 100$ and the number of models was set to equal the number of slave processors for all three tables.

In the first column we see the average number of reduced model function evaluations, in the second column we see the average completion time for the initialization and in the third column we see the average speedup compared to the worst case, i.e. the sequential completion time for the graph where the number of model function evaluations has not been reduced.

As in Section 8.1, a sleep-function is used to emulate the computational time of the FMI-functions `fmi2SetXXX` and `fmi2GetXXX`. This time the function is defined as follows.

- Vertices which are inputs are assumed to take 0.1 seconds.
- Vertices which are outputs are assumed to take 1.0 seconds.

- Vertices which correspond to algebraic loops are assumed to take 5 seconds.

The GA has a multitude of parameters that can be adjusted. The two main parameters are the size of the population N_{pop} which was set to $N_{pop} = 250$ and the number of generations N_{gen} which was set to $N_{gen} = 50$.

Algorithm	Avg. Reduced Evals	Avg. Completion Time [s]	Speedup
Algorithm 1	n/a	8.8	1.8
Algorithm 2.1	6.4	5.8	2.6
Algorithm 3	5.9	5.1	2.8
GA	3.4	6.9	2.4

Table 8.8: Average completion time of parallel initialization with 4 slave processors and 1 master with the parameter settings: number of disconnected components was set to 3, the number of inputs was set to 21, the number of outputs was set to 15.

Algorithm	Avg. Reduced Evals	Avg. Completion Time [s]	Speedup
Algorithm 1	n/a	9.3	2.3
Algorithm 2.1	7.5	5.8	2.9
Algorithm 3	7.0	5.1	3.1
GA	3.7	7.3	3.0

Table 8.9: Average completion time of parallel initialization with 7 slave processors and 1 master with the parameter settings: number of disconnected components was set to 4, the number of inputs was set to 28, the number of outputs was set to 20.

Algorithm	Avg. Reduced Evals	Avg. Completion Time [s]	Speedup
Algorithm 1	n/a	7.7	2.4
Algorithm 2.1	8.9	4.9	3.8
Algorithm 3	8.9	4.5	4.1
GA	5.3	5.3	3.5

Table 8.10: Average completion time of parallel initialization with 7 slave processors and 1 master with the parameter settings: number of disconnected components was set to 7, the number of inputs was set to 21, the number of outputs was set to 21.

Chapter 9

Conclusions and Further Development

In this thesis, the problem of initializing weakly coupled dynamic systems has been studied. The main purpose of this thesis has been to speedup the computation time of the initialization. This has resulted in three different methods of achieving the speedup.

9.1 Reduction of Model Function Evaluations

In Chapter 4, a method for initializing coupled systems is presented in Algorithm 3. The method computes an evaluation order of the inputs and outputs, detects structural cycles, i.e. algebraic loops in the system in case these are present and most importantly for this thesis, reduces the number of model function evaluations by rearranging the evaluation order. The performance of Algorithm 3 was compared to the performance of Algorithm 2 and 2.1 and to the optimal solutions which were computed with an exhaustive search method.

In Table 8.1 and 8.2 we see that Algorithm 3 clearly outperforms Algorithm 2 and 2.1 in terms of reducing the number of model function evaluations. An observation of these results is that the performance of all algorithms decreases as the graph increase in size. This is not surprising as the number of possible groupings of the model outputs increases very fast with graph size. Although the optimal solution is in general not unique, any given grouping still has the potential of making the optimal solution unattainable and the likelihood of doing so is much higher when dealing with a larger graph.

In Section 4.3 the proposed algorithm was also demonstrated on an real industrial example for which it was able to find the optimal solution. Although the example has a relative large number of connections, i.e. 172 connections, the structure of the graph is also very symmetric. This means that finding the optimal solution is near-trivial and further testing with other industrial examples should be done in order to evaluate the practical benefits of Algorithm 3 compared to Algorithm 2.

Studying the results of the execution time of Algorithm 2 and 3 in Figure 8.1 and 8.2, one notes that for practical purposes both algorithms have an execution time which is reasonable. The graphs used in Figure 8.1 have a more vertical structure and increase in size while keeping the number of disconnected components fixated at 2. In contrast, the graphs used in Figure 8.2 increase in size by adding disconnected components. From these results one concludes that Algorithm 2 has a better execution time on graphs that have a more vertical structure and Algorithm 3 has a better execution time on graphs that have a horizontal structure. This due to the first step in Algorithm 3 which tries to group all outputs which are on the same precedence order level. If the graph has a more horizontal structure, this leads to a larger number of initial groupings and thus less iterations in the while-loop are required.

9.2 Parallelization of Model Function Evaluations

In Chapter 5, a parallel method for initializing coupled systems is presented in Algorithm 4. The method instantiates the FMUs in parallel as well as initializes the system in parallel while respecting the precedence constraints. The performance of Algorithm 4 was compared to the sequential case to measure the speedup.

In Table 8.3 we have three different systems of various sizes which are instantiated. We notice that parallel instantiation of Controlled Temperature and Race Car is more costly than the sequential method. This is expected since Controlled Temperature is a considerably small system and the overhead of starting up new processes is big in relation to instantiating a subsystem. In the case of the Race Car, the chassis is the only subsystem which is of considerable size whereas the wheel subsystems are relatively small. Since this creates a bottleneck which combined with also having to deal with the extra overhead involved in starting up new processes, the result is reasonable. In the last system we used two instances of the chassis

from the Race Car. The previous bottleneck is thus eliminated and we see a big improvement in the speedup. We can conclude that in order for parallel instantiation to be beneficial it is preferable if all subsystems are large and do not differ much in relative size from each other.

In Table 8.4, 8.5 and 8.6 we see that without using Algorithm 3 to reduce the number of model function evaluations, the speedup is increased with the number of slave processors used. However, the speedup increases at a considerably lower rate than one would expect from an idealized parallel program, i.e. where the speedup is equal to the number of slave processors used. This is due to the precedence constraints which prohibits a task from being executed before its predecessors. An interesting observation is that the speedup seems to approach 3, which is the same as the number of disconnected components. A hypothesis is that a higher number of disconnected components favors parallelism. This is also backed up in the result from Table 8.7 where the number of disconnected components was set to 6. Comparing Table 8.7 with Table 8.6, we see that the former has a higher speedup than the latter while the number of slave processors is the same in both cases, i.e. 7.

In Section 5.3 the proposed algorithm was also demonstrated on the industrial system Race Car where a speedup of 4 was achieved. Considering that the system requires 5 slave processors, a speedup of 4 is very close to the ideal speedup of 5. This is a considerable better performance in terms of efficiency compared to the synthetic test cases from Table 8.4, 8.5, 8.6 and 8.7. A reason for this might be the symmetric structure of the graph which allows for better parallel computations whereas in the synthetic test cases, the generated graphs had a more complex and randomized structure which introduces more constraints.

In Figure 8.3 we see that the simulation result of the Race Car is identical to the reference result which was computed in a sequential manner. We conclude that the system was initialized properly and that the precedence constraints were respected. Since each slave processor is assigned a subsystem, it should be possible to extend the parallelization to include the computations involved during the simulation. Such parallel methods already exists within PyFMI but without parallel instantiation and initialization. A proposal for future work is to parallelize the whole procedure, i.e. from instantiation to simulation. Moreover, the parallel algorithm as designed in this thesis does not have support for algebraic loops. This could also serve as a proposal for future work.

9.3 Parallelization with a Genetic Algorithm

In Chapter 6, a genetic method for initializing coupled systems is presented in Algorithm 6. The method computes an evaluation order of the inputs and outputs by finding a balance between reducing the number of model function evaluations and maintaining a parallelizable program. The performance of Algorithm 6 was compared to the performance of Algorithm 2, 3 and to the sequential case of Algorithm 1 to measure the speedup.

In Table 8.8, 8.9 and 8.10 we see that Algorithm 3 outperforms the other algorithms, including Algorithm 6. This is not what was expected since Algorithm 6 tries to compute a more sophisticated evaluation order which takes more factors in consideration. A possible explanation for the underperformance of Algorithm 6 is that the parameter settings were far from optimized. Genetic algorithms are very flexible, wherein lies both their strength and weakness and optimizing the parameters requires a lot of empirical testing.

An observation of the results is that Algorithm 6 reduced roughly half of the model function evaluations compared to Algorithm 3 and 2.1 but still managed to achieve a considerable speedup. Factoring in the time penalty of fewer reduced model function evaluations of Algorithm 6 does not explain the time gap between the performances of the algorithms, thus one can conclude that Algorithm 6 had some success with considering the structure of the graph to allow for parallel computations.

Other factors that come into play is the execution time of the algorithm and what kind of assumptions that are made of the evaluation costs. If the evaluation cost of an output is many orders of magnitude more expensive than that of an input, there is probably a stronger incentive to reduce the number of model function evaluations. In this case Algorithm 3 is probably preferable. The same holds if the execution time is not very small relative to the evaluation costs. The execution time will obviously depend on what parameter settings that are used. During our tests, the execution time was about 0.5 s which corresponds to half of the evaluation cost of an output.

In this thesis, only a simple GA was used for Algorithm 6 and the results should be viewed more as a proof of concept rather than a fully implemented method. There is room for big improvements for the use of a genetic algorithm to solve the initialization problem that could yield significant computational gains. An example of such an improvement is that one could optimize the performance by feeding the output from Algorithm 3 and 2 into the initial population of Algorithm 5. Doing so, the algorithm has a

much better starting point than the random solutions which are currently used as the initial population. Moreover, such an improvement would also ensure that the algorithm would find a solution which is at least as good as the other algorithms. Other suggestions for improvements are optimizing the parameters of the GA, using a more sophisticated fitness function instead of a simple list scheduling technique, and speeding up the execution time by parallelizing the algorithm. These improvements are suggested as a proposal for future work.

Bibliography

[1] Functional mockup interface 2.0: The standard for tool independent exchange of simulation models. *Proceedings of the 9th International Modelica Conference*, 2012.

[2] C. Andersson. *Methods and tools for co-simulation of dynamic systems with the Functional Mock-up Interface*. Doctoral theses in mathematical sciences. Lund : Centre for Mathematical Sciences, Faculty of Engineering, Lund University, 2016.

[3] J. Andreasson and M. Gäfvert. The vehicledynamics library - overview and application. *Proc. 5th Int. Modelica Conf. Modelica Association*, 2006.

[4] M. Arnold, C. Clauß, and T. Schierz. Error analysis and error estimates for co-simulation in fmi for model exchange and co-simulation v2.0. *Progress in Differential-Algebraic Equations*, 2014.

[5] E. Coffman and R. Graham. Optimal scheduling for two-processor systems. *Acta Informatica*, 1, 1972.

[6] Z. J. Czech. *Introduction to parallel computing.* Cambridge : Cambridge University Press, 2016.

[7] O. Sinnen. *Task scheduling for parallel systems.* Wiley series on parallel and distributed computing. Hoboken : Wiley-Interscience, 2007.

[8] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[9] G. van Rossum and F. Drake. Python reference manual. 2017. [Accessed 8-April-2017].

[10] K. J. Åström and R. M. Murray. *Feedback systems: an introduction for scientists and engineers.* Princeton : Princeton University Press, 2008.

Master's Theses in Mathematical Sciences 2017:E53
ISSN 1404-6342

LUTFNA-3042-2017

Numerical Analysis
Centre for Mathematical Sciences
Lund University
Box 118, SE-221 00 Lund, Sweden

http://www.maths.lth.se/