DISSERTATION

Distributed Hybrid Co-simulation

Submitted at the Faculty of Electrical Engineering and Information Technology, TU Wien in partial fulfillment of the requirements for the degree of
Doktor der technischen Wissenschaften

under supervision of

Prof. Dr. Peter Palensky
Institute number: E370
Institute of Energy Systems and Electrical Drives

and

Prof. Dr. Wolfgang Gawlik
Institute number: E370
Institute of Energy Systems and Electrical Drives

by

Muhammad Usman Awais
Matr. Nr. 1228472
Forsthausgasse 2-8/2305 , 1200, WIEN

Date _____

Kurzfassung

Computersimulationen zählen heute zu den populärsten Ansätzen für die Analyse komplexe Systeme. Darunter fallen auch Energiesysteme, bei deren Design und Analyse Computersimulationen nicht mehr wegzudenken sind. In den letzten Jahren hat die vermehrte Einbindung von erneuerbaren Energieträgern dabei die Komplexität von Energiesystemen deutlich erhöht. Die Integration von unterschiedlichen Energiequellen, beispielsweise Photovoltaikanalgen, Solarthermieanlagen oder Geothermalanlagen, machen die Analyse dieser Energiesysteme zu einer großen Herausforderung. Um einen konkreten Anwendungsfall analysieren zu können, ist es daher wichtig eine Simulationsumgebung zu haben, die eine korrekte und effiziente Simulation des Gesamtsystems erlaubt.

Betrachtet man die Entwicklung von Simulationspaketen (SP) bemerkt man, dass diese auf natürliche Weise demselben Schema der gegenseitigen Abgrenzung folgen, das man auch zwischen unterschiedlichen wissenschaftlichen Bereichen vorfindet. Die meisten SPs fokussieren sich hauptsächlich auf eine ganz bestimmte Problemdomäne. Das ist vorteilhaft, denn die Entwickler dieser SPs sind Experten auf ihrem Gebiet, sodass der Einsatz dieser SPs in der spezifischen Domäne die Zuverlässigkeit der entwickelten Modelle erhöht. Dazu im Widerspruch befinden sich aber komplexe Modelle, wie etwa im Bereich moderner Energiesysteme, die oft mehrere Domänen umfassen. Um solche komplexen Modelle entwickeln zu können gibt es zwei Optionen: entweder man entwickelt ein neues Simulationspaket, dass alle Domänen abdeckt, oder bereits entwickelte Simulationspakete werden in gemeinsamer Verbindung verwendet. Offensichtlich erfordert die erste Option viel mehr Ressourcen als die zweite, sofern es allgemeine Ansätze für die Kopplung beliebiger Simulationspakete gibt.

Viele Wissenschaftler haben sich bereits diesem Thema aus verschiedenen Richtungen gewidmet. Eine Gruppe von Wissenschaftlern hat sich dabei auf die Entwicklung einer Simulationsmethodik spezialisiert, welche die Modellentwicklung komplexer Domänen ermöglicht. Das ist der Erstellung eines übergeordneten Simulationspaketes sehr ähnlich, was zwar vorteilhaft ist aber keine schnellen Resultate liefert. Eine andere Gruppe spezialisierte sich auf die Entwicklung von Standards für die Simulationsinteroperabilität, während wieder andere sich auf die mathematischen Aspekte von Simulationskopplung fokussierten. Andere versuchten einfach verschiedene Simulationspakete auf die eine oder andere Art miteinander zu koppeln. Aus den Fehlern der letzten Gruppe konnte der Autor ableiten, dass es einen Bedarf an daran gibt mathematisch gut fundierte Methoden für Ingenieure, die sich mit Simulation befassen, zur Verfügung zu stellen. Der Beitrag der vorliegenden Arbeit ist, dass sie einen Blick auf mathematische Methoden für die Kopplung verschiedener Simulationen wirft, die gebräuchlichsten Standards für Simulationsinteroperabilität wählt und dann diese Methoden in Algorithmen für Simulationskopplung konvertiert. Die entwickelten Algorithmen gehen mit den ausgesuchten Standards für Simulationsoperabilität konform. Die die meisten Simulationspakete in separaten Prozessräumen laufen, handelt es sich inhärent um ein verteiltes Problem. Deshalb sind die resultierenden Algorithmen auch verteilt. Algorithmen mit all diesen Eigenschaften sind nach bestem Wissen des Autors noch nie der wissenschaftlichen Gemeinschaft präsentiert worden. Nachdem es in diese Richtung laufende Forschungsaktivitäten gibt, sind hier die populärsten Methoden ausgewählt worden, um verteilte Algorithmen zu erstellen. Die ausgewählten Standards für Simulationsinteroperabilität sind High Level Architecture (HLA) und Functional Mock-up Interface (FMI). Die präsentierten Algorithmen wurden getestet, um ihre Vor- und Nachteile zu evaluieren. Das Endergebnis der präsentierten Arbeit ist ein Framework, dass es Ingenieuren mithilfe von Simulationspaketen, die HLA unterstützen, erlaubt, Simulationen zu entwickeln, die unterschiedliche Domänen verbinden. Das Framework beinhaltet Implementierungen all der abstrakten Algorithmen, die in der vorliegenden Arbeit diskutiert sind. Dabei abstrahiert es die Komplexität von HLA und FMI. Um die laufende Forschung zu unterstützen, ermöglicht das präsentierte Framework Wissenschaftlern die Entwicklung und das Testen von neuen Algorithmen auf einfache Weise.

Abstract

Simulation is currently one of the most popular methods for analysis of complex systems. Like many other domains, energy systems rely heavily on simulations for model verification and analysis. In past few years promotion and increase of renewable energy systems have added many complexities into energy systems. Introduction of many different forms of energy sources, like, solar electrical energy, solar thermal energy, geothermal energy and others, have made the analysis of energy systems much more challenging. To analyze a complete scenario given at hand, it is important to have a simulation framework which is able to simulate the scenario accurately and effectively.

Looking at the development of simulation packages (SPs), it is natural that they have followed the same pattern of separation as found in different scientific fields. Most SPs are focused mainly on one certain type of problem domain. It is beneficial, because the people who have developed SPs are experts in their fields, so using SPs for their specific domain increases reliability in the developed models. Contrarily, complex models, like modern energy systems, are often related to multiple domains. To develop complex models there are two options, either a new simulation package could be developed which covers all domains, or already developed simulation packages could be used in conjunction. Clearly, first option requires much more resources than the second one, only if techniques could be devised to couple any number of simulation packages.

Many scientists have already been focusing on the problem from different directions. One group of scientists focused on developing a simulation methodology which could enable model development of complex domains. It is similar to creating a grand simulation package for all types of simulations, which is beneficial but does not give rapid results. Another group focused on developing standards for simulation interoperability, while others focused on mathematical aspects of simulator coupling. Some others just tried to couple different simulation packages by one way or the other.

Learning from mistakes of the last group, it is realized by the author that there is a need of putting mathematically sound techniques into the reach of simulation engineers. The contribution of this work is that it looks into the mathematical techniques of coupling different simulations, and choses most popular simulation interoperability standards, then converts those techniques into algorithms for simulator coupling. The developed algorithms conform to the selected simulation interoperability standards. As most simulation packages run in separate process spaces, so the problem is inherently distributed. Consequently, the resulting algorithms are also distributed. As per knowledge of the author, algorithms having all these capabilities have never been presented before in the research community. As there is ongoing research on simulator coupling techniques, so here most popular techniques have been chosen to be converted into distributed algorithms. The selected simulation interoperability standards are the High Level Architecture (HLA) and the Functional Mock-up Interface (FMI). Tests have been performed on the presented algorithms to measure their limitations and benefits.

In the end the combined effort is converted into a framework, which allows simulation engineers to develop multi-domain simulations, using simulation packages conforming to HLA or FMI. The framework contains implementation of all the abstract algorithms discussed in the presented work. While doing so, it abstracts away the intricacies of the HLA and the FMI. To facilitate ongoing research, presented framework enables the scientists to develop and test new algorithms with ease. A simulation framework enabling standardized co-simulation with the ability to use such wide range of simulation packages in a distributed environment, is not known to be presented before.

Table of Contents

ABBREVIATIONS

FMI	Functional Mock-up Interface
HLA	High Level Architecture
FMU	Functional Mock-up Unit
RTI	Runtime Infrastructure
SP	Simulation Package
WR	Waveform Relaxation
FTS	Fixed Time Stepped
FTSS	Fixed Time Stepped Simulation
DES	Discrete Event based Simulation
DEVS	Discrete Event Specification
CS	Continuous Simulation
IRM	Interoperability Reference Models
ICT	Information and Communication Technology
DAE	Differential Algebraic Equation
ODE	Ordinary Differential Equation
GUID	Globally Unique Identifier
GUI	Graphical User Interface
TAR	Time Advance Request
TARA	Time Advance Request Available
NER	Next Event Request
NERA	Next Event Request Available
FQR	Flush Queue Request
FOM	Federation Object Model
SDK	Standard Development Kit
TSO	Time Stamped Ordered
IJCSA	Interface Jacobian-based Co-Simulation Algorithm
SAHISim	Standardized Architecture for Hybrid Interoperability of Simulations

1 Introduction

Simulation engineering is very important in present day scientific studies and industrial applications. Some philosophers [Win99] argue that simulations offer a complete new method to prove hypotheses and observing real world phenomena. They argue that this is an augmentation in the human methods of observations. In many industries, simulation has now become an essential part of design and engineering. Weather forecasts have always been dependent on simulations of weather models. The increasing complexity in energy systems has also motivated scientists to consider simulation as a method of analysis. Albeit, its large acceptance poses some intriguing challenges.

1.1 Motivation

To properly motivate the reader of importance of the presented work, some background knowledge is necessary. Following subsections introduce the relevant material and emphasize the importance of the presented work. Based on the discussion, grounds for the problem statement (section 1.2) are set up.

1.1.1 Necessity of Co-Simulation

As modeling and simulation has become a corner stone for verification of mathematical models of systems found in nature. Development of simulation packages followed the same pattern of categorization and segmentation which is natural to the study of natural phenomena. Physicists, chemists, mathematicians and mechanical engineers have developed simulation packages suitable for their domain. For past few decades there have been a number of simulation packages popular for simulation engineering. In one way or the other all Simulation Packages (SP) either focused on one domain of problems, or a specific type of simulation engineering.

Due to the easy availability of simulation packages, recently researchers have found themselves in a situation where developing simulations in programming languages like C++ or Java has not remained time and cost effective. Because already developed simulation packages offer great time efficiency and cost effectiveness, scientist and practitioners are motivated to use them. As almost all of the simulation packages conform to certain domain of problems, their use is restrictive. This is one reason that scientists are finding ways to use more than one simulation packages in conjunction to create a larger simulation. The technique has additional benefits, as the models developed using domain specific SPs are well tested and properly verified, so a simulation constructed using them is supposedly stable.

A very handy example is of Cyber Physical Systems (CPS), where scientists want to couple cyber aspect of a system to its physical aspect. In many cases cyber and physical parts are simulated using separate simulation packages. Another example is modern energy systems. Complexity of energy systems has increased due to many factors. One reason is that the modern energy systems comprise of many subsystems which themselves are significantly complex. For example, traditional power systems were studied as a standalone system, but this is not true anymore. Just to keep a power system running in an optimal way, it is necessary to use many other systems to its aid. For example, an optimized model-predictive control of an urban power system is not possible without Information and Communications Technologies (ICT). Adding heating system into it will increase the complexity even more.

1.1.2 Hybrid Simulation

From the perspective of time management a simulation can be divided into three categories [Van00]. A brief overview of all three types is given below.

- **Fixed Time Stepped Simulations (FTSS)**: Such simulations have fixed time advances. If there are more than one simulations, then they communicate with each other after a fixed logical time. All simulations interacting with each other must have a common notion of logical time, and the logical time should advance by a fixed unit .
- **Discrete Event based Simulations (DES)**: Such simulations advance in time only when an explicit event occurs. An event may be anything important for the simulation. For example, network simulators are commonly discrete event based simulators. An event occurs when an element in network produces or transmits a packet.
- **Continuous Simulations (CS)**: Such simulations are normally a mathematical representation of a physical system. The state variables in such systems change their values infinitely. To simulate these systems computers have to apply some discretization. To reduce the discretization errors normally the time steps are kept very small. Because the physical systems are represented as Ordinary Differential Equations (ODEs), Partial Differential Equations (PDEs), Differential Algebraic Equations (DAEs) or as a mixture of all, so numerical methods are applied to solve such systems. The boundaries of CS and FTSS paradigm are overlapping, due to the way continuous systems are simulated. Many a times continuous systems are simulated in a fixed time stepped manner. The simulating module has an interface which turns it into a strictly FTSS interface. Still the differences remain there and cannot be ignored. For example, adding a step size control mechanism to a continuous simulation may take it out of the realm of FTSS simulation.

In literature, the term heterogeneous simulation and hybrid simulation is used interchangeably for different purposes. Sometimes, it is used where more than one methodologies of simulation are used in conjunction [EJL+03]. It is also used when there is a simulation comprising of more than one components where each component has different time advance mechanism [KKK03], [LK01]. Sometimes the term is used for the simulation of systems which are hybrid in nature [SK11]. Hybrid systems are the ones which have discrete and continuous aspects in them, like many cyber physical systems. Simulating a hybrid system may not necessarily result into a heterogeneous simulation, it mainly depends on the nature of the system that it can be simulated by a single simulation methodology or not. Sometimes it is possible to simulate a hybrid system with the help of a purely CS, FTSS or DES. Later in chapter 2 simulation packages will be introduced which are designed specifically for hybrid systems and they use only one type of simulation paradigm. In the thesis, the term hybrid refers to the simulation of hybrid systems. Heterogeneity in a simulation occurs when different types of simulation mechanisms are combined together.

1.1.3 Standardized Simulation

Using already verified models to construct a simulation of a larger system, like an urban energy system, is very beneficial, but the benefits can go in waste if the developed technique itself requires the simulation packages to modify. Suppose that in result of the presented study a technique is developed to use many different simulation packages. One thing is for sure that such a technique must have following components

- **Data Sharing**: because more than one simulation has to interact with each other, so there has to be a way to share the values of state variables among all simulations.
- **Time Synchronization**: from section 1.1.2 reader knows the importance of time in simulations. To create a simulation from more than one simulation packages, there has to be a common notion of time for all the simulations, or a way to synchronize time for all of them.

Now suppose that both of these components are developed from scratch, then it will be another problem to make the developed technique acceptable for the practitioners. So intuitively the most useful technique could only be the one which follows already developed standards of simulation interoperability (co-simulation). Using standards can make the technique acceptable in industry as well as research.

1.1.4 Distributed Simulations

Computers have made solving systems of differential equations much easier than before. Numerical methods made solving some systems much easier in comparison to very tedious analytic methods. New advances in computer technology has opened even new horizons in simulation of differential equation systems. Mostly, simulation of non-linear systems assumes a monolithic approach towards a complete system. A common approach is to symbolically simplify the given system by applying different algorithms, such as Tearing algorithm [CK06], and then apply the numerical techniques on the "simplified" system. In this way the simulated system is a perturbation of the original system, but it is still valid under given conditions.

With the advent of parallel computing, there had been much focus on devising algorithms which could benefit from the parallel architecture. The overall technique remains the same, the only difference is that scientists try to find algorithms which can parallelize the instructions and thus benefit from the parallel architecture. For example, Runge-Kutta methods have been intensively studied for this purpose [KW14].

Another way to achieve parallel execution is to exploit the inherent distributed structure of the problem and execute the atomic parts as separate executables on different machines. Figure 1.1 show the difference between serial, parallel and distributed techniques. In order to use distributed technique the system should have some specific properties. First and foremost, the structure of the system should allow such distributed execution. Secondly, solver algorithm needs some modification for such construction. Thirdly, data exchange among individual components has to be synchronized. Despite its limitations, such a technique has many benefits.

- A complex simulation, for example a smart grid [PK13] simulation, must use models generated from many different simulation packages. Network simulators and power simulators have to work together to produce a "correct" simulated result. The problem can be used for benefit, by exploiting the distributed nature of the problem and orchestrate a simulation using a distributed topology.

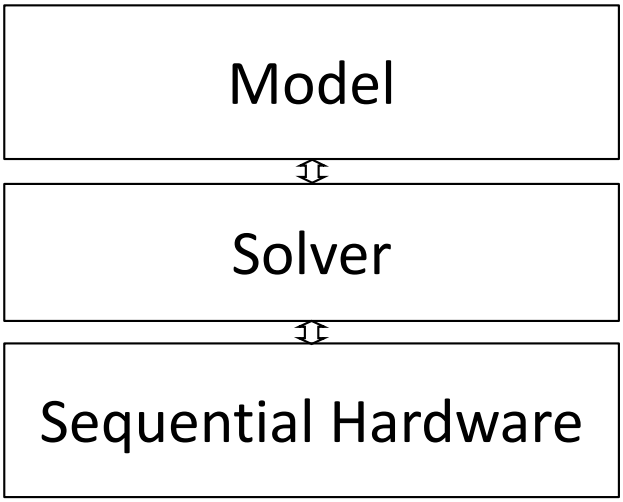
- In some cases, there is an option of creating a simulation from scratch, or to use an already developed and tested simulation component in the target simulation. In such cases, it is much cheaper and less error prone to use an already tested simulation components. Distributed simulation is useful in this regard.
- Distributed simulation techniques are also useful when there is no complete mathematical definition of a system. For example, agent based simulations normally target such problems where modeling component interaction is much easier than modeling a complete mathematical description of the system. Smart grids simulations and other similar co-simulation problems also fall in the category.
- A direct consequence of distribution is process isolation. In any simulation, where there is a need to simulate parts of a system in different simulation packages, distributed simulation techniques are the **only** choice.

1.2 Problem Statement

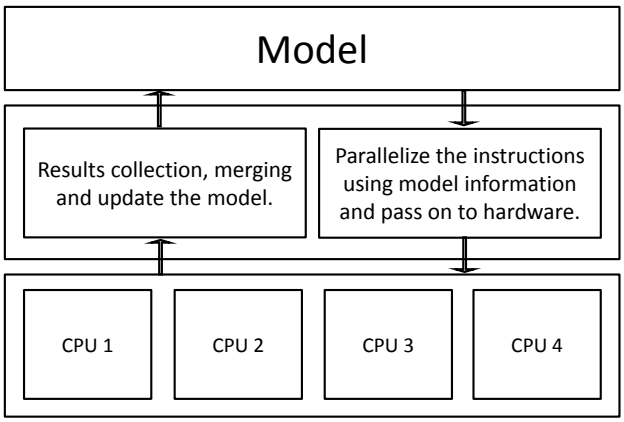
Summarizing above sections, following are the main aspects of the study.

- **Co-Simulation**: due to the reasons described earlier, scientists and practitioners want to use already developed simulation packages to create a larger simulation.
- **Hybrid Simulation**: co-simulation becomes harder when hybrid nature of the system or simulation is introduced, yet it is essential to overcome this hurdle.
- **Standardized Simulation**: if the goal of co-simulation is achieved without using standards then it will not be very useful for industry.
- **Distribution**: simulating using separate simulation packages is already a distributed phenomena, especially when more than one simulators are running in separate processes. The distribution is a challenge as well as an opportunity.

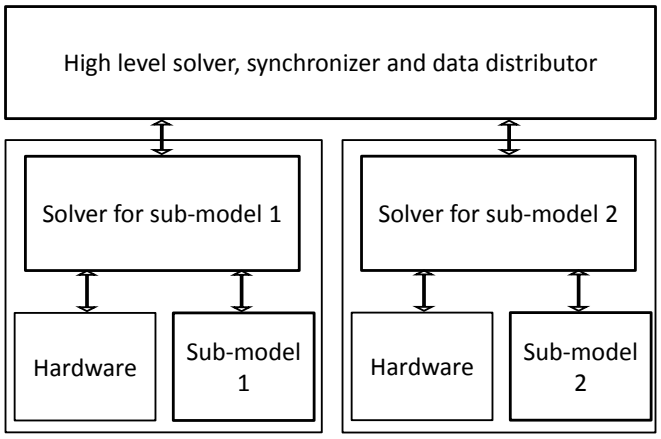
In order to achieve above mentioned goals it is clear that a collection of **distributed algorithms** have to be developed. The algorithms should cover range of already proposed solutions for **co-simulation**. The algorithms should conform to most suitable **simulation interoperability standards**. The algorithms should allow engineers to create a large simulation using more than one simulation packages conforming to the chosen simulation interoperability standards. One or more algorithms should also be able to tackle the **hybrid** nature of the simulation.



(a) Simple separation of three main components of a simulation.



(b) Organization of simulation components in parallel techniques of simulation.



(c) Organization of simulation components in distributed techniques of simulation.

Figure 1.1: Different types of simulations

2 State of the Art

Many scientist have tried to solve different aspects of the problem described in section 1.2. Until the time of writing no body focused on distributed algorithms, although parts of the problem were researched. Researchers from the filed of mechatronics have addressed the problem of hybrid simulation [Val09], but they did not focus on its distributed and algorithmic aspect. Some other domain specific solutions have also been presented in the research, for example smart grids simulators, but they also lack in different respects, as discussed in subsection 2.4.

Interoperability Reference Models			
Category	Name	Description	Example
A	Entity Transfer	An entity leaves a representative model and another must reclaim its ownership.	Assembly lines consist of many such examples.
B	Shared Resource	Two or more representative models depend on the same source, shared in common. Synchronization is the main problem in this case, as changes made to the source should take effect properly.	A sensor making a reading and many models needing that information.
C	Shared Event	One event has a broadcast nature, affecting many representative models simultaneously.	Signals for same kind of processes are common in many forms.
D	Shared Data Structure	At an implementation level a variable or structure that must be accessed commonly.	It is much the same as shared resource but here things are handled at data structure level.

Table 2.1: Interoperability Reference Models

2.1 Interoperability Reference Models (IRM)

The Product Development Group (PDG), working under Simulation Interoperability and Standardized Organization (SISO) has suggested the standard Interoperability Reference Models (IRM) for simulation packages [BBC+10].

These IRMs provide a structured way to compare the capabilities of any interoperability or co-simulation standard. Table 2.1 shows the summary of these IRMs. For detailed explanation reader is directed to [BBC+10]. Keeping these IRMs in mind, different simulation interoperability standards can be analyzed better.

2.2 Simulation Interoperability Standards

The modeling and simulation community has presented many different standards for simulation interoperability, targeting the same problem from different perspectives.

- US Modeling and Simulation Coordinating Office (M&S CO) developed the Distributed Interactive Simulation (DIS) standard [C+98]. The DIS was specifically designed for military simulations.
- During the development of the DIS the MITRE corporation was developing its own standard named, the Aggregate Level Simulation Protocol (ALSP) [WW94].
- Later lessons learned from these simulation standards were applied to present the High Level Architecture (HLA) [DFW97]
- A web based framework named as the Extensible Modeling and Simulation Framework (XMSF) [BPM+04] is an extension of both DIS and HLA. The XMSF addresses the issue of creating a web-based interoperable platform for simulations.
- Few years back the M&S CO had proposed another architecture named the Test and Training Enabling Architecture (TENA) [DoD02]. TENA is not yet available openly.

Researchers working on topics related to weather and climate modeling have also faced the same difficulties in sharing each others experiences, so they have proposed solutions for their own spectrum of problems. Few of these solutions are listed here.

- The Model Coupling Toolkit (MCT) [LJO05] is a Fortran 90 toolkit for exchanging earth models. It targets multiprocessor computers and clusters. It is similar in architecture to the Message Massing Interface (MPI). It is an API based on Message Passing Interface so, it may be considered another layer above the layer of MPI. It promotes a specific programming paradigm (parallel programming in Fortran). It has constructs which are specialized to parallel coupling of the models, and does not allow flexibility in organizing different aspects; like data distribution and time management.
- The Earth System Modeling Framework (ESMF) [HDS+04] is one of the biggest initiatives taken for modeling earth systems. It envisions coupling many different models into one entity.
- The Open Modeling Interface or OpenMI [GGW07] was initiated as a model coupling platform for earth systems, but now it is further refining itself to become a generic interoperability solution for simulations.

2.3 Hybrid Simulation Techniques

Discrete EVent Specification (DEVS) provides a systematic way of converting a continuous simulation into a discrete event simulation [DW03]. DEVS has been considered an important concept for heterogeneous simulations. Although, the biggest draw back though is, with DEVS arbitrary simulation packages are difficult to couple. Only those simulation packages can be used which create DEVS specific simulations, or produce DEVS compliant components.

Ptolemy II [EJL+03] is another attempt to make development of heterogeneous simulations easier and faster. It may effectively be used to solve any system of differential equations. Ptolemy II, in a limited perspective, may be considered as a tool which supports distributed simulation. The biggest disadvantage of using Ptolemy II is its lack of flexibility. Although the methods for implementing a new solver or incorporating a new "director" are well documented, yet there are some limitations imposed by the Ptolemy II kernel. For example, how it treats the events in the queue, and when and how it processes them. The behavior can only be changed by changing the kernel of the Ptolemy II.

2.4 Smart Grid Simulators

Due to introduction of ICT into the power grids management, it has become vital to simulate power grids in conjunction with ICT infrastructure. In recent past there have been quite a number of efforts to couple ICT network simulators with power system simulators [MOD14] to simulate cyber physical energy systems or in other words smart grids [PK13]. There are a few flaws in the proposed systems.

Most smart grid simulators do not try to couple more than one continuous systems. Mostly only a continuous power system simulator is coupled with a discrete network simulator. In this case, the continuous system formed does not have any algebraic relationship among simulation components, which make things much easier and manageable. Despite this, there are some problems which remain unanswered.

Many of the smart grid simulators include implicit assumptions about the synchronization of simulation. For example, many of smart grid simulators do not even address the problem of time synchronization. They implicitly assume it to be a solved problem. It is not mentioned whether there is any level of parallelism involved or not? If yes, then how they make one process to wait for the other process to reach the same time step?

The use of fixed time step or a variable one is also not clear in many solutions. If variable step size is used, then how the time step is adjusted? Some power system SPs used in aforementioned solutions, only support fixed time step execution. It can cause problem, when a discrete event does not occur right at the boundary of a fixed time step. Researchers do not mention how the problem is solved?

Some solutions give discrete events higher priority than other events. Before advancing to the next time step ($t_{n+1} = t_n + h$), they first process the discrete event and then proceed forward. Calculation will be erroneous if discrete event does not occur right at the boundary of a fixed time step. If fixed time step has to be altered to match the discrete event, then only those power system SPs can be used which support variable step size, and their step size control is programmable. Moreover, if there is some error discovered after publishing the discrete event, then solutions do not mention how to tackle the problem.

Lack of theoretical basis is not customary in all the solutions developed for smart grids, there are solutions developed under formally defined simulation paradigms, for example, solutions developed using Ptolemy and DEVS.

3 Opportunities and Challenges of Co-Simulation

Before looking into the proposed solution presented in the document, it seems appropriate to have more knowledge about the problem itself. In the present chapter, nature of the problem is discussed, along with the opportunities already available for the targeted solution.

3.1 Challenges of Co-Simulation

First the challenges of the problem are discussed, which also educate the reader about the nature of the problem.

3.1.1 Correctness of Results

A co-simulation problem is different from a normal simulation problem. Let us first examine what is a simulation, and why it is helpful? Figure 3.1 shows the steps to properly model a physical phenomenon and simulate it. One critical part in this is "validation" of results. Anything which is simulated does not necessarily present correct results. Instead, after preparing a model, the scenario in physical model should be altered with some specific parameters, then the same parameters should be applied to the model and the results produced by simulation and the real world should be compared. With some error tolerance the results should match if the model is correct.

With respect to validation there are following main problems with co-simulation

- Sometimes the model is complex enough that the scientists do not understand the model completely, instead, they know parts of the system and model them. Now the expectation from the co-simulation algorithm is to produce results near to the "expected" correct results. In such cases verification may not be possible at all, as a researcher may not be certain that how alteration in a parameter would produce difference in results.
- In many cases, as discussed in section 1.1.4 the phenomenon is just not possible to be simulated by one simulation package. The only option remains to rely solely on the co-simulation algorithm to produce correct results. The results are verifiable in this case, only if this is practically possible. As sometimes the phenomenon under study is too large or too complex to verify at all. For example simulation used in planning phase of a power grid of a city. In the long run it may become possible to get real life data for the city grid but in short term the simulated scenario my not be verifiable.

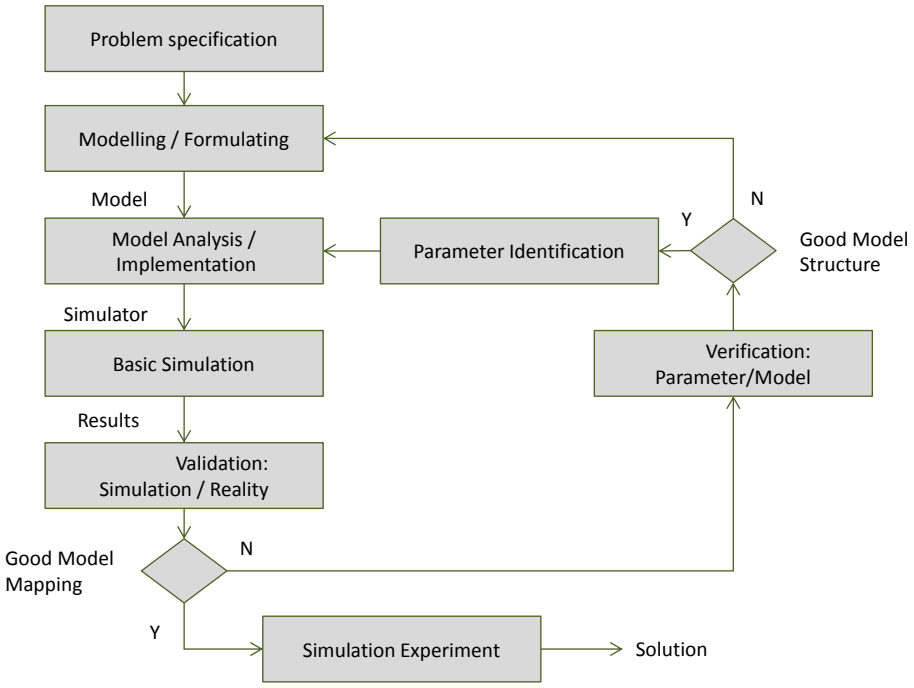


Figure 3.1: How to model a physical phenomenon and simulate it [BS86].

In scenarios where simulation results are not verifiable, modelers would naturally like to choose an algorithm which is known for its correctness of results, no matter how much slow it is, as correct results produced after some time are more useful for deduction, than wrong results produced swiftly. So one challenge of co-simulation is to produce correct results in many different scenarios and for many different problems.

3.1.2 Differential Algebraic Equations

A Differential Algebraic Equations (DAE) is a differential equation with implicit derivatives. The general form of a DAE is given as following

$$F(\dot{x}, x, t) = 0, \qquad t_0 < t < t_f \tag{3.1}$$

Here t is the independent variable which is mostly time. General purpose solvers first analyze a DAE and try to reduce it to an Ordinary Different Equation(ODE). DAE is also formed when implicit derivatives are introduced in a system of ODEs and algebraic equations. Often the situation occurs in co-simulation, because coupling more than one systems can cause algebraic loops to be constructed. If the complete system is not available, then it is not reducible to a system of ODEs. In such situations modelers will like to choose solvers which produce correct results for DAEs. Chapter 5 presents algorithms for such problems.

3.1.3 Heterogeneity

Chapter 1 described different types of simulations with respect to time synchronization. A co-simulation may require to contain components with different time synchronization techniques. Mixing different time synchronization techniques offer few challenges. For example, there are certain limitations in mixing FTSS and DES simulations. Suppose an FTS simulation has a time step of 5 logical units. If the accompanying DES simulation generates events after such logical time intervals, where the length of interval is an integer multiple of 5, then there will be no problem in mixing both simulations. If this is not the case, then there will be problem of synchronization in them.

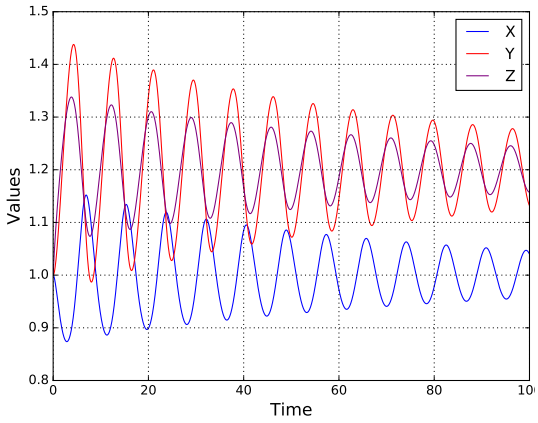
Similarly, coupling a continuous time simulation with any DE or FTS simulation poses many challenges. For example, if a solver for continuous time simulation converges on a result after some fixed point iterations, and a discrete event occurs after the last successful time step, then how to manage the time step of continuous simulation that the discrete event only occurs at the boundary of the time step. If discrete event does not occur at the boundary of a time step, then either the fixed point iteration will not successfully converge at any solution, or the result will be erroneous. The convergence, most likely, will not be achieved if the change in values of state variables resulted in consequence of discrete event, are considered in the calculation. If discrete event is completely ignored, then no doubt results will be erroneous. Section 5.5 presents an algorithm to tackle this problem.

3.1.4 Sensitivity of Model Parameters

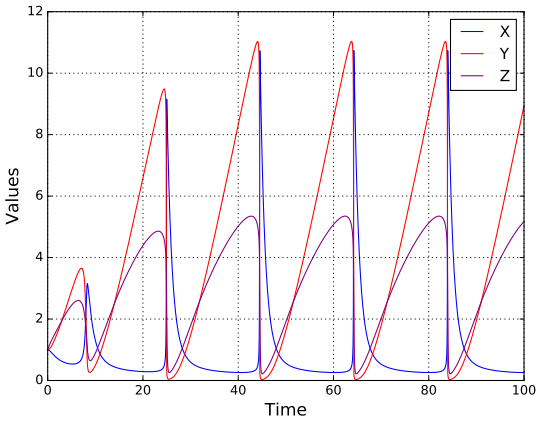
When coupling two subsystems using techniques mentioned in later sections, few things has to be kept in mind. No matter how good the coupling algorithm is, with current state of the art, it cannot point out the modeling mistakes. One mistake a modeler is likely make while simulating coupled systems, is bad separation of models. It will be discussed later in section 5.2, that how important separation of sub-models can be, here only one example is presented to illustrate its importance. The example model is named as "hopf bifurcation" [WH91].

Before coupling a subsystem with any other, the least a modeler must know about the subsystems is the valid boundary of each parameter value. As shown in following example, some systems are very sensitive to some of their parameters, if naively such parameters are chosen to be inputs to the subsystem, and their values come from the output of another subsystem, then problems may arise. Before doing so a modeler must verify that the outputs bound to such sensitive inputs do not produce trajectories which force the system to go into an undefined state. The exemplary model is described by following system of equations, which is very sensitive to its parameter α. Changing the value of α from 1.2 to 1.5, changes the system state to a great deal. Further exceeding the value of α beyond 1.5, takes the system into undefined state.

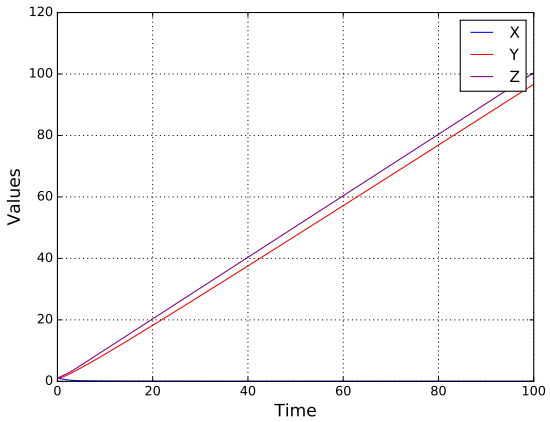
$$\dot{x} = 1 + x^2 y - (z + 1)x$$
$$\dot{y} = xz - x^2 y$$
$$\dot{z} = -xz + \alpha$$



(a) Hopf bifurcation, here $\alpha = 1.2$



(b) Hopf bifurcation, here $\alpha = 1.5$



(c) Hopf bifurcation, here $\alpha = 2.0$

Figure 3.2: Hopf bifurcation model simulated using different values of α.

3.2 Tools for Simulation Interoperability

As the presented problem has a focus on interoperability of the simulation packages–which can be considered tools for simulation–so it is very important to look into the tools for "simulation interoperability".

Chapter 2 enlists different interoperability standards. Choosing the right standards is very important, due to their diversity. Following are some of the characteristics used for the selection.

- How well the standard is accepted in industry or research? More conforming SPs–pertaining to different domains of simulation—will enhance the usability of the standard.
- How modern are the conforming SPs of the standard? Are conforming SPs also offer support, and are they improving their quality? It has been observed that software decay with the passage of time, so software under continuous development are observed to be more suitable for use with modern systems.

- Is the standard openly available? Open availability enhances the chances of continuous improvement. Also, it enhance the number of users and contributors, which may be helpful in improving the quality.
- Is the standard specific to some domain of applications, e.g. weather simulations, mechanics, etc. If this is the case, then it may not be useful to develop any algorithms based on that standard, as it will enforce domain specific constructs, limiting the generality of the proposed architecture.
- Is the standard capable of being used in conjunction with other standards? The answer may be subjective, and may depend upon the answers of the above questions. Ideally, a generic standard should be usable with others.

Based on the above mentioned criteria, according to author's knowledge the Functional Mock-up Interface (FMI) and the High Level Architecture (HLA) are two standards which suite best for the proposed solution. To further support the claim, let us examine the interoperability standards mentioned in section 2.2, based on the criteria just mentioned.

- In interoperability standards developed by US Modeling and Simulation Coordinating Office (M&S CO), the HLA is the most popular one. The standard proposed before HLA was DIS and it has become obsolete after development of the HLA. M&S CO does not support it anymore, neither there is any ongoing research activity on the DIS. Similarly ALSP was merged into the HLA and have lost its separate identity. TENA standard though is not obsolete, but it is not open source. Access to it is very hard to get, so it is not useful to make it part of an open research activity.
- The Extensible Modeling and Simulation Framework (XMSF) is not a separate standard, rather it enhances the capabilities of the DIS and HLA to be executable on the web. So it may only be chosen for further research if the purpose is to target any web based application.
- The Model Coupling Toolkit (MCT) is an open source toolkit. It cannot be called as a standard, because it is not only restricted to specific implementation tools like Fortran 90, but it is only executable on certain hardware platforms. It does not give a strong sense of data sharing and time synchronization like the HLA. It enforces a specific programming model. Its scope is also limited because there are not many general purpose simulation packages which support MCT. Its application is confined mainly to the weather simulations, which are developed using the specific toolkit.
- The Earth System Modeling Framework (ESMF) is also not a standard, rather it is a software, enforcing some rules on the developer who wants to use it. Again, not many general purpose simulation packages conform to the programming model of ESMF.
- The Open Modeling Interface (OpenMI) is one standard that can be compared to the FMI. It mainly focuses on the correct mechanism of data sharing among different models. As its documentation mixes up the specification with the implementation, so even the specification is characterized by the implicit assumptions of the runtime to be used. OpenMI proposes a Standard Development Kit (SDK), which includes the specification of the runtime. It may be argued that the HLA also enforces the implementation under the assumption of using an RTI. The difference is that HLA is more close to a distributed protocol, but OpenMI requires to use its SDK, which is much more restrictive. OpenMI SDK also gives greater flexibility of developing the simulation but it also limits the researcher from using choice of his tools. OpenMI is not distributed in nature, supporting distributed simulation will require additional effort. The biggest disadvantage though is, there are hardly any “general” purpose simulation packages which conform to it. As it is focused on weather simulations, so only the relevant software try to conform to it.

In general the HLA provides a very abstract way of data sharing and time synchronization constructs which are essential for any simulation. Additionally its distributed nature gives even greater flexibility and makes it scalable. The FMI is relevantly silent about the data sharing mechanisms and time synchronization. It leaves all of these things to the user code, but it does provide an abstract way to control a simulation component. It represents a simulation in a pure mathematical way. As HLA is silent about the representation of the simulation component, so both HLA and FMI fit very well together to provide a researcher the basic constructs he needs for the research. The vast acceptance of the FMI in general purpose simulation packages makes the research highly practical and beneficial.

3.2.1 The Functional Mock-up Interface (FMI)

Functional Mock up interface is given a special focus in the research. Mainly because it provides an opportunity to use many different simulation packages for development of specialized simulation components. Secondly, it has been developed on a thorough and sound mathematical basis.

Historically, the FMI (in FMI 1.0) had mainly two portions.

1. FMI for Co-Simulation
2. FMI for Model exchange

The main difference among both of these standards is the level of encapsulation. Components conforming to the FMI for co-simulation have a very straight forward interface, mainly due to the fact that, the integrator of the model is included inside the component. While a component conforming to the FMI for model-exchange does not include the integrator, so the user code has access at more granular level, making it less encapsulated. Later, this clear separation was abandoned in FMI 2.0, yet the underlying concept of different level of encapsulation remained intact. As this difference of encapsulation provides a better way to investigate the two approaches, here they are explained separately. The documentation of the latest FMI standard can be viewed at [60].

Some terminologies of the FMI are mentioned before we go into the detail of these specifications.

FMU Functional Mock-up Unit (FMU) is a component in form of a zipped archive which conforms to the FMI specifications. The most important part of this zipped archive is a shared C/C++ library which can be imported and used as component. This shared library contains the implementation of the simulated model.

Master The master is an application or component using the FMU. This normally is a simulation developed in a simulation package, which supports the FMI. Ideally it uses one or more FMU components for assistance in a larger simulation. In hierarchical architecture, the master can be any other FMU itself.

Slave The slave is the FMU being used by some other simulation component or application.

In order to completely understand the functionality of FMI, it is necessary to understand the construction of FMU. As mentioned earlier an FMU is a zipped archive, and following are the contents of this archive.

- Model Description file: This is an XML file which contains all the important information needed to use the implementation. The main contents of this XML schema are the input variables of the FMU with their data type, its parameters and its output variables. In addition, it encloses the information about the default start time, end time of FMU along with the tolerance. It has the author and version information. It also includes, the GUID attached to the FMU, model prefix or name, number of continuous states and number of event indicators. It tells whether the FMU can be used without any integrator (meaning integrator is inside the FMU) or not, this is equal to telling that the FMU supports co-simulation or not?

  Some of the descriptive tags and attributes vary depending on the fact that the FMU is for co-simulation or model exchange. It contains the information, whether the FMU can accept events or not, it can run asynchronously or not, the maximum order of the derivatives, it can raise events or not, does FMU has a GUI interface or not? Besides these, there is a lot of information which may be provided into model description file, details of this can be seen in the documentation provided at [60].

- The most important part of an FMU is the shared library. This can be a .dll file in case of shared library supported by Windows, and .so file in case of *nix based systems. It contains the implementation of the model. In case of FMI for co-simulation it also contains the integrator. The access to this model is according the interface specification defined by the FMI.

- Third optional part is the code of the FMU. In case of propriety FMUs this part will not be present, but in case of open source FMUs it is zipped inside the archive for further improvements.

3.2.1.1 FMI for co-simulation

The concept of "master" and "slave" in the FMI is very different from that of HLA's. When an SP imports an FMU, it becomes its "master" and the loaded component becomes its "slave". FMI for co-simulation provides the means to utilize models using an API, where slave appears as a black box to the master. It can react to inputs and gives outputs at discrete time steps. While using an FMU conforming to FMI for co-simulation, one does not need to know which integration method is actually applied to solve the model.

Parameters to this black box can be set in the initialization phase and cannot be changed afterwards, while the inputs can be changed between discrete time steps. Functions in the FMI can be categorized in following categories. Figure 3.3 gives possible execution steps of an FMI component.

- **Initialization and instantiation functions:** These are used to load the component, supply the parameters and allocate memory.

- **Progress functions:** There is "fmiDoStep()" function which requests the component to step one time step ahead.

- **Getter and setter functions:** Getter functions are used to read the output values and their derivatives. Setter functions can be used to set input values and their derivatives for interpolation.

- **Termination functions:** These are used to unload the component and free the memory.

From figure 3.3 it is clear that there are few alternative paths of execution for an FMU, hence it is possible to write a program which loads it and performs these actions repeatedly.

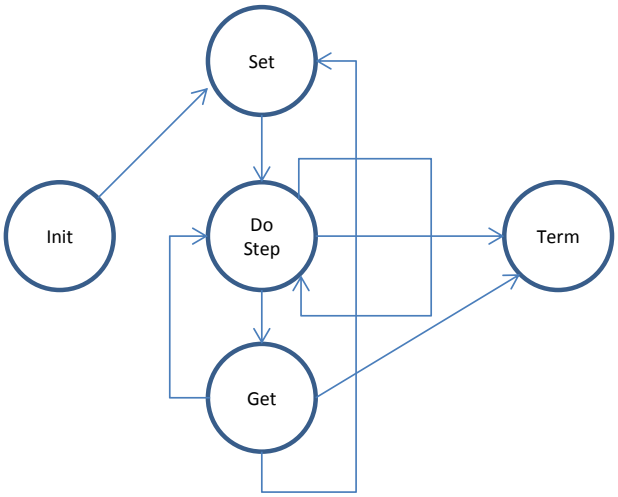


Figure 3.3: States of an FMU

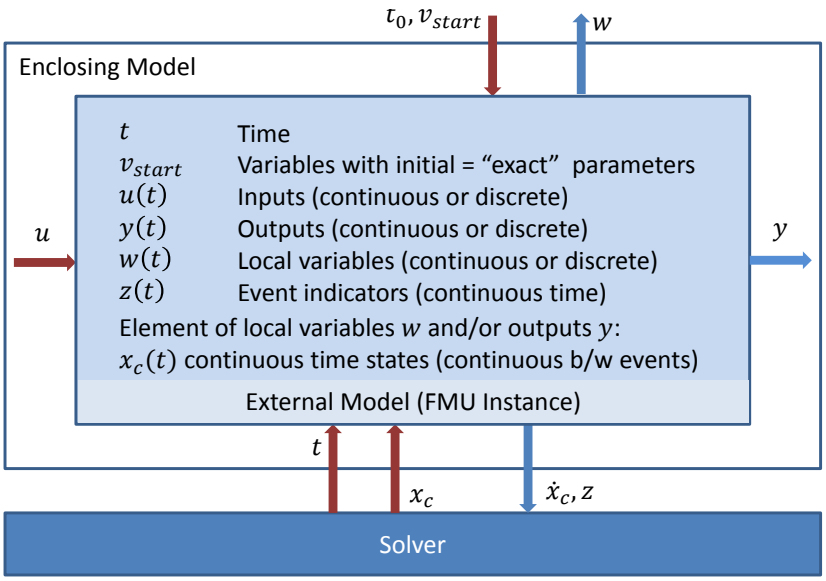


Figure 3.4: FMI for model exchange (courtesy FMI 2.0 specification)

3.2.1.2 FMI for Model Exchange

All FMUs are exported the same way as described earlier in section 3.2.1. The only difference is the kind of data they export for the user and the functionalities they offer.

According to FMI 1.0 specifications [BOA+11], the FMI for Model Exchange (ME) is intended to couple hybrid Ordinary Differential Equations (hybrid-ODE). However, FMI allows that a component conforming to FMI, called Functional Mock-up Unit (FMU), may have algebraic equations within. One most important thing to remember about FMI for ME is that is does not come with a solver or integrator. It virtually acts like a "right hand side" of a conventional ODE. User has the responsibility of providing the solver for this ODE. Despite this FMI for ME gives much greater control over the FMU. User can access

the model in a very flexible way and hence it is ideal to be used for co-simulation. Figure 3.4 gives an overview of the data exposed by a model exchange FMU and how it may be used.

3.2.2 The High Level Architecture (HLA)

The HLA [C+00] was developed by the U.S. Modeling and Simulation Coordination Office (M&S CO). The HLA considers individual simulations at the level of processes, rather than libraries. Later sections make it clearer that the HLA provides solutions to the most common simulation interoperability problems. It facilitates a distributed environment, suitable for military simulations. It provides a specification for the development of simulation components, such that they remain usable even after the changes in the data model of the overall simulation. The data model is called the Federation Object Model (FOM).

In the HLA terminology, a distributed simulation is called a "federation". A federation comprises of several components called "federates". The HLA was designed at a level independent of any language and platform. Hence it may also be considered as a protocol. For the detailed specification of the HLA, reader is directed to [C+00]. In the forthcoming discussion the HLA is examined at the functional level and not at specification level. The Run Time Infrastructure (RTI) plays a key role in the HLA implementation. The RTI regulates individual federates. It is the central point for communication, time synchronization, event passing and data exchange. All the communication among federates must take place using the RTI. Any federate taking an active part in HLA federation should advance in simulation time only when permitted by the RTI. It should also obey all the commands issued by the RTI, for example, state updates and event handling commands.

Sufficient understanding of working of the RTI is required to completely comprehend the next coming discussion. Reader may consult [KWD99] and [DFW97] for a deeper understanding. Important services of the RTI, directly related to this discussion, are mentioned below

- **Declaration management:** These services are related to publishing and subscription of objects and attributes. Declarations of all the objects and events are done inside FOM. All federates must publish their global instances and attributes. Interested federates can subscribe to attributes of any instance.
- **Object management:** The RTI propagates the update of an object or an attribute to all subscribing federates.
- **Ownership management:** At one given time any attribute should belong to one and only one federate, otherwise there are conflicting updates.
- **Time management:** These services regulate the advances in federation time. The RTI supports many different synchronization approaches. Conservative time synchronization approach guaranties that state updates or events are passed to all federates in non-decreasing time order.
- **Event management:** Messages can be passed amongst federates in the form of events. If conservative approach is used, then RTI guarantees that no event arrives at the receiver later than its logical simulation time. Events can be used to propagate a condition to all the federates, for example an emergency alarm.
- **Data distribution management:** When there is large amount of data—including events and state updates—needed to be passed over to different simulations, then there must be some routing mechanism in place. Routing makes the delivery faster, and reduces the bandwidth requirements.

3.2.2.1 Timing Services of the High Level Architecture

In order to grasp different algorithms presented later it is useful to understand the timing services of HLA. Timing services comprise the core of synchronization. The data exchange and other services mentioned earlier are normally used invariably for a broad scope of scenarios.

3.2.2.2 Time Advance Request

Time Advance Request or TAR for acronym, may be the most widely used time advance service of the HLA. Many open source RTIs normally only implement this service. It is used for Fixed Time Stepped simulations. The idea is to advance time a fixed amount of time t_s in each cycle by each simulation. At the elapse of each t_s all the simulation components (also called as federates) are synchronized. The time t_s can also be called as "step size".The bigger the interval is the faster the simulation will proceed. Albeit, the step size is not the only thing to consider, the second main timing component is "lookahead". Each simulation must register a lookahead time t_{la} at the start of the simulation, which normally should not change during the simulation. Lookahead t_{la} is the amount of time beyond the simulation elapsed time t_{sim} a federate assures that it will not generate any event. For details of the concept reader may look at [Fuj00]. Normally the t_{la} is greater or equal to t_s. Lookahead becomes more important when the federates do not want to step ahead in time with equal step sizes t_s. In this case the synchronization process becomes more complex. With lookahead value equal to 0 ($t_{la} = 0$) and variable step size the simulation may end up hanging up at some stage. Lookahead time also makes the simulation process more efficient.

The ordinary simulation cycle starts when a federate fed_1 asks for a time advance t_0 by calling TAR service. Once the time advance t_0 is granted federate is not allowed to generate any event at or below the time $t_0 + t_{la}$. After sending some messages to other federates, federate fed_1 goes into a state where it can only receive messages. During this state it issues a call to TAR service for time $t_1 = t_0 + t_s$ repeatedly, but the time advance is not grated until it is safe to proceed. It will be safe to proceed when the RTI knows that there will be no more messages from other federates for federate fed_1 below t_1. In this way the simulation cycle proceeds.

3.2.3 Time Advance Request Available

Time Advance Request Available (TARA) service is specifically used for zero lookahead based simulations. Zero lookahead based simulation is needed when there is a situation that needs repetitive message passing at an instant of simulation time. The technique is useful to simulate partitioned differential equation systems. In contrast to TAR service a federate fed_1 calling TARA can send and receive messages on time t_0, where t_0 is the time requested from the RTI using TARA. The federate fed_1 cannot send and receive messages below t_0. Another difference from TAR service, federate fed_1 can call the TARA service for t_0 multiple times. So the message exchange among federates can continue indefinitely at t_0, until a federate calls TAR service. When a federate calls TAR service then it assures RTI that it will not generate any event at or below t_0. The time advance beyond t_0 will only be granted when all the federates call TAR and hence assure that there is not going to be any event generated by any federate at t_0. After this the simulation cycle proceeds.

3.2.4 Next Event Request Services

When using Next Event Request (NER) and Next Event Request Available (NERA) services a federate does not asks for time but instead it asks for the next event enqueued for it in the message queue. Due to requesting for next event rather than next time advance the federate does not know in advance what time it will be granted to proceed, when the time advance will be granted. This is very useful for variable step size simulations and discrete event simulations. The relationship of NER and NERA is the same as TAR and TARA. Using NERA federates can communicate a certain instance indefinitely, just like the TARA service.

3.2.4.1 Flush Queue Request

Flush Queue Request (FQR) service of RTI was originally proposed for the purpose of optimistic simulation algorithms [KWD99]. Time advances in optimistic simulations rely completely on the federates. The RTI just provides the delivery of the messages in a manner called Time Stamped Order (TSO). TSO means that the messages from one federate to any other will always be delivered in a non-decreasing time stamped order. This assures that if a federate receives a message from any other federate at a certain time stamp, then that federate will not send any other message whose time stamp is less than the previously sent messages. In later sections there will be some examples of the usage of FQR service.

3.3 Using HLA and FMI in Collaboration

Chapter 1 motivated the necessity of a simulation framework which should have following attributes

- Supports interoperability among different simulation packages.
- Is able to add parallelism, preferably using distribution.
- Supports hybrid simulation

Here it is argued that using FMI and HLA together can provide all these functionalities. For this the capabilities of both standards (HLA and FMI) are examined while considering the opportunities to realize the desired framework.

3.3.1 The FMU-Federate

The executable part of an FMU is a software component in form of a shared library. A shared library cannot replace a standalone process, but still it is possible to write code which can host an FMU as a process. To write a generic code which can convert an FMU shared library into a process requires generic programming techniques to be used. Second important question is, how should the standalone process interact with the outside world, or with other similar processes? The answer is, by using the HLA standard. Enabling the processes hosting FMUs to communicate using HLA standard makes them capable of using best of both worlds, the HLA and the FMI. We call such a program an “FMU-federate”. Using HLA terminology, a federation of such federates should be called an “FMI-federation”. Referring back to IRMs mentioned in the section 2.1, we can say that if an FMU-Federate fully obeys the rules set by the HLA then it also supports the IRMs supported by the HLA [APE+13].

Advantages of generically hosting FMUs as HLA compatible federates are following.

- The complexity of a real world system can be great. Generating an FMU from a simulation package abstracts away the complexities. Creating a generic wrapper guaranties the integrity of a complex simulation component.
- Debugging of an FMU becomes much easier because all test cases can be checked easily in isolation by just providing the desired input trajectories.
- Process separation provides easy mechanism for distribution. Simulation components can be hosted on remote machines easily. In absence of a standard interface, a master importing an FMU must also implement an interface for remote procedural call.
- Conforming to HLA implies that the communication among simulation components can occur on all protocols and media HLA supports. Different architectures supporting these mediums become easily usable, including clusters and clouds.
- Separation of communication layer enables new communication technologies to become available much more easily. With the traditional importing technique of FMI, new communication schemes would also require to change the importing code of the master.

3.3.2 The FMI-Federation

In order to orchestrate a simulation federation using FMU-Federates few capabilities must be present in the coordinating component. Here the HLA RTI is used as that component. Following are few capabilities present in the RTI which makes it a very suitable choice as a FMI-Federation orchestrator. The code which converts an FMU into FMU-Federate should be able to respond to these RTI services properly.

Declaration management: From solely the perspective of an FMU it is clear that when they form a federation they must share among them. An FMU has inputs and outputs. It is clear that in a publish-subscribe model the outputs should be published and the inputs should be subscribed.

The FOM provides the publish-subscribe model. The problem here is how to define the inputs and outputs of FMU-Federates in a FOM. One possible way is to add additional information to the model-description file, which covers all the information about which input is bound to which output variable. The problem is that the model-description file represents a single FMU, but FOM represents a complete federation. Putting global information into model-description file is not only misplaced but it will also require to add redundant information into different model-description files. From this it is clear that the creation of FOM itself should be separate from the model-description file, but there are other possibilities to put model-description file to a good use.

A simulation modeler is the person who should know that how different FMU-Federates are connected with each other by input output relationship. Based on this information the modeler should design a FOM. Based on that FOM each FMU must know which state variable of the FMU corresponds to an attribute in the FOM.

The connection of FOM attributes with the state variables of FMU-Federates can be coded into model-description file. Although, currently it is performed using command line parameters. Based on the command line parameters the subscription and publishing of the attributes is automatically performed in the code.

Object management: There are situations where complex objects may need to be shared among the federates, yet currently FMI does not support setting or getting of complex objects. Each state variable

should be considered on it own. Despite the fact the HLA FOM supports every type of an object in a federation, it may not be needed in a FMI-Federation.

Due to the flat structure of FMUs there may not arise any complexity in FMU-Federate code regarding managing the objects in a federation. The straightforward mapping from attributes to state variables allows the straightforward management of updates provided by the RTI. The hosting code however should perform the activities in a generic way that does not depend on the structure or type of FOM or the state variable of the FMU. As mentioned before complex command line parameters are used to elaborate the mapping for hosting code.

Ownership management: Ownership management is mainly a problem when there are many different objects. In a flat hierarchy where attributes simply represent a state variable in some FMU-Federate, ownership management is simple. Although, the simulation modeler must ensure that one attribute should only be published by a single FMU-Federate. Subscription by many attributed is not a problem, rather is needed in many real life scenarios. It is hard to imagine a cyber-physical simulation where the ownership of the objects is absolutely essential to be transferred. Due to this fact and based on the flat view provided by the FMUs it is assumed in the presented work that ownership of the objects do not change during the course of simulation.

Time and event management: FMI 1.0 for co-simulation supported events only partially. In order to use discrete event there were few problems which were not addressed. FMI 2.0 though, is much more elaborate and clear about the usage of events [BOAk+12]. It specifies many different scenarios where a discrete variable can be used in an FMU. For details of this matter reader it is suggested to read the FMI 2.0 specification in detail.

In a nutshell, using FMI 2.0 makes it very easy to identify the time of discrete events. According to the policy defined, an FMU automatically stops processing before, after or right at the moment a discrete event occurs. It is up to the importing code how to tackle the situation. In the presented implementation it is assumed that FMU progresses the time until the discrete event occurs, meaning it reports the exact time when the value of a discrete variable changes.

The HLA is designed around the concept of discrete event simulation, so there is no problem incorporating the idea. As soon as the FMU raises an event the generic code importing the FMU reports the value to the RTI with time stamp of the event. The RTI based on this schedules the next time step.

The RTI works as a middleware for the distributed simulation. The RTI provides the “declaration management”, “object management”, “ownership management”and “time management”services to all the federates over the lifetime of the simulation. The FMU provides the actual simulation services, while FMU-Federate provides a hosting services to any FMU to become part of a simulation federation. The architecture is shown in figure 3.5. Although, it should be kept in mind that the architecture is just a general description, during the course of this document slight different configurations of the same components will be presented.

3.3.3 An Experimental Case Study

The simplest possible integration of HLA and FMI can be using fixed time stepped simulation. An example can lay the foundations and examines the potentials. The test case is simulation where two balls are moving freely in a box colliding with each other and changing directions. The simulation can contain any number of simulation federates representing a ball, but in this example only two balls are used to present

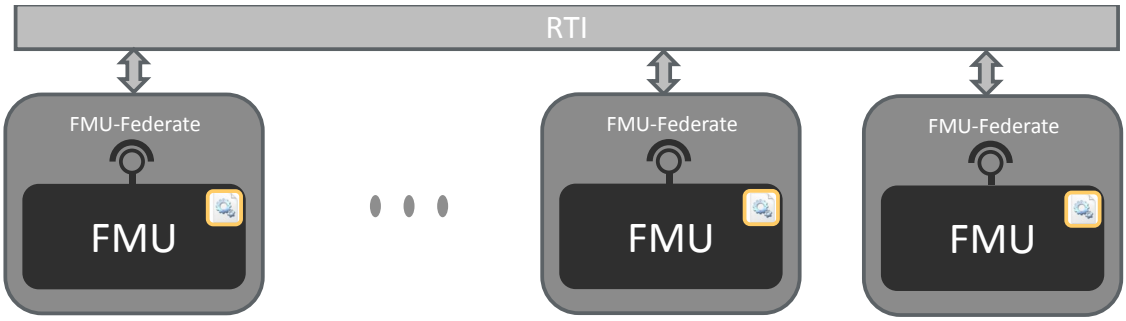


Figure 3.5: A general architecture of how FMUs, FMU-Federate and the RTI fit in.

the synchronization mechanism easily. The mechanics of a ball are modeled using an FMU. The FMU is developed using FMU SDK [64].

The object management services of RTI are responsible for the event and data sharing. The FOM structure contains a class named "BilliardBall" having attributes "x" and "y". Both attributes "x" and "y" represent a ball's position at a specific time. Each FMU-Federate creates an instance of this class managed by the RTI. Each FMU-Federate also subscribes to both of the attributes because it needs updated information of all other instances, in order to check whether there is a collision or not. Boundaries of the frame where the balls are moving are passed as parameters to the FMUs. The figure 3.6 shows a sample run of the application.

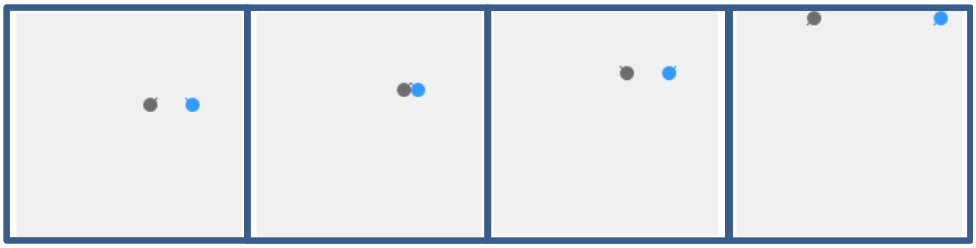


Figure 3.6: Billiard balls moving in different directions, colliding and changing directions. Images from left to right show how the system is evolving.

Collision detection occurs at every ball. The model of collision detection is quite straightforward as there are only two balls. If there had been more than two, then there could be some additional complexity. Each ball publishes its x and y position, and subscribes to the x and y position of the other one. Similarly, each ball publishes its direction of movement and subscribes to the direction of the other. The direction is represented as a vector of two elements. Each element can be either 1 or -1, indicating whether the ball is moving in positive or negative direction of each of the **x** and **y** axes. At the end of each time step each ball checks whether its position overlaps with any other ball or the frame boundary. If the answer is positive, then it calculates its new direction based on its own direction of movement and the direction of the other ball.

For time synchronization and data exchange a straightforward algorithm is used. Each FMU-Federate works independently following the algorithm shown in figure 3.7. By abiding to the presented algorithm, time synchronization and data integrity is automatically achieved with the help of the RTI.

It is beneficial to mention that the algorithm shown in figure 3.7 is not an ideal algorithm to be used in continuous simulations. The shortcomings of this algorithm will be discussed shortly in the following

sections.

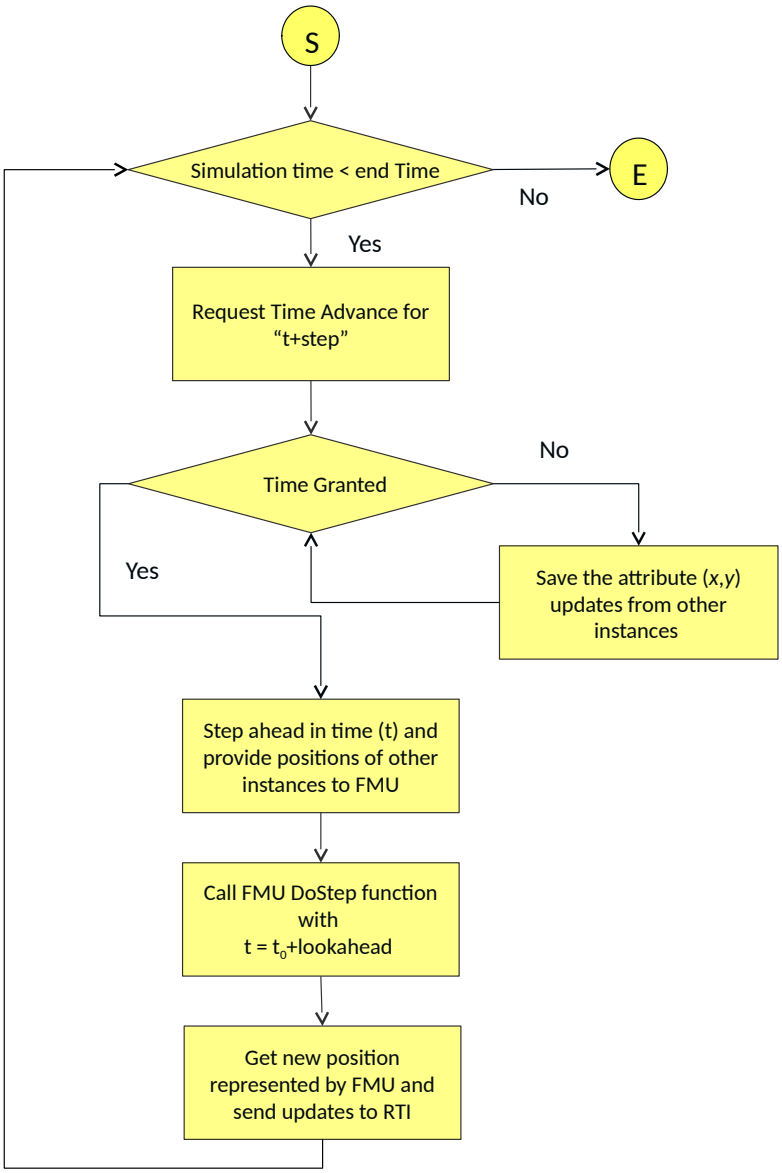


Figure 3.7: Time synchronization algorithm

In case of fixed time stepped simulation, time synchronization is easy. The only thing that has to be assured is the proper "Time Stamp Ordered delivery" (TSO delivery) of updates and events. TSO delivery of events and updates means that if any event is generated, or a change in any of the shared objects is made by an FMU-Federate at a time t_0 then the event or the updated value will be delivered to all other FMU-Federates, before any of them have reached simulation time greater than t_0. Also any two events (or updates) e_0 and e_1 generated at t_0 and t_1, where $t_0 < t_1$, e_0 will always reach at the target FMU-Federate before e_1.

The time stepped simulation with a non-zero positive lookahead makes time management easy. The "lookahead value" is always defined in terms of "logical simulation time" (or shortly logical time) of a federate. The "logical time" of a federate is the time advance it asks from the RTI using Time Advance

Request (TAR)[1]. If logical time of an FMU-Federate is t_0 and it has a lookahead value of t_{la} then by issuing TAR at t_0 it promises to RTI that is will not generate any event before $t_0 + t_{la}$. When the lookahead value t_{la} is same for all federates and is also equal to the time step t_s then the synchronization is straightforward. As when a federate issues TAR at t_0 then it means that the RTI can grant time to any other federate to time $t_0 + t_{la}$, when the federate asks for it. In this way lookahead value provides the opportunity of parallel execution of the simulations.

In figure 3.8 a petri net diagram is used to demonstrate the synchronization of FMU-federates for the case study. The transitions in figure 3.8 represent function calls or in automatic actions taken by the RTI. The places in the figure represent the states achieved by the RTI or the FMU-Federates. Actions or function calls have a name asking to do something, while name of states represent something is already done. The central rectangle in figure 3.8 represents the actions and states of the RTI. On each side of this rectangle are states and actions of two balls.

The yellow colored highlighted states and transitions impose synchronization. The first transition imposing synchronization is named as "Process Pending Updates". At this transition the RTI must make all those federates wait who may receive any event from others, and granting them time may mean the violation of TSO delivery principle. In other words to ensure the TSO delivery of events, in this particular case, one thing RTI has to ensure is that if t_0 is the minimum time requested by any federate then it should not grant any other federate time greater than $t_0 + l_a$. Here l_a is the lookahead value, which is equal for all federates in this particular case. Secondly, before granting time t to any federate, it should first be ensured that the time stamped events queued with time stamps less than or equal to t are already delivered to the federate. These are the two "constraints" which are ensured in the state named "Constraints Ensured". By ensuring these constraints the TSO delivery of events is guaranteed in case of Fixed Time Stepped simulation, where lookahead value and length of time step is equal for all federates.

Looking at the loop of execution in figure 3.7 and comparing it to figure 3.8, the execution starts from where both FMU-federates send a Time Advance Request (TAR) to the RTI. When RTI receives TAR it makes sure that if there are any pending updates to be sent to one or more federates then it should send them immediately. After sending the updates according to the constraints described earlier, the time advance can be granted. After the grant both FMU-Federates simulate the FMU to the time granted. There could be different methods of simulating them. In case of "FMI for co-simulation" the "DoStep()" function is called. In case of "FMI for model exchange" more steps have to be performed. In the example under consideration there is not much sense to use "FMI for model exchange", as the rules for modeling the behavior of a ball are already coded inside the FMU. In later sections though, "FMI for model exchange" will be used for greater flexibility in relatively different scenarios. After simulating the FMU, the states of FMU are updated. The updated states are propagated to the RTI. The RTI enqueues the updates and cycle ends. The updates enqueued are revisited again in the transition named "Process Pending Updates", after which the synchronization procedure starts as explained earlier.

3.3.4 HLA and FMI for Continuous Simulation

In contrast to Fixed Time Stepped Simulations discussed above, continuous simulations are more complex in nature to be coupled together. Primarily because there are numerical stability and accuracy issues in continuous simulations. Not only the time synchronization and data sharing has to be taken care of, but it should be ensured that the synchronization schemes devised do not produce any incorrect results and

[1] A detailed discussion on timing and other simulation constructs can be seen in [Fuj00].

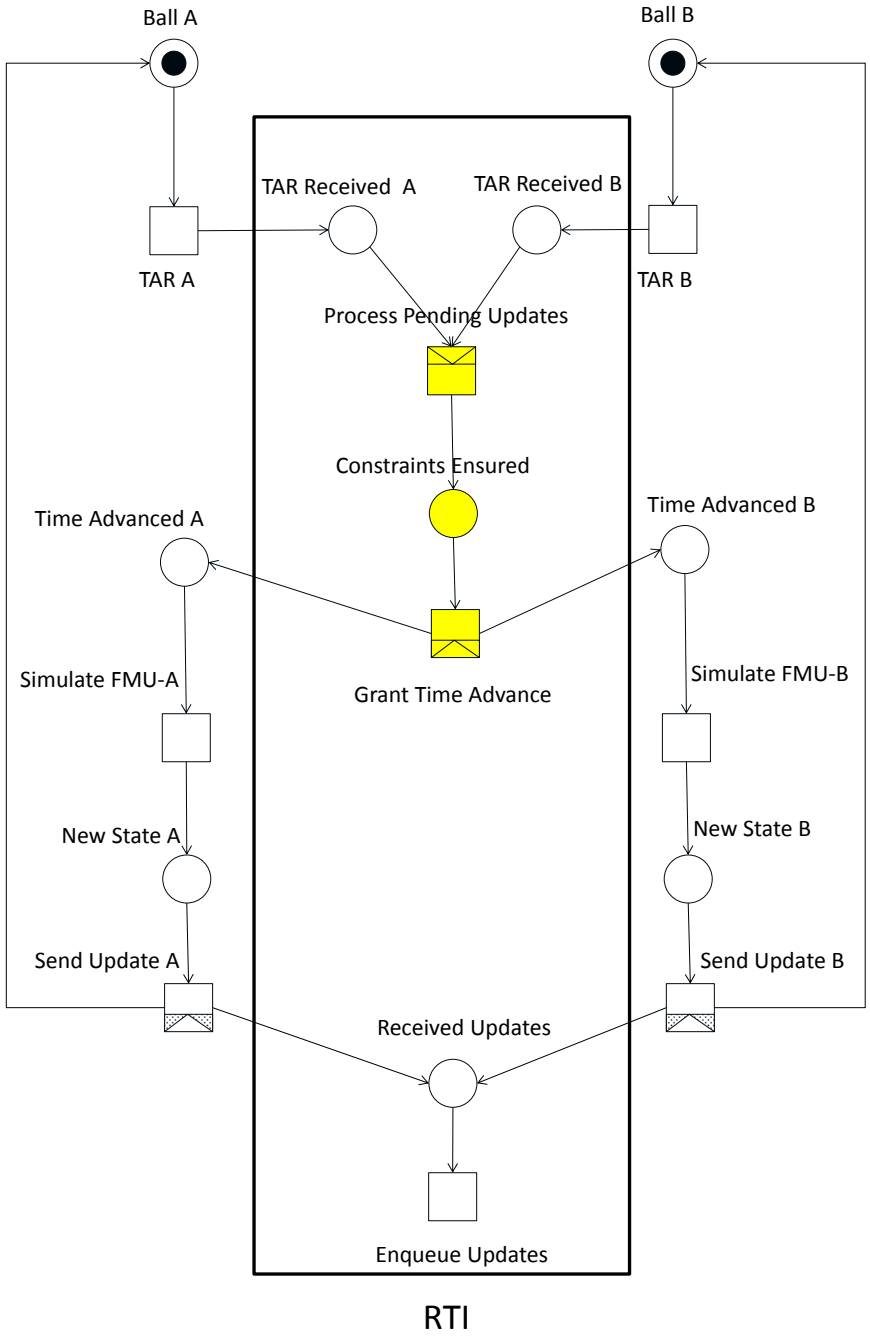


Figure 3.8: The synchronization implied by the RTI during communication with the FMUs is shown in the form of a Petri net diagram.

are numerically stable. The notion of simulation interoperation is much stronger in the case of continuous simulation.

The researchers working in the area of continuous simulation use the word “simulator coupling” for the phenomenon of “simulation interoperation”. To be precise “simulator coupling” itself represents a broad range of studies. It seems appropriate to introduce some definitions and categories of “simulator coupling”, or in other words “methods of simulation interoperation”. Introducing these definitions is also important because it puts the presented work in larger context of the research being conducted in the field.

- **Strong or Tight Coupling:** A type of coupling where the solver or the integrator for two or more models is the same, is called as "tight coupling" . Sometimes it is called as coupling using "function call", because models have to be incorporated into the complete simulation using direct function calls [Val09]. FMI for model exchange represents this type of coupling.
- **Weak Coupling:** A type of coupling where the solver or the integrator for all models are independent, is called as "weak coupling" [Val09]. FMI for co-simulation represents this type of coupling.

Here it seems useful to mention that the term "co-simulation" is also mentioned in literature for variously different phenomena. For example, sometimes it is used to represent a hardware in the loop simulation [R+94], while sometimes it is used for a simulation where more than one simulation packages want to interact with each other [Val09]. Sometimes, it is used for a simulation where two or more models have to interact with each other and both of these models have their independent internal solvers, regardless of the fact that the simulation packages are the same or not. In the field of simulation engineering the last definition of the word "co-simulation" is most popular. Referring back to "weak coupling", the definition of "co-simulation" is also similar to the definition of "weak coupling", which sometimes in literature is also called as "solver coupling" [SL14a].

As the literature suggests, for the course of this document, it seems best to use the word of "co-simulation" for "weak coupling" approach. "Weak coupling" is sometime also mentioned as "solver coupling" in literature. It should be kept in mind that "FMI for model exchange" provides to implement "tight coupling", while "FMI for co-simulation" provides easiest methods to implement "weak coupling". Although, most of the text in the document is served for the "weak coupling" techniques, yet for the sake of completion one "tight coupling" technique is also mentioned in section 5.4.1.

There are following three types of "solver coupling" techniques mentioned in the literature [BS12]. Although, the definitions stated below may be a little unclear for some readers but during the text they will become more elaborate when each one of the methods will be discussed in detail, in light of HLA and FMI.

1. **Explicit methods:** For these methods the macro-step, or the communication among the simulators at a certain point in time, does not have to be repeated. This is advantageous because it reduces communication among simulators, but these methods are less stable.
2. **Implicit methods:** Macro steps have to be repeated for these methods, requiring more communication and computation, but the methods are more stable.
3. **Semi-implicit methods:** Macro steps for these methods have to be repeated, but the communication among simulators is reduced.

A distributed co-simulation is more suitable to be "weak" because there are more than one different processes involved, and the integrator is more often kept separate. Although in later sections it is presented how HLA and FMI can be used for "tight coupling".

It was previously mentioned that FMI for model exchange" is built to support tight coupling, but it is important to mention that when a model exchange FMU is converted into an **FMU-Federate**, then it contains an independent solver or integrator, unless in a very special case mentioned in section 5.4.1. So in the presented work, weak coupling is achieved using model exchange FMUs, because co-simulation FMUs offer far less flexibility and control. Another important thing to mention is, from here onwards the complete emphasis remains on the **FMI 1.0 specification**, mainly because no open source simulation package supports **FMI 2.0 specification** when this document is being written. However, explanation of both standards is included, where necessary.

3.3.5 Timing Constructs in Continuous Simulation

Normal computers cannot do anything purely continuous, let it be simulation. There has to be some discretization applied on some level to use computers to solve numerical equations. Different fields in science came across this phenomena, so scientists devised ways to discretize the continuous signal produced by real world systems. Similar techniques are benefited by modeling and simulation community. Following are two types of discretization techniques used.

- **Time Discretization (Sampling):** Sampling may be the oldest and most used method of discretization. It divides the time into discrete chunks, taking the value of the signal at that time. Figure 3.9 explains the sampling of a signal on discrete time steps. The idea is to take values of the signal at selected times $(t_0, t_1, t_2, \ldots, t_n)$ and use them to understand the complete signal. The points of interest $t_0, t_1, t_2, \ldots, t_n$ are at equal distances, meaning $t_{i+1} - t_i = constant$.

  Similar thing happens in the case of time discretized numerical solvers. Nearly, all traditional algorithms that numerically solve differential equations, rely on this technique. Let us suppose that there is only a single state variable associated with the differential equation, then the idea is to solve the differential equation at discrete times $(t_0, t_1, t_2, \ldots, t_n)$ and then use interpolation to find the values at any other time. The distance between $(t_0, t_1, t_2, \ldots, t_n)$ is constant, when not, it is in some relation to the previous distance and the error estimate. So if the estimated error of the calculated value is η at any time step t_i then the distance to the next time step t_{i+1} is a function of the error η, i.e. $t_{i+1} - t_i = f(\eta)$. The distance between time steps is called as "step size"

- **Value Discretization (Quantization):** The idea of quantization is shown in figure 3.10. In this case the discretization is done on the value rather than the time. A representative value is selected as soon as it crosses a threshold. The time t_i at which the value passed the threshold is also important, but it does not have to be in any relation to the previous interest point t_{i-1}. In case of differential equations, the values of state variables are mostly not directly available, they have to be calculated. Mostly, only the time derivatives of state variables are available. In such cases, the solvers using quantization as a medium of discretization, have to employ much different techniques than time based solvers.

Most of the literature for solving differential equations has been relevant to time based or sampling based techniques. Currently, Discrete EVent specification (DEVS) is the only technique relying on the quantization based discretization [FCK10]. Due to the well established mathematical basis of the time based theory, in the presented work time based techniques are used widely. Although quantization is also used where appropriate, for example in section 4.2.3.

It seems appropriate to discuss the type of time discretization done in co-simulation environment. The main difference in a traditional solving of differential equations with a co-simulated method of solving differential equation is that in the former case there is only one solver (or integrator) involved, while in the later case there must be more than one solver. So there are time steps for each solver and there are time steps when all solvers should communicate with each other. The internal time step of each solver is called as the "micro time step" or simply the "time step" while the points in time when some or all solvers communicate with each other are called "macro time steps", or sometimes "communication points". Figure 3.11 explains both time steps when there are only two solvers involved.

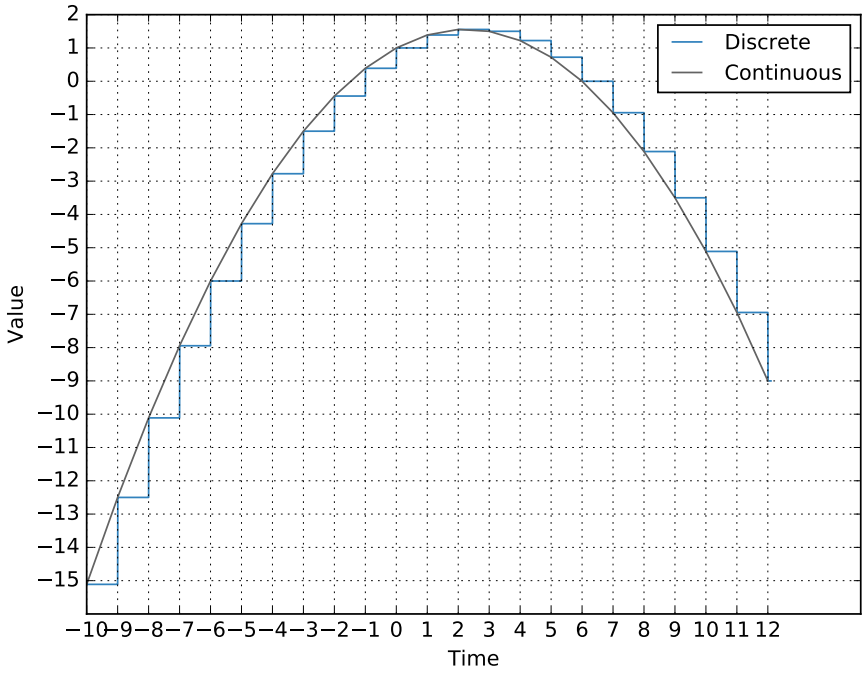


Figure 3.9: Time based discretization or sampling of a continuous signal.

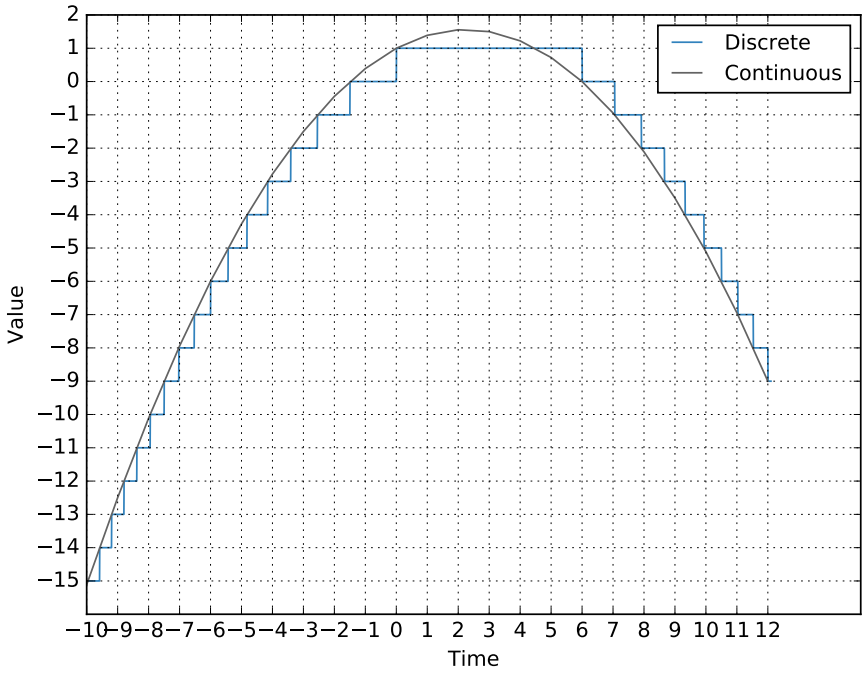


Figure 3.10: Value discretization or quantization of a signal using floor function.

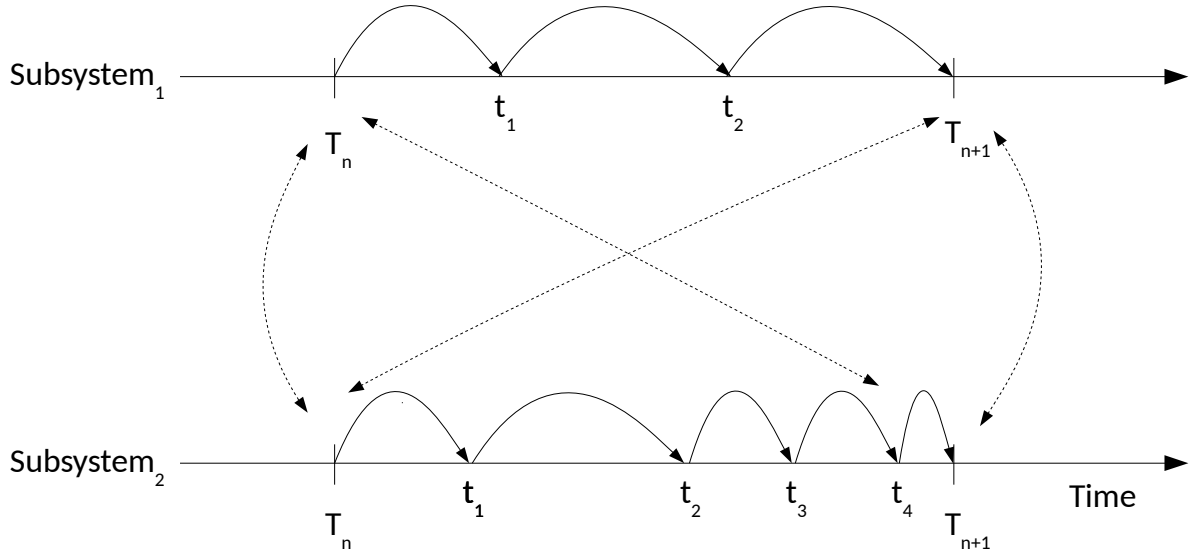


Figure 3.11: Macro and micro time steps in continuous simulation. Dotted lines show the possible paths of communication among subsystems, which can occur only with respect to macro time steps ($T_n, T_{n+1}, \dots$). Internal integration time steps ($t_1, t_2, t_3, \dots$) do not have any meaning with respect to overall system evaluation.

4 Standalone Explicit Algorithms Using HLA and FMI

As mentioned in chapter 1, the presented work looks into the state of the art mathematical concepts of solver coupling, and further develops them for distributed simulation. In the last chapter, an idea was presented that how the HLA and the FMI can be used together for distributed simulation. The idea was first presented by the author in [APE+13]. In the current chapter, explicit methods of solver coupling are discussed (for solver coupling methods, see section 3.3.4). In section 4.2 the Jacobi method is discussed, while in section 4.3 Gauss-Seidel method is discussed. Each section starts with the introduction of the method, which is part of state of the art. Based on the mathematical concepts presented in the first few lines of each section (4.2 and 4.3), following subsections develop "distributed" algorithms for solver coupling. Section 4.3 presents one algorithm and section 4.2 presents two distributed algorithms, one fixed time step based, and another discrete event based. Section 4.2 also discusses a step size control mechanism, which is not the major focus of the thesis. Section 4.2.2 presents the idea of using quantization as step size control mechanism, and its relevance to the DEVS (section 2.3) approach of simulation. All the distributed algorithms presented here have not been presented before in the research community. If any results have been published before by the author of the thesis, then the references are mentioned at appropriate places.

4.1 Introduction

Standalone algorithms are only applicable to the explicit methods of coupling. To understand explicit coupling techniques first it is essential to present the problem of co-simulation in mathematical way. When there is only one differential equation system then it is represented by equation 4.1

$$\begin{aligned} \dot{z} &= f(z, u) \\ y &= g(z, u) \end{aligned} \tag{4.1}$$

The state space representation of the same system is given in equation 4.2

$$\begin{aligned} \dot{z} &= f(z, u, t) \\ y &= g(z, u, t) \end{aligned} \tag{4.2}$$

Here z represent the state variables, u inputs, y outputs, g represents the algebraic portion of the system while f represents the differential portion. In co-simulation environment there are at least more than one

such systems interacting with each other. So each system can be numbered from 1 to n, the i^{th} subsystem represented by equation 4.3, where $1 \leq i \leq n$ and n is an element of natural numbers.

$$\begin{aligned} \dot{z}_i &= f_i(z_i, u_i, t) \\ y_i &= g_i(z_i, u_i, t) \end{aligned} \tag{4.3}$$

In result, inputs $\bar{u}$ and outputs $\bar{y}$ are connected to each other in different ways. Their connection can be represented by equation 4.4. Where matrix $\bar{A}$ contains real numbers defining how outputs of different subsystems are connected to inputs of others. It is important to mention that any subsystem shown in equation 4.3 can have one or many inputs and outputs.

$$\bar{u} = \bar{\bar{A}}.\bar{y} \tag{4.4}$$

Referring to figure 3.11 T_n and T_{n+1} are two points on time horizon. Both subsystems are being solved on the same time horizon, having separate solvers for each one of them. At "communication points" such as T_n and T_{n+1}, outputs of all subsystems are provided to the inputs of subsystems. The output to input relation is derived from the matrix $\bar{\bar{A}}$ referenced in equation 4.4. From time T_n to T_{n+1} both subsystems (here FMU-Federates) are being integrated internally, with different step sizes and perhaps different types of numerical algorithms.

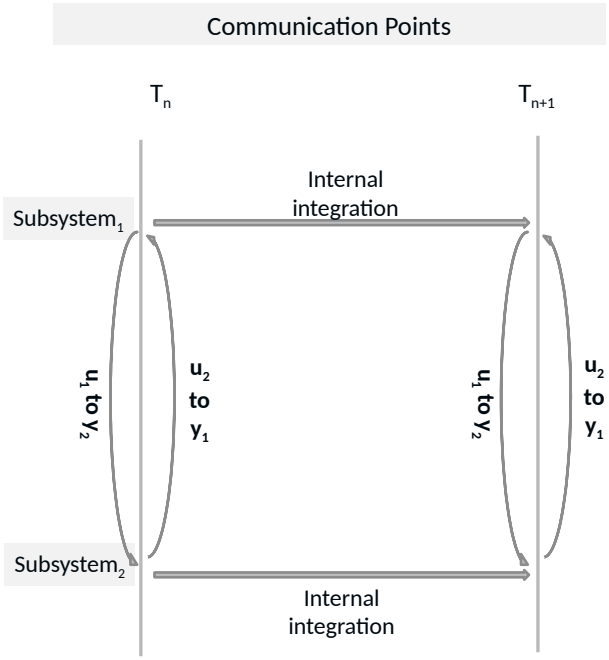


Figure 4.1: Jacobi scheme of data sharing at the end of each time step, in explicit coupling. The figure only represents the case when only two subsystems are coupled.

4.2 Jacobi Method

As mentioned earlier explicit methods do not require the macro steps to be repeated. Shown in figure 4.1 is the Jacobi scheme of explicit coupling. It is clear from the figure 4.1 that at each communication

point the input-output exchange is done only once. Once the values are exchanged they are not exchanged again at that point in time. The figure 4.1 only shows the situation when there are only two subsystems coupled with each other. If there are more than two subsystems then the same idea can be extended easily. It is a very important quality of the Jacobi method that extending it to any number of FMU-Federates is easy. Even if there are circular or complex dependencies among FMU-Federates, the scalability is not a problem. For example, figure 4.3 and 4.2 shows two scenarios of complex dependency. Subsystem S_1 provides its output to S_2, in figure 4.3 S_2 provides its output to subsystems S_4, while in figure 4.2 S_2 provides its output back to S_1. In both of these cases Jacobi method will not have any problem in solving the system.

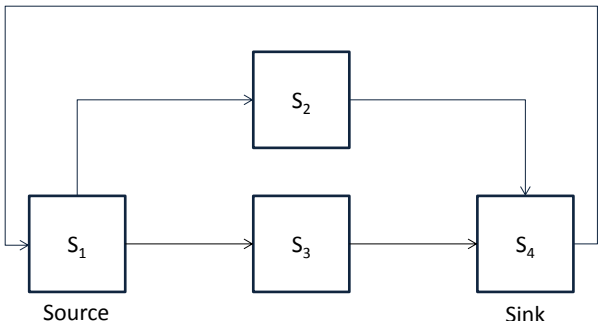


Figure 4.2: Complex dependency among subsystems, easily solvable using Gauss-Seidel method and Jacobi method.

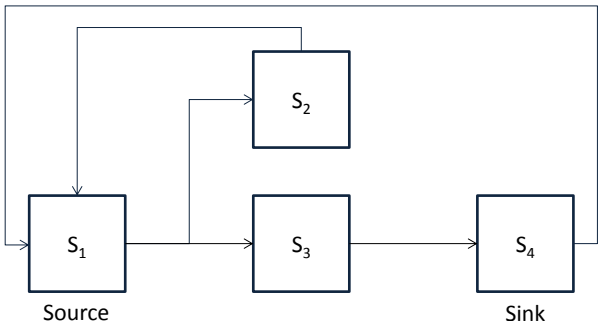


Figure 4.3: Complex dependency among subsystems, difficult to be solved using Gauss-Seidel method, but can be solved easily using Jacobi method.

4.2.1 Fixed Time Step Based Distributed Algorithm

As mentioned in section 3.3.5 the time based solvers may use a constant step size or may try to vary it based on the error estimate. If a constant step size is used then it looks appealing to use a similar algorithm as Fixed Time Stepped algorithm mentioned in section 3.3.3. Conceiving the same concept as general, it takes the shape of figure 4.4 [APM+13].

It may be termed naive to use such an algorithm when its drawbacks are examined. Figure 4.5 highlights one important drawback of the algorithm. Figure 4.5 shows a situation where one FMU-Federate has its state variable, say v_1 updated from 0 to 1, the effect of that update should take effect on the other FMU-Federate by changing value of another state variable v_2 from U to D. The process goes fine enough, only

with one problem that the two federates do not get the updates as soon as they occur, rather there is a delay in propagation of the updated values due to the non zero lookahead value. So for FMU-Federate F_1 there is a delay of at least one time step in seeing the effect of its actions. If lookahead value is greater than the time step, the delay will be even more than one time step. In solving differential equations this discrepancy can cause numerical errors. Even if there is no direct feedback, if change originated from one component is propagated to the other one with a delay then the numerical solution will exhibit errors.

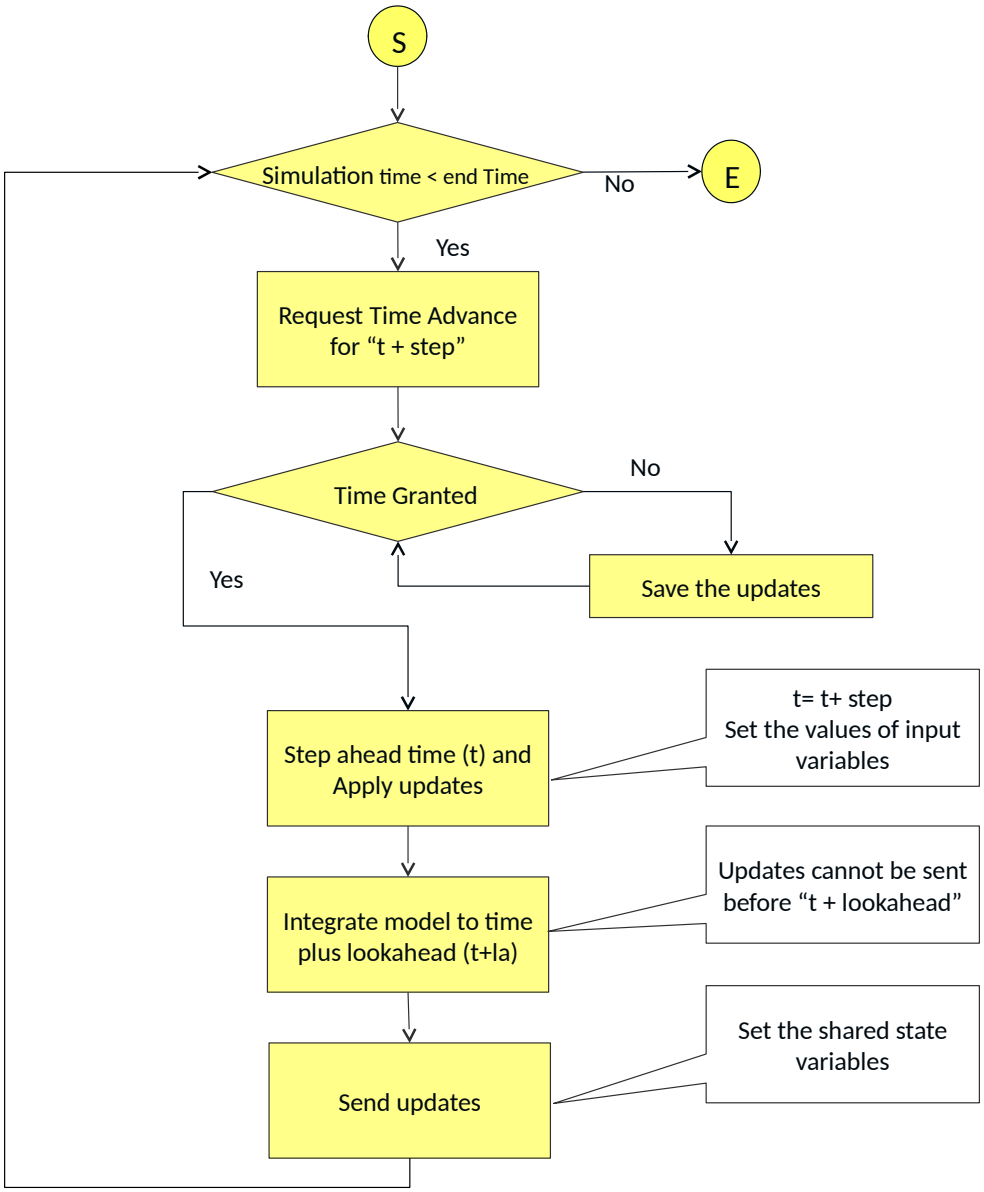


Figure 4.4: Naive algorithm is identical to the algorithm used for simulation of "billiard balls".

Figure 4.5 shows a scenario where two FMU-Federates have different lookahead values, but both of them are less than the "step size". A rather more common case is when "step size" is equal to the "lookahead" value. This situation is shown in figure 4.6, which is not much different from figure 4.5 with respect to cause and effect delay. It should be clear from these figures that if lookahead value is greater than the step size, then the delay will be even greater than one step size.

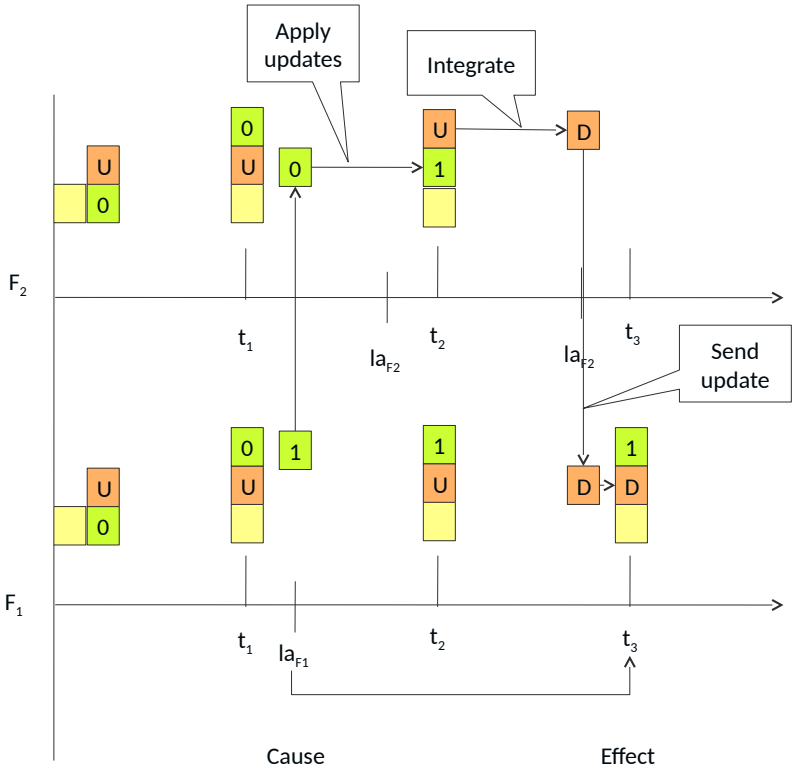


Figure 4.5: Draw back of naive algorithm.

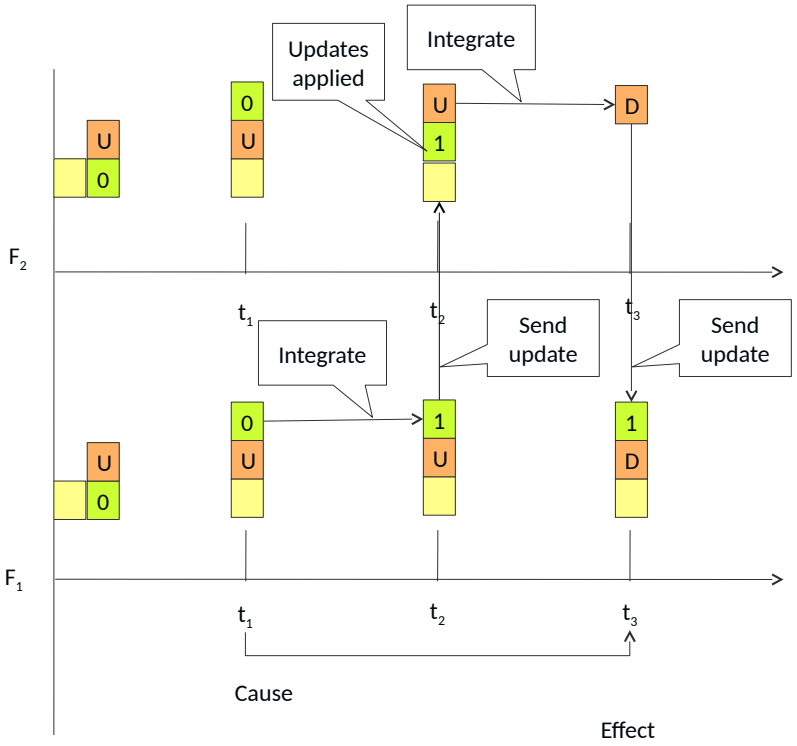


Figure 4.6: Draw back of naive algorithm with lookahead value equal to step size.

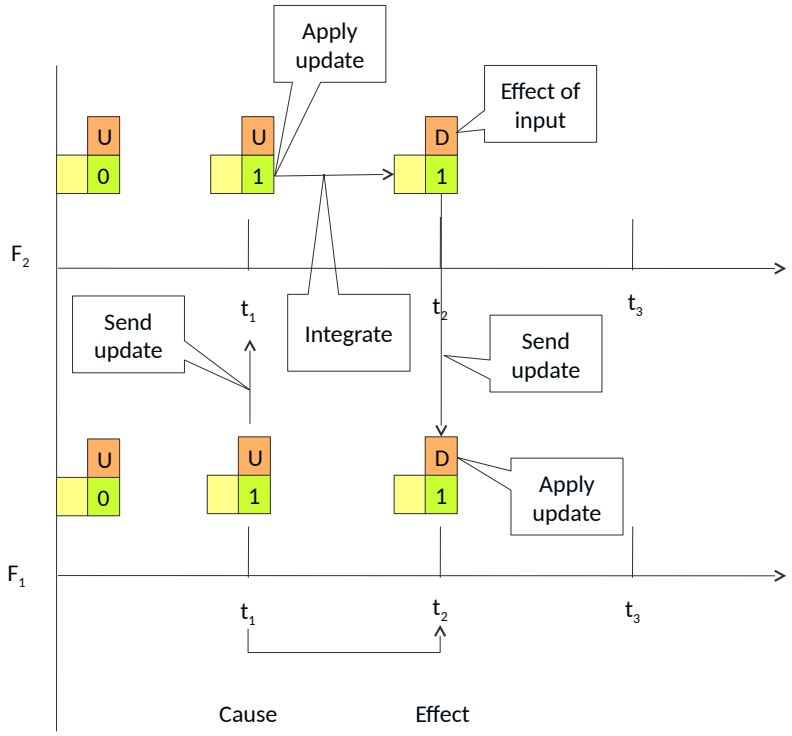


Figure 4.7: Cause and effect model in improved fixed time stepped algorithm

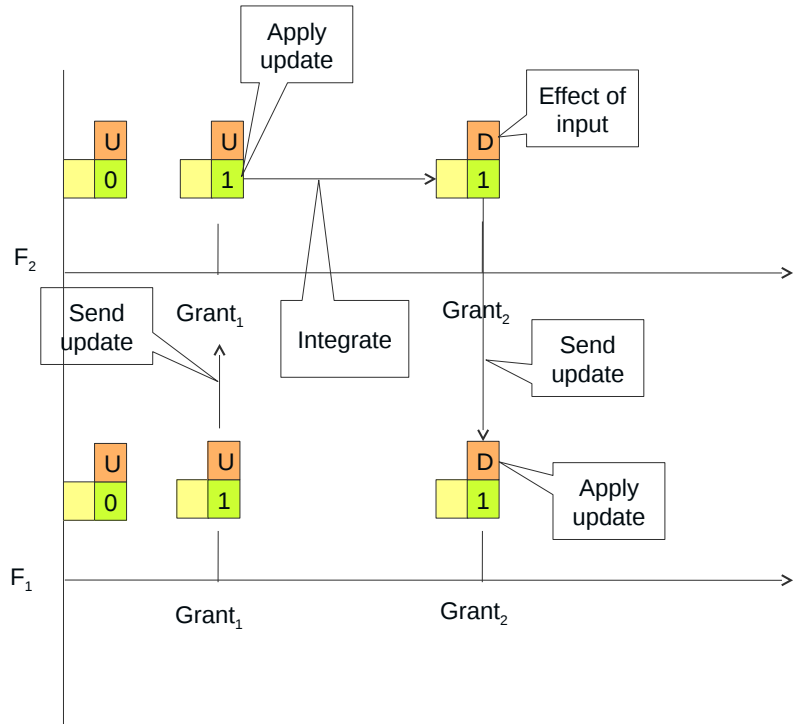


Figure 4.8: Cause and effect model in discrete event algorithm

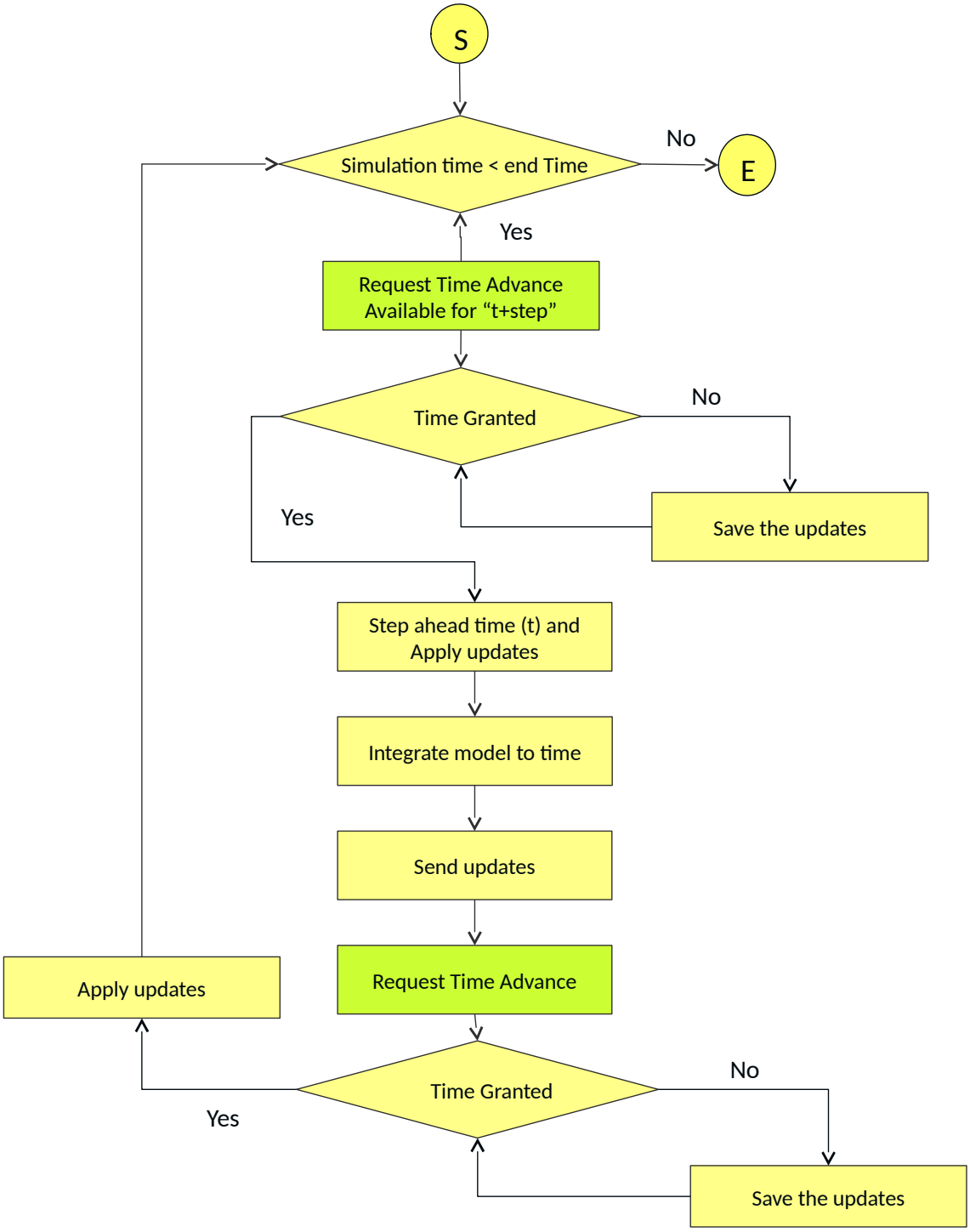


Figure 4.9: Fixed time stepped algorithm

The improved algorithm shown in figure 4.9 has two significant changes.

- It does not have a lookahead value, in this way simulation is synchronized as soon as the time step has elapsed.

- It does not use TAR service rather it uses combination of TAR and TARA service. First the TARA service is used which allows all the federates to get synchronized at the granted time. Secondly the TAR service is issued when it become clear that federate is not going to generate any other messages.

Algorithm of figure 4.9 can be named as improved Fixed Time Stepped algorithm or FTS algorithm. The improved mechanism of TARA service is very useful. First of all it does not allow to have the time lag between cause and effect as shown in figure 4.7. It also becomes important when we need to have error control or fixed point iteration for more than one FMUs integrating simultaneously.

4.2.2 Discrete Event Based Distributed Algorithm

It looks appealing to use variable step size for explicit algorithms, because step size control can bring performance improvement to the simulation. Being a separate field of study, step size control in co-simulation is not the main topic of discussion, so it is briefly discussed in later sections. For more details on step size control in co-simulation [SAC12] and [BS10a] are suggested.

One way of implementing variable step size is to use Discrete Event based algorithm or DE algorithm [APM+13]. From implementation point of view, the only difference of DE algorithm from FTS algorithm is its use of Next Event Request Available (NERA) and Next Event Request (NER) services, instead of TARA and TAR services. The algorithm is shown in figure 4.10. Unlike FTS algorithm, which advances time in fixed time steps, DE algorithm can advance with variable time steps. The cause and effect model is just the same, shown in figure 4.8, with only one difference of variable time advances. Both algorithms share the data at the end of each time step like shown in figure 4.1.

As highlighted in figure 4.10, prediction of next event is the only thing which allows to use variable step size in DE algorithm. Prediction of next event can be done in many different ways. Quantization (introduced in section 3.3.5) can also be used for prediction of next event time, which effectively is another form of step size control [CK06]. In case of coupled simulations, continuous simulations can be transformed into discrete event based simulations by using quantization. The methodology of using quantization to convert a continuous simulation into a discrete based simulation was proposed by DEVS (section 2.3) community [KJ01]. As DEVS community is doing a credible work on the topic so in detail discussion is avoided. However, the algorithm presented here does a valuable contribution of showing the opportunity of coupling DEVS based simulation components with other FMI based simulation components in a distributed topology.

In many different ways quantization can be used for predicting next event time, the simplest one though, is to use linear interpolation. Suppose during a simulation we have stored some number of past values of all output trajectories with their time. For example, an output y at time step t_n has value v_n and at time step t_{n-1} it has value v_{n-1}. As told in section 3.3.5, quantization process divides a continuous signal into discrete values such that consecutive values have the same absolute difference. For example, a floor function discretizes a continuous signal in a way that the absolute difference of consecutive values is always 1. Here absolute difference of consecutive values is named as "quantum", symbolized by q. Given the values t_n, t_{n-1}, v_n, v_{n-1} and q for output y, the time of next event t_{n+1} can be calculated from equation 4.5. Equation 4.5 uses slope formula and assumes that the slope of y is not going to change from t_n to t_{n+1}. Many other advanced techniques can be used for prediction of next event time. For example, instead of using linear interpolation, a better cubic interpolation technique may produce more accurate results.

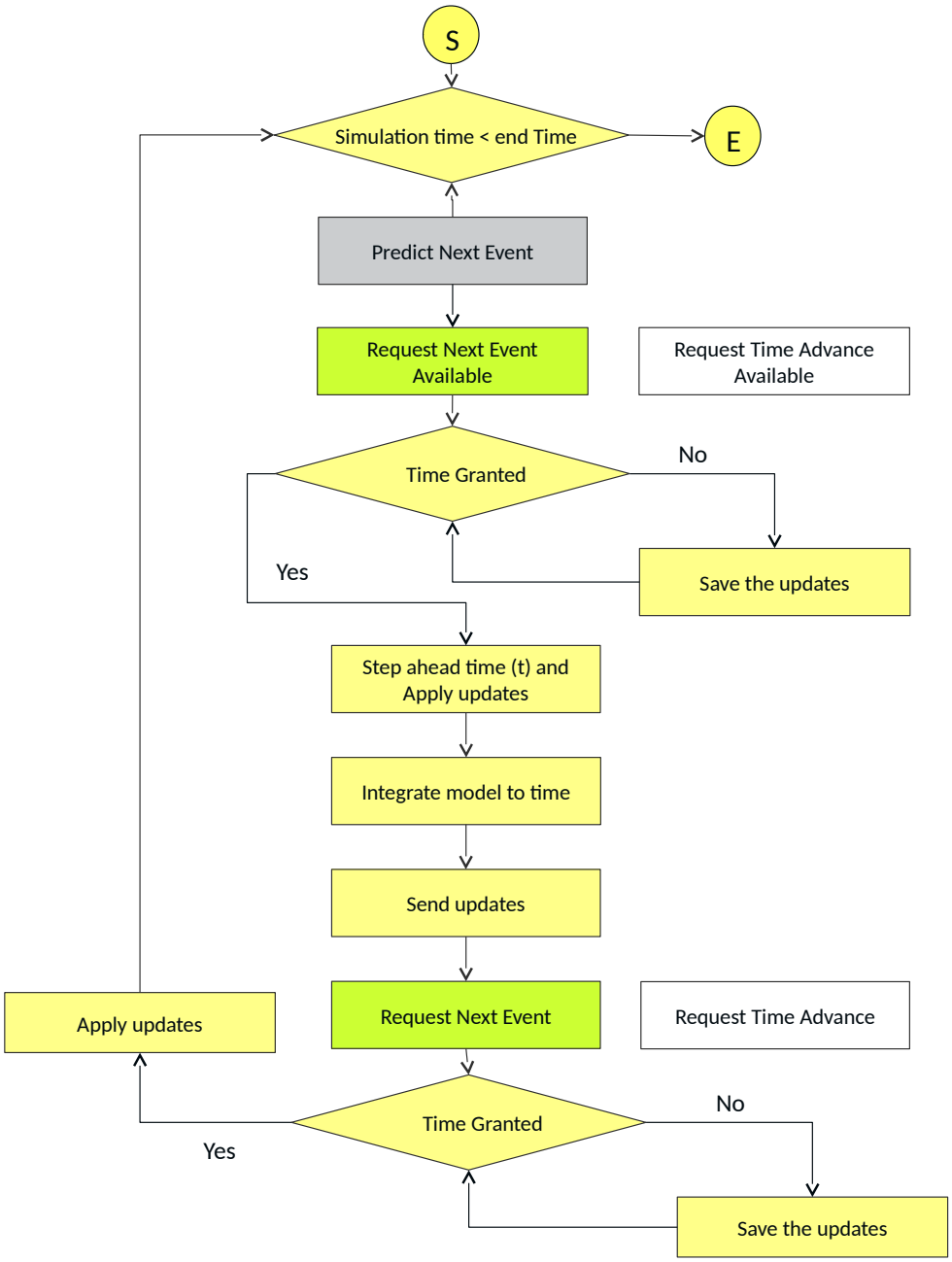


Figure 4.10: Discrete event algorithm. Gray colored state is a new state introduced with respect to FTS algorithm. White colored states show the the respective fixed time stepped states. Discrete event algorithm uses event based timing services.

$$t_{n+1} = q\left|\frac{t_n - t_{n-1}}{v_n - v_{n-1}}\right| + t_n \tag{4.5}$$

If there is an algebraic loop (interdependence of FMUs) among subsystems, then there must be at least one output of each subsystem. Considering a simple case of two coupled subsystems shown in figure 4.8, the DE algorithm uses the minimum of the predicted next event time. The mechanism is straight forward.

Both FMU-federates predict the next event time and ask for time grant using NERA, the RTI only grants the minimum of both, say F_1 had requested for the minimum of time advances t_g. As the output of F_1 is passed to F_2, so F_2 is also granted a time advance of t_g. Both systems integrate themselves till t_g and share their outputs. After exchange of outputs, both subsystems predict the next even time once again, and the cycle continues. In case of more than two coupled subsystems, the process executes the same way.

4.2.3 Case Study

To demonstrate the correctness of the above presented algorithms, a test example has been chosen. The choice was made based on the fact that the example has been tested with many different tools. See [WPE12] and [EWP12] for the details of problem and the tools used for simulating the problem. Here only a brief overview of the problem will be presented.

There are three main components of the model

1. House : Consists of volume, a heater with controller and an agent.
2. Environment temperature: A fluctuation of ambient temperature.
3. Market: Component regulating the price of electricity based on the consumption.

A “house” is a thermal system, which consumes energy in order to maintain its temperature. An “agent” is responsible for setting the maximum and minimum temperature threshold of the house, based on the price of energy. There is a heater, which is responsible for heating the house, it is controlled by a controller, which gets its minimum and maximum temperature threshold from the agent. The volume of the house has a significant effect on its temperature profile. Each house is also attached to the ambient temperature, coming from “environment”. The “environment” is a very simplified model, based on sinusoidal fluctuation of temperature. The “market” defines the price of energy based on the total consumption of the houses. Figure 4.11 shows the overview of the model. For the details of the model and the mathematical description see [EWP12].

In order to use the full functionality of the HLA RTI, there is no option but to use a commercial version of the RTI. The academic version of commercial RTIs offer a limited number of federates, hence there are only 7 houses simulated to produce results.

Significant values of the model are enumerated below, which form the basis of comparison.

1. Average inside temperature of all the houses.
2. Price fluctuation, based on the energy consumption.

Three runs of the sample simulation were executed. One with OpenModelica [59], its results were used as a reference. Second, with DE algorithm functioning as the main algorithm for all federates. Third, where all of the federates were running FTS algorithm, except the “market” federate. The “market” federate could not be used as a time stepped federate due to a limitation in OpenModelica. The FMU for “market” was not producing the correct results, presumably due to the fact that it was a piecewise continuous function.

As described earlier, all the components were first implemented and simulated using OpenModelica. Then the same components were exported as FMUs, except the “market” component. The “market” component

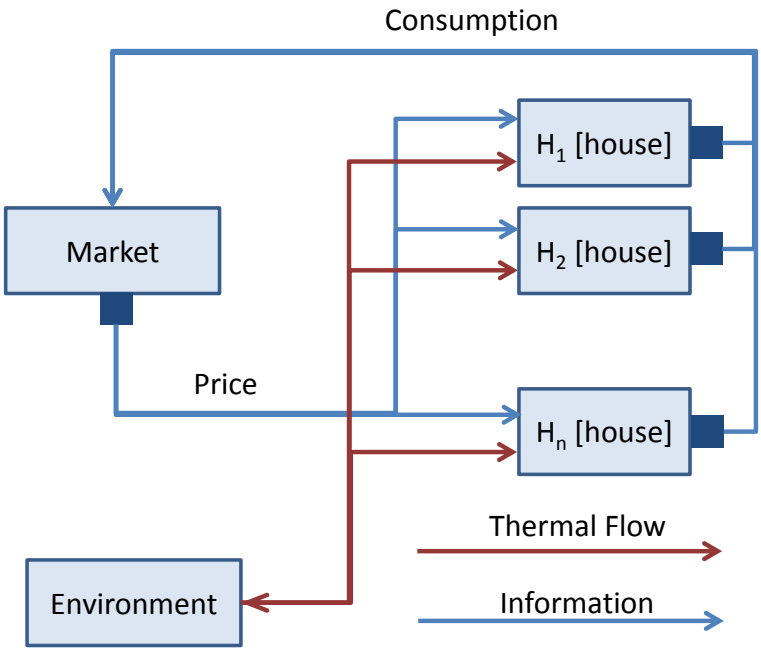


Figure 4.11: Model overview (taken from [WPE12])

took a reading of all consumptions after each 15 minutes, took the averages and calculated the new price. In the FMU generated by OpenModelica fluctuations were not occurring as they were supposed to. To replace it, a federate was implemented by hand, which acted identical to the prescribed definition of the "market".

The initial values for all the simulations were identical and fixed, so the results are reproducible. The simulation was run for 3 days of simulation time, which is $2,59,200$ seconds. One unit of simulation time represented one second. For the integration of FMUs, any step size could be chosen, even a fraction of a unit was also allowed. The communication among FMU-Federates took place after 5 seconds of simulation time, this is only valid for FMU-Federates using FTS algorithm. FMU-Federates using DE algorithm could vary their step size, so they could send updates after arbitrary time intervals. Their time step was calculated using "Quantization" mechanism. Each federate predicted the time of next event and requested for the the "time advance". The time was granted to all federates with the minimum amount of time requested. The concept is explained in figure 4.8.

Before comparing the results it is important to have a look at figure 4.12, it shows the ambient temperature. It is a simplistic sinusoidal fluctuation, which governs all the values in the simulation. It can be seen in figure 4.12 that the temperature reaches its peak at noon, when the sun is shining at its maximum. After noon the temperature starts to fall and reaches to its minimum at midnight. The temperature keeps fluctuating between 5 °C to 15 °C. Although the model is not very elaborate but it can be replaced by any statistical model.

Incorporating a statistical model into a FTS simulation could be more problematic as compare to DE simulation. Assume that the statistical profile of region is available in a dataset $\mathcal{D}$. If $\mathcal{D}$ contains the temperature information at equidistant time points, then incorporating $\mathcal{D}$ into an FTS simulation is easy. One option is to represent the time step t_s of the FTS simulation by the same interval of time I after which $\mathcal{D}$ provides the temperature information. If that is not possible, then use a time step t_s whose whole

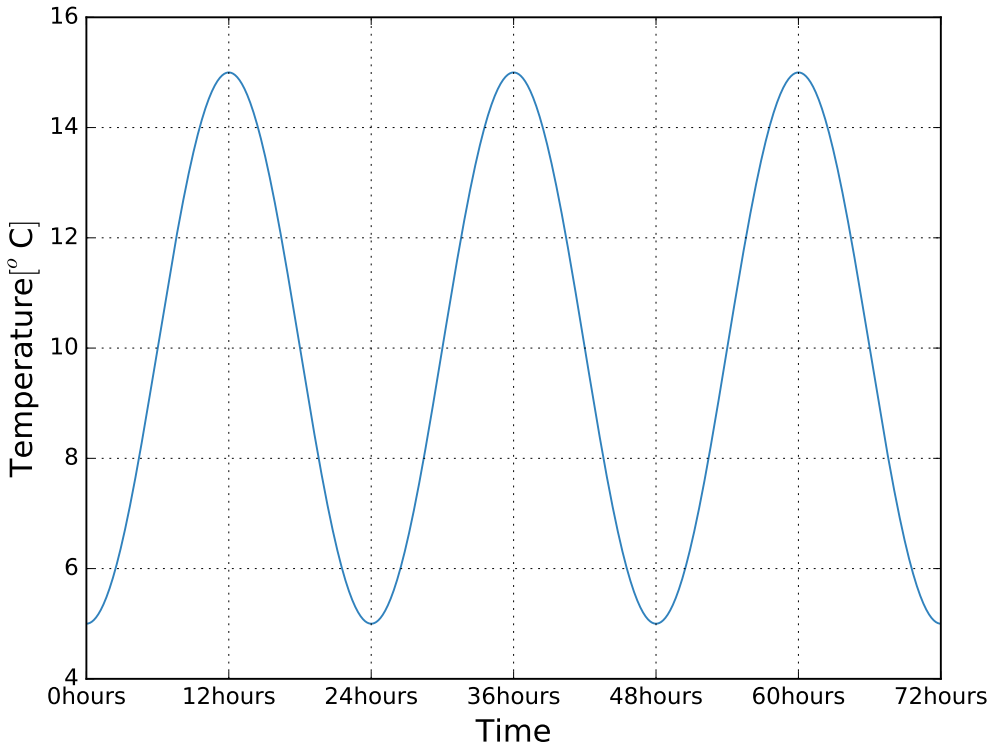


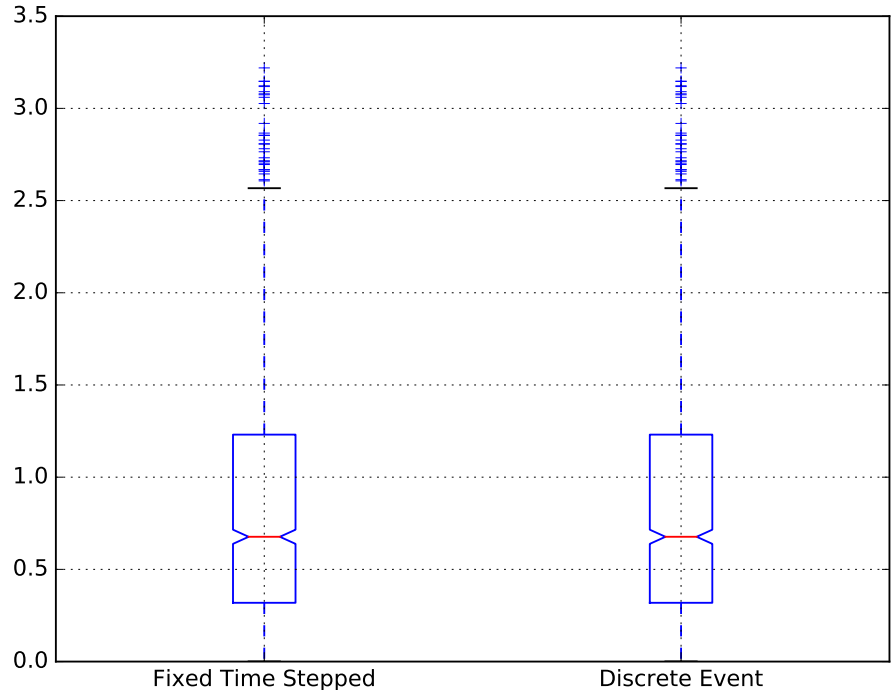
Figure 4.12: Environment profile

number multiple w can represent I, meaning $I = w \times t_s$. Alternatively, if t_s has to be made larger, then the inverse could be possible, meaning $t_s = w \times I$.

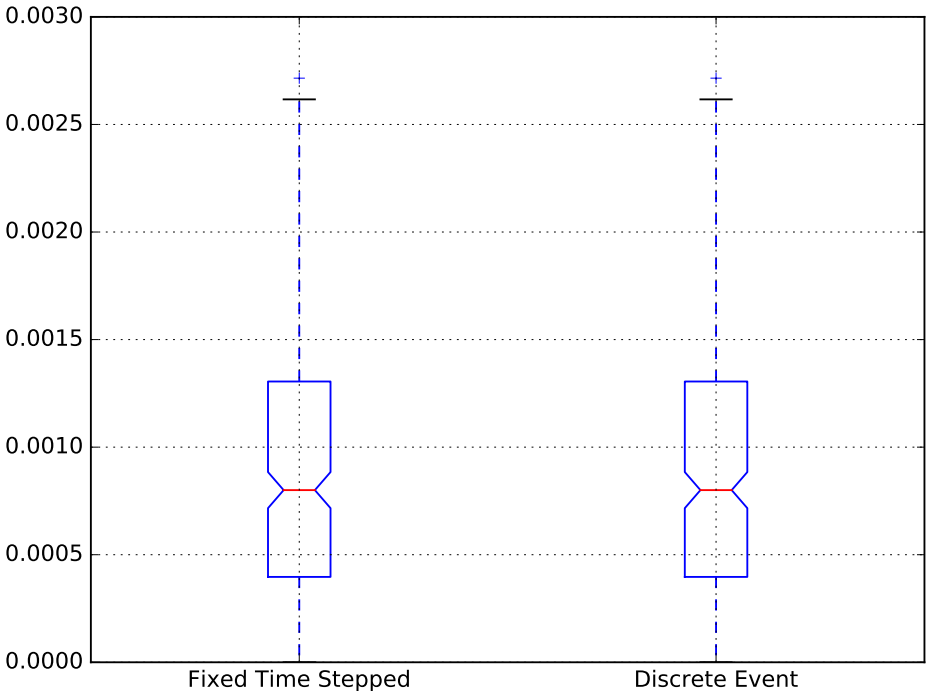
In all three figures, figure 4.15, figure 4.14 and figure 4.16, the upper most figure is for FTS algorithm, middle one for DE algorithm and the bottom one is for OpenModelica. The difference in the values of both variables are shown in figure 4.13. Figure 4.13 shows the difference in results with the help of a "box plot". A box plot graph divides any statistical data into four quartiles, using three points. The first point is the middle number between the minimum and the median of the data. The second point is the median of the data, and the third is the middle value between the median and the maximum. The dotted lines in box plot show the variability outside the upper and lower quartiles. The outliers are plotted as individual points. In the case presented here, each box plot shows the difference from OpenModelica results. In figure 4.13a the median of the difference lies at around 0.7 °C. The most probable maximum difference in average temperature is around 1.25 °C. Sometimes there is a difference of temperature up to 3.2 °C.

The difference of results for both FTS and DE algorithms are identical. This is an evidence that quantization step size control has given the best results, as changing the step size using it did not make much difference in the results. The same fact can be seen in figure 4.16, where an interval of simulation is zoomed in to show how the average temperature of houses changes during the simulation. It can be seen in figure 4.16 that both FTS algorithm and DE algorithm produce almost identical curves. The markers in figure 4.16 show the communication points among the federates. Figure 4.16b has much less number of markers than shown in figure 4.16a, which shows that DE algorithm had less number of communication points (or macro step). In fact, the number of communication points in DE algorithm were 10 times less than the FTS algorithm. In a distributed simulation this is a big achievement because more communication

among federates leads to poor performance. The relation of communication points and performance of the simulation is discussed in chapter 5.



(a) Difference for the variable "Average Temperature"



(b) Difference for the variable "price"

Figure 4.13: Difference of results from OpenModelica simulation.

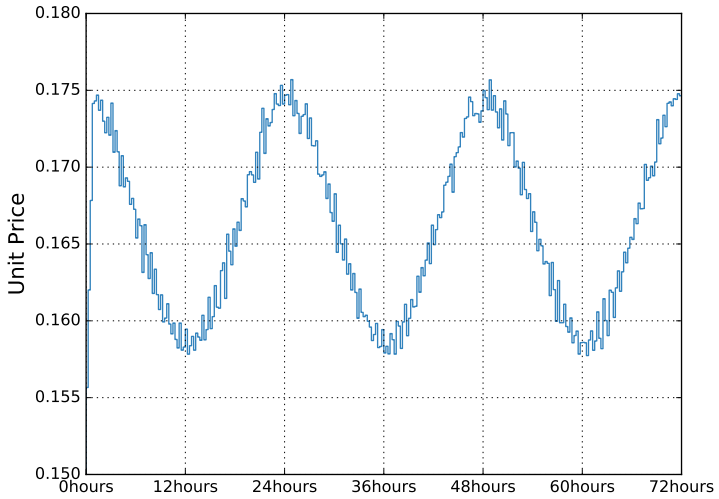
Looking at the figure 4.13b this can be seen that the differences at certain points in the results for "average

temperature", have a little affect on the calculation of the "price". The maximum difference found from OpenModelica results does not exceed more than 0.0028, which is close to 1% of the maximum unit price 0.177 (see figure 4.14). The median of the differences lies at less than 0.0009, which is an indication that most of the times the values of variable "price" calculated by DE or FTS algorithm were not different from the values calculated by OpenModelica, more than 0.0009 units.

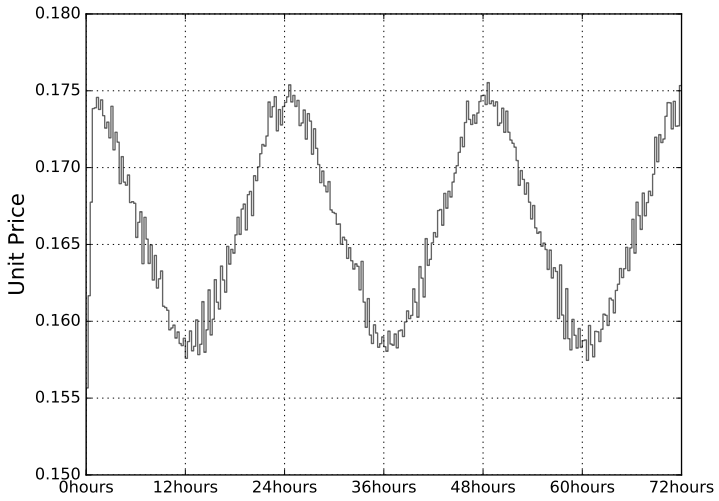
It is quite clear from the results and especially from figure 4.13 that all simulations are behaving similarly. The minimum and maximum values are also the same. In reality there was no difference in "market" component using FTS algorithm or DE algorithm, in both cases the price was being updated after 900 seconds (15 minutes), and the same code was used for both fixed step and discrete event simulations. Only the consumption behavior of the houses could make any difference. The small difference in both the results proves that there was no significant change in the consumption behavior of the houses, even with changed algorithms.

The calculation of difference in results is easy in case of variable "price", as all three simulations produced results at precisely the same time intervals. This was not the case for figure 4.15, where approximations had to be used. As OpenModelica, FTS algorithm and DE algorithm, calculated values at completely different intervals, so for the sake of comparison only those points were considered which coexisted at the same simulation time.

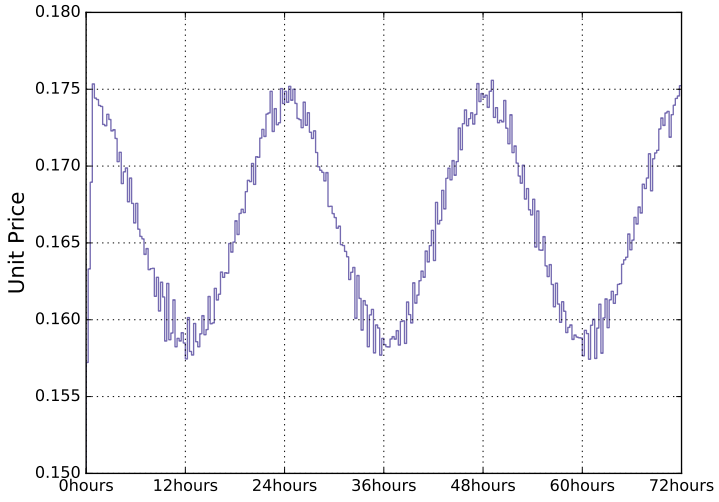
The "average temperature" profile in figure 4.15 also shows a similar behavior for all the simulations. It was a variable with relatively high fluctuations, so its similarity shows promising results with the developed technique. It can be concluded from the results that the main statistical information of the model is not changed by changing the simulation platform and algorithms used for simulation.



(a) Fixed Time Stepped Simulation

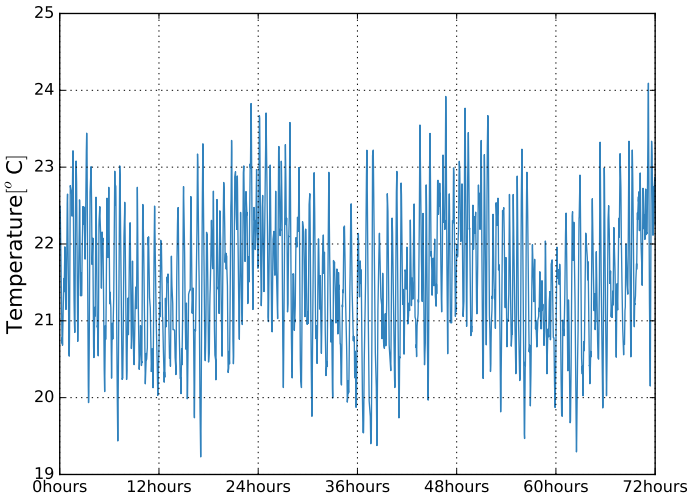


(b) Discrete Event Simulation

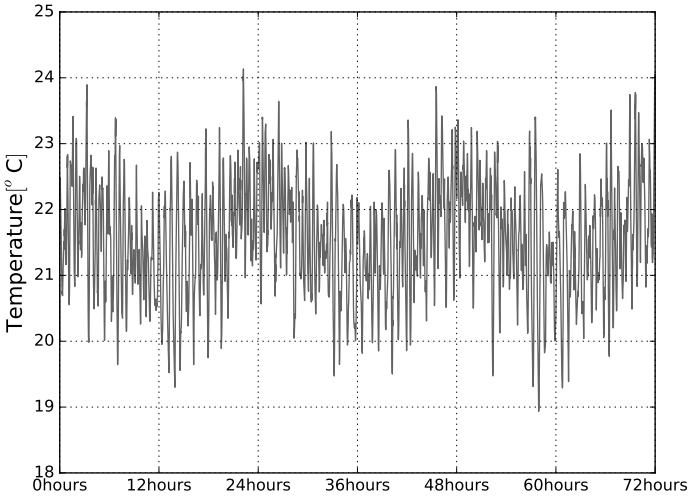


(c) OpenModelica Simulation

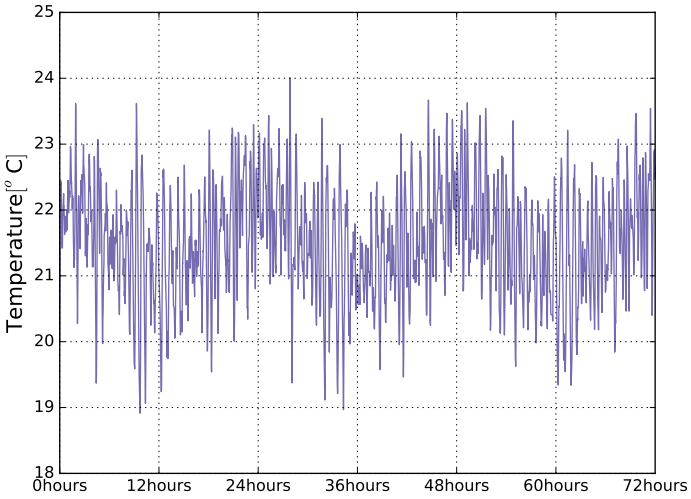
Figure 4.14: Results for variable “Price”



(a) Fixed Time Stepped Simulation

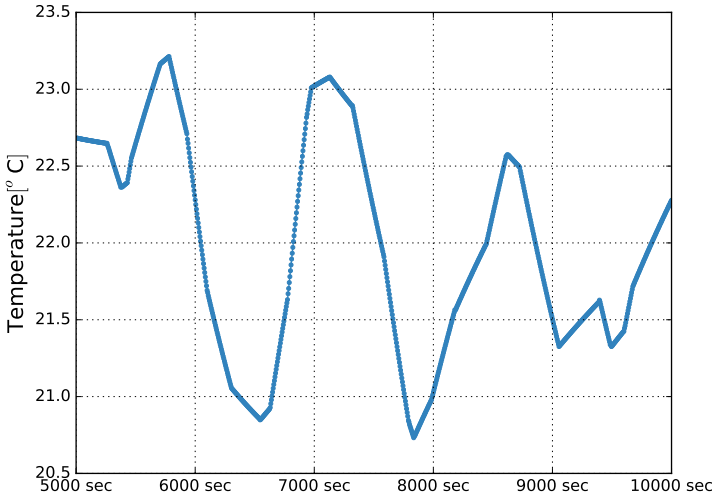


(b) Discrete Event Simulation

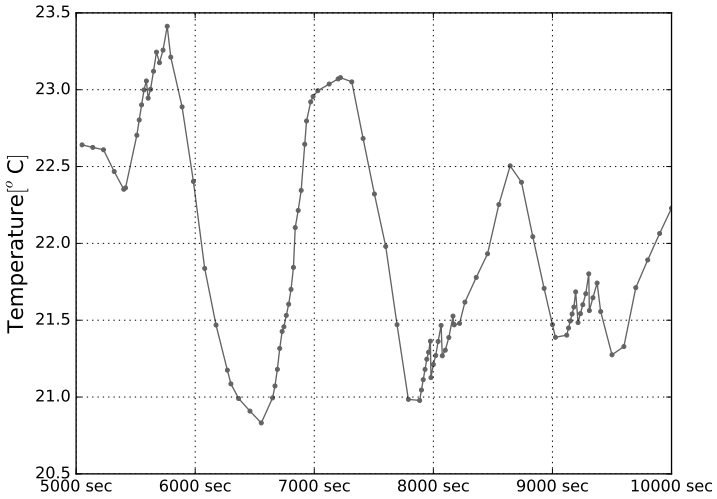


(c) OpenModelica Simulation

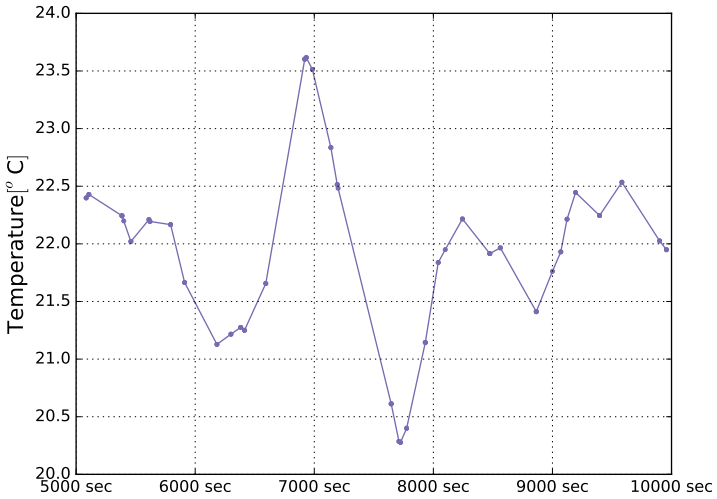
Figure 4.15: Results for variable “Average Temperature”



(a) Fixed Time Stepped Simulation



(b) Discrete Event Simulation



(c) OpenModelica Simulation

Figure 4.16: Results for variable "Average Temperature" are zoomed in at a randomly selected interval. The markers show the macro steps, their numbers clearly indicating how quantization step size control reduces the number of events (compare **a** and **b**).

4.3 Gauss-Seidel Method

Similar to Jacobi scheme, Gauss-Seidel scheme is also used in explicit coupling. In explicit schemes the values are not repeatedly exchanged at a "communication point". The main difference from Jacobi method is that in case of two subsystems coupled as shown in figure 4.17, only the first subsystem is integrated without knowing the value of its inputs at the next communication point T_{n+1}. The $Subsystem_2$ is integrated after updating the values of input variables. Unlike Jacobi method Gauss-Seidel has problems

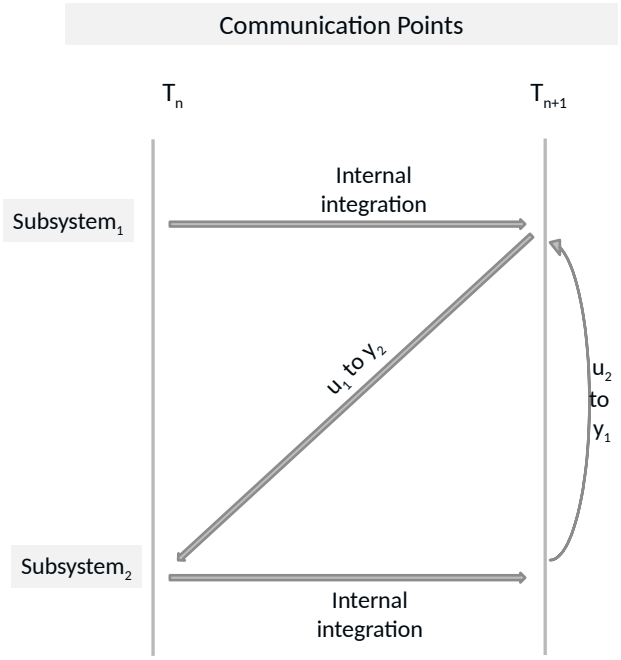


Figure 4.17: Gauss-Seidel scheme of data sharing in explicit coupling, only for two subsystems.

in scaling up. For example, figure 4.2 is relatively complex but there are no problems in using Gauss-Seidel method, because there is a unique "source" and a unique "sink" in the system topology. If the dependency from sink to source is removed, then the dependencies among subsystems form a directed acyclic graph. Also, all paths from source to sink have the same number of nodes in between. If any of these properties are missing, then the Gauss-Seidel algorithm can be used in a straightforward manner. One example is shown in figure 4.3, unlike Jacobi method system shown in figure 4.3 is not easily solvable using Gauss-Seidel technique.

4.3.1 Model of Distributed System

To explain the Gauss-Seidel scheme it is important to change the medium of description. Until now flow charts were used to describe the algorithms, but due to the relatively complex nature of the algorithms described afterwards, it is necessary to change the way of expression. In the present section, the distributed computation model is described, based on which all the subsequent algorithms are designed. As mentioned in [AW04], a system having different processes connected via a networking medium is best described as a "asynchronous message passing system". A "message passing system" is a system where different processes communicate with each with the help of messages. This is in contrast to a "shared memory system", where processes do not send messages, rather they communicate using a shared memory space.

Benefit of a message passing system is, there is no need for concurrency control in them, as compared to shared memory systems. The model of the HLA RTI is close to the representation of an asynchronous message passing system. The only difference with a traditional message passing system described in textbooks like [AW04], is the "Time Stamped Ordered delivery", or TSO delivery. In the asynchronous model there is no relation between the order of messages sent and received. Contrarily, in TSO delivery there is always a "time stamp" on each and every message sent, and the same order is retained when the messages are received on the receiver end, although the messages sent with the same time stamp need not follow any order. The RTI follows two different models of communication which correspond to the two main communication protocols namely, TCP and UDP. When "reliable" mode of communication is desired, the TCP protocol is used, while in case of "best-effort" mode the UDP protocol is used [KWD99]. Here it is assumed that "reliable" mode is used in all algorithms discuss earlier and later.

A system is considered to be an "asynchronous message passing system" when there is no fixed upper bound on the messages to reach at the destination, or there is no fixed time limit on how much time should be spent on any step. Due to these conditions, if there are two messages m_1 and m_2 sent from processors p_1 and p_2 to a destination process p_d, then there is no guarantee which message will reach the destination first, unless m_1 has a time stamp less than m_2. Each process p_i in a message passing system has q number of input queues, and output queues, to store incoming and outgoing messages to store for a limited time. So q is also the degree of p_i, meaning the number of processes connected to the process p_i. For a better understanding of the algorithms the RTI will be considered as a separate process, rather than a middleware as shown in figure 3.5, and it will be referred to as p_{rti}. It is important to remind that describing the complete procedure of the RTI is far beyond the scope of this document. It is useful to have a small description of the p_{rti} for the specific algorithm under consideration. The specific description will contain the information how the RTI is assumed to react on specific messages under consideration, according to the HLA standard, without going into the details of their implementation.

Because it is a message passing system, so while reading the algorithm it must be kept in mind that all the processes are executing in separate binaries, remote or local, and they take actions as an interrupt driven system. As the distributed processors assume an infinite execution of the processes [AW04], so in order to represent the state where they do not entertain any new messages, a special state **terminate** is introduced. Any distributed process acts in a specified manner on receiving a message. In the algorithm, a message is identified by ⟨Message⟩. A message may contain different parameters, like time of message and the value of some attributes, sometimes all these values are not mentioned in the algorithm, to avoid cluttering of information. In such a case, either this information is clear from the context, or else meaning of each parameter is explained in text.

4.3.2 Distributed Algorithm Using Gauss-Seidel Method

The distributed algorithms described in the previous sections were both fixed time step based and discrete event based. For the Gauss-Seidel method only fixed time stepped algorithms can be designed. The reason is the relatively complex dependency of simulation components formed in result of applying Gauss-Seidel scheme [AME+13]. Later discussion will make the point more clear.

Algorithms described above had a specific topology shown in figure 3.5. Each FMU-Federate runs the same algorithm and the synchronization is achieved by the guarantees provided by the RTI. Later in the document, the algorithms have different types of FMU-Federates, which work in cooperation to achieve the final goal of simulation. Explicit Gauss Seidel algorithm has three types of FMU-Federates. The "source" , the "sink" and the "common" FMU-Federate.

Algorithm 1 Explicit Gauss-Seidel Algorithm.

Initially ⟨Step Forward, *time*⟩ is enqueued for p_{source}, where $time = 0$

1: Code for p_{rti}:
2: upon receiving ⟨Flush Queue Request, *time*⟩ from p_i:
3: send all messages queued for p_i to p_i with time stamp ≤ *time*
4: upon confirmation of sending all updates send ⟨Time Grant⟩ to p_i
5: upon receiving ⟨Update, *time*⟩ from p_i:
6: enqueue ⟨Update⟩ with time stamp *time*, for all subscribing processes p_j
7: upon receiving ⟨Step Forward, *time*⟩ from p_{sink}:
8: enqueue ⟨Step Forward, *time*⟩ for p_{source}
9: upon receiving ⟨Terminate⟩ from p_{source}:
10: enqueue ⟨Terminate⟩ for all processors.

11: Code for p_{sink}:
12: send ⟨Flush Queue Request, *time*⟩ to p_{rti} and wait for response
13: upon receiving ⟨Terminate⟩:
14: **terminate**
15: upon receiving ⟨Update, *time*⟩:
16: save the ⟨Update, *time*⟩ in queue Q_{sink}
17: upon receiving ⟨Time Grant, *time*⟩:
18: **if** $|Q_{sink}| < subscriptions$ **then**
19: goto 12
20: **else**
21: $time \leftarrow \text{Proceed}(Q_{sink}, FMU)$
22: **end if**
23: Send ⟨Step Forward, *time*⟩ to p_{source}

24: Code for p_{source}:
25: send ⟨Flush Queue Request, *time*⟩ to p_{rti} and wait for response
26: upon receiving ⟨Update, *time*⟩:
27: save the ⟨Update, *time*⟩ in queue Q_{source}
28: upon receiving ⟨Time Grant, *time*⟩:
29: dequeue messages from Q_{source} and update input variables of FMU.
30: upon receiving ⟨Step Forward, *time*⟩:
31: integrate model to $time + step$
32: **if** $time + step > SimulationEndTime$ **then**
33: send ⟨Terminate⟩ to all processors
34: **terminate**
35: **else**
36: send updated states using ⟨Update, $time + step$⟩ to p_{rti}
37: **end if**

Algorithm 1 Explicit Gauss-Seidel Algorithm (Continued).

38: Code for p_{common}:
39: send ⟨Flush Queue Request, *time*⟩ to p_{rti} and wait for response
40: upon receiving ⟨Terminate⟩:
41: **terminate**
42: upon receiving ⟨Update,*time*⟩:
43: save the ⟨Update,*time*⟩ in queue Q_{common}
44: upon receiving ⟨Time Grant,*time*⟩:
45: **if** $|Q_{common}| < subscriptions$ **then**
46: goto 39
47: **else**
48: *time* ←Proceed(Q_{common}, FMU)
49: **end if**
50: **Function** Proceed(Q, FMU)
51: dequeue messages from Q and update input variables of FMU.
52: save the *time* of updates //all updates have the same *time*.
53: integrate model to *time*
54: send updated states to RTI using ⟨Update⟩ message
55: **return** *time*

It should be noted that to ensure TSO delivery every message must be sent with a time stamp. For clarity the parameter *time* is mentioned in all messages. The p_{rti} is an exceptional process, it uses the time stamp *time* of each message to ensures the TSO delivery of the message. The messages sent from p_{rti} also have the same time stamp with which they were sent to p_{rti}. Secondly, it is reiterated that the "reliable" medium of communication among processes is used for all events and updates. The exceptional path when a message does not reach the destination, is not discussed here to avoid unnecessary complexity. It is important to mention that there are lines of code which appear to send messages directly to another process. Such way of description is only adopted for brevity and simplicity, in reality no process can communicate to another process directly. All communication has to happen through the RTI.

- ⟨**Flush Queue Request**⟩: Flush Queue Request (FQR) has been explained earlier in section 3.2.4.1. Here it is symbolized that by sending this messages FQR service of the RTI is invoked. After sending this message the FMU-Federate goes into a condition where it only receives messages from the RTI, until it receives the ⟨Time Grant⟩ message. The parameter of this message is the *time* to which federates wishes to advance.

- ⟨**Time Grant**⟩: In result of FQR or any other time request (section 3.2.2.1) RTI has to grant the time. Here it is symbolized using the ⟨Time Grant⟩ message. When this message is received by the FMU-Federate, the FMU-Federate is allowed to proceed and produce messages greater than or equal to the granted *time*. This message contains a parameter which is the *time* to which federate is allowed to advance the time to.

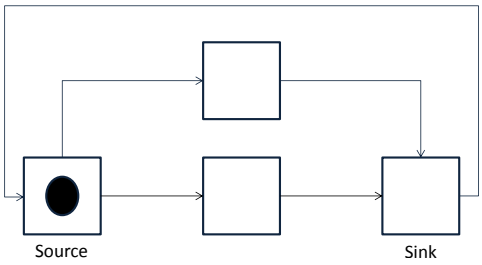
- ⟨**Update**⟩: As explained earlier FOM is used to share the data amongst federates. It is also clear that HLA works in a publisher subscriber model. So in the algorithms presented here, when the ⟨Update⟩ message is sent from FMU-Federate to the RTI then it means that it needs to update some value of the attribute, whose ownership belongs to it, in other word it has the right to publish that value. When the ⟨Update⟩ message is sent to any federate then it means that the published value of

some attribute needs to be communicated to the subscribers. The values are passed on by the RTI exclusively.

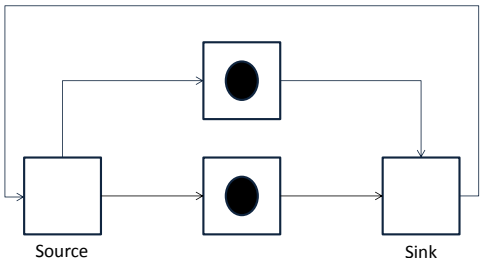
- ⟨**Step Forward**⟩: There are two types of data sharing options in HLA. One is through obects and classes, the second is called the "interactions". These are messages with some parameters. When an interaction is sent from publisher all subscribers receive that interaction. Although, the interaction always travel from origin to target through RTI, but for simplicity there is no problem in believing that origin sends an interaction to the destination. So here it assumed that p_{sink} sends the ⟨Step Forward⟩ interaction message to p_{source}, although, in reality it goes through the RTI.

- ⟨**Terminate**⟩: When the p_{source} identifies that the simulation execution has passed the desired end, it sends the ⟨Terminate⟩ message to all processors, which makes the processors change their state to **terminate**. The ⟨Terminate⟩ has a time stamp *time* only to ensure TSO delivery. It is sent only once in the life time of simulation.

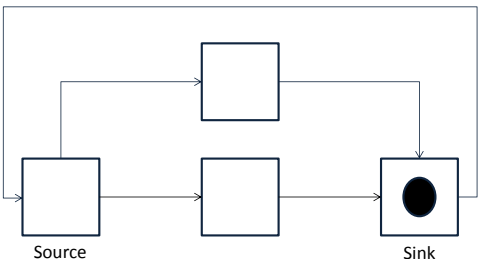
Gauss-Seidel algorithm is different from aforementioned algorithms in that it does not have the same execution order for all FMU-Federates. It has three types of FMU-Federates "sink", "common" and "source", represented in Gauss-Seidel algorithm as p_{sink}, p_{common} and p_{source}. Algorithm 1 works like a token based machine. The "source" initiates the token and the "sink" consumes it. No matter how many FMU-Federates are there in between "source" and "sink", the "source" and "sink" should always be unique. In case of more than one "source" or "sink" algorithm will not work at all. It was mentioned earlier that the system shown in figure 4.3 is not solvable using Gauss-Seidel algorithm. To see how, let us first have a look how token based execution solves the system shown in figure 4.2.



(a) Source starts the execution by integrating and sending its outputs as an ⟨Update⟩ message to the RTI.



(b) Token is passed to "commons". They can proceed in parallel as there is no dependency amongst them.



(c) Execution cycle of a time step ends at "sink", who sends ⟨Step Forward⟩ message to pass on the token to source again. The output values alone from the "sink" do not cause the token to pass.

Figure 4.18: Token based execution of explicit Gauss-Seidel method.

Figure 4.18 shows the token based execution of the Gauss-Seidel algorithm. At the first step shown in figure 4.18a, the token is originated by source, this refers to the ⟨Step Forward⟩ message already enqueued for the p_{source} in the initial stage of Gauss-Seidel algorithm (see the first line of the algorithm). At second step this token is passed to all subscribing FMU-Federates. These are "common" FMU-Federates, represented as p_{common} in Gauss-Seidel algorithm. In the second step (figure 4.18b) the token is passed due to the updates originated by "source", which are received by both "common" FMU-Federates. The update is sent by p_{source} in line 37. In line 45 p_{common} checks whether it has received all updates that it was subscribed to. In the current example, it is assumed that "source" has only one output variable to which both "common" FMU-Federates have subscribed to. As the "source" has sent the update which will "eventually" reach both "common" FMU-Federates, so they have the token of execution now. Similarly the "sink" receives the updates from both "common" FMU-Federates and sends the ⟨Step Forward⟩ message to the "source". In result "source" gets the token again and the cycle continues.

Comparing figure 4.3 with figure 4.2 and the execution order described here, it becomes clear that when the token is passed from the p_{source} to both "common" processes, one of the "common" processes S_2 will pass the token back to the p_{source} again. The token will be passed to the "source" for the same reason "sink" receives the token, i.e. the updates generated by S_2. The sink receives the token due to the updates generated by S_3. In this way "source" can pass on another token before the previous one has been consumed, resulting is erroneous execution.

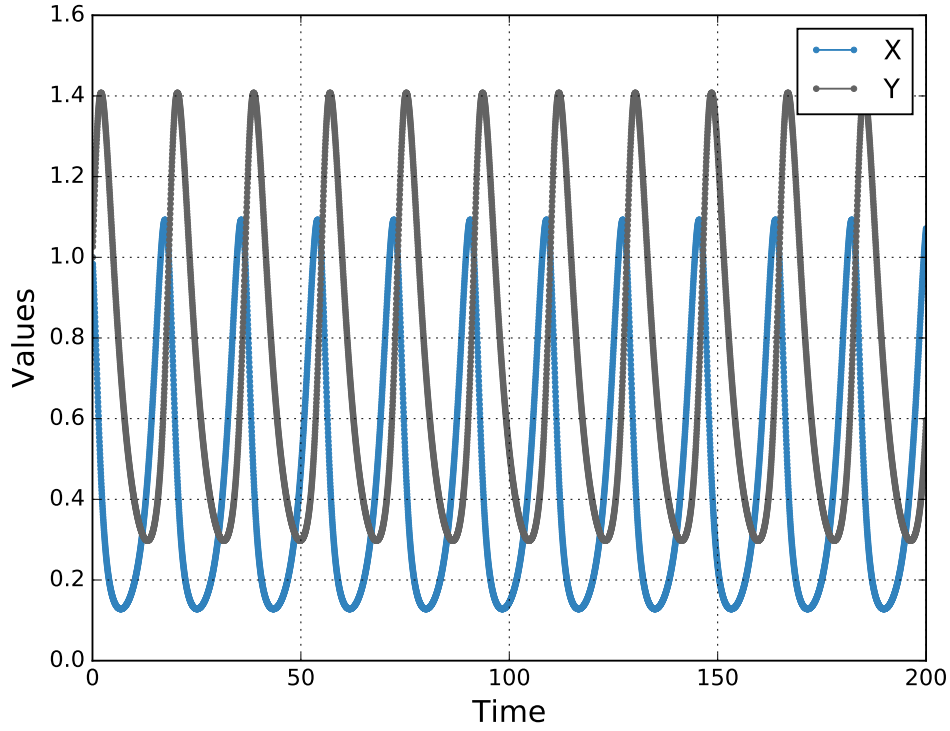
By looking at the execution cycle it is clear that there are number of limitations on the kind of systems Gauss-Seidel algorithm can operate on. First of all, there has to be a unique "source" and "sink". Secondly, if the edges are drawn among FMU-Federates from the publisher to subscriber, just like it is done in figures 4.2 and 4.3, then it should form a tree where there are should be only a single cycle, which is formed due to the feedback loop from "sink" to "source". The "source" has to be at the root of the tree, and "sink" should be the only leaf node without any children. Connection among siblings do not harm, but if there is a connection from child to parent then the algorithm will not work. This limitation can be removed by using some heuristics, but this is not discussed here, because better algorithms can be designed, which are described later.

4.3.3 Case Study

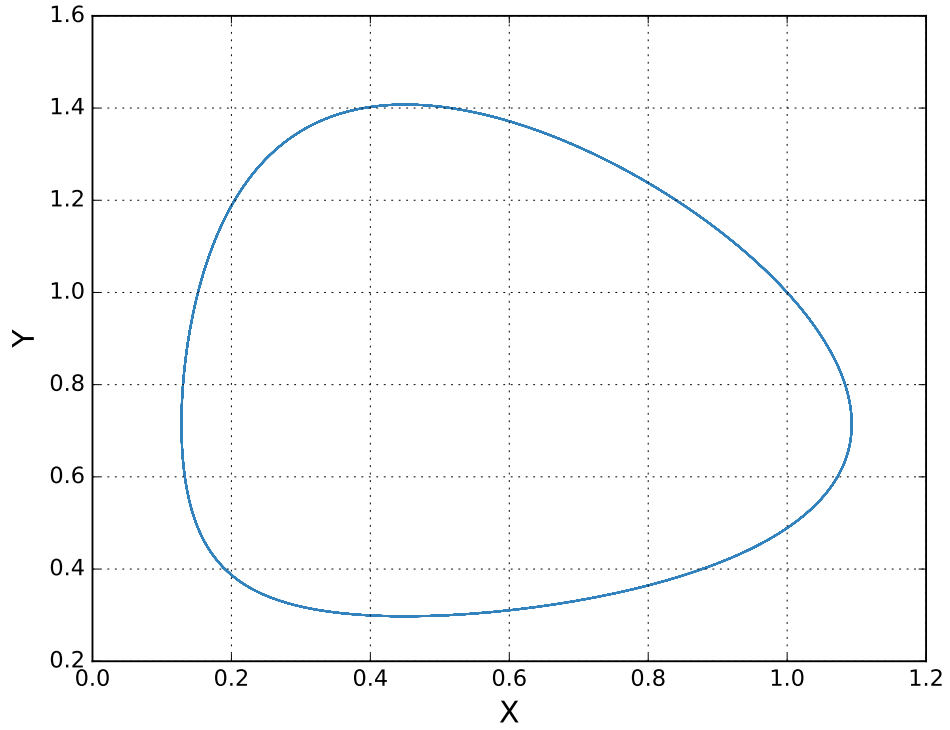
It seems appropriate to describe a case study which distinguishes the importance of Gauss-Seidel method over previously defined Jacobi method. The model is "Lotka-Volterra equation", which is also called as "Predator-Prey equation". It is described as follows

$$\begin{aligned} \dot{x} &= \alpha x - \beta xy \\ \dot{y} &= -\gamma y + \delta xy \end{aligned} \tag{4.6}$$

The parameters α, β, γ and δ are chosen to be $\alpha = 0.5$, $\beta = 0.7$, $\gamma = 0.27$ and $\delta = 0.6$. The **reference** solution for the model is shown in figure 4.19. Figure 4.19a shows the variable x and y plotted separately against the simulation time, while in figure 4.19b variable y is plotted against variable x to show the actual behavior of the "Van der Pol" oscillator. Figure 4.19b shows the egg shaped condition of the oscillator due to the parameters selected for the simulation.



(a) Variables x and y plotted separately against simulation time.



(b) Variables y plotted against variable x.

Figure 4.19: "Lotka-Volterra " system simulated using OpenModelica. The "step size" is internally controlled by OpenModelica, to which user does not have any access.

The results of the reference solution are compared to the results produced by the Gauss-Seidel algorithm. It is shown in figures 4.21 and figure 4.22 that how the results vary by just changing the method of data exchange among subcomponents. In both cases the internal solver was the same, also the step size on time horizon was also the same. In figure 4.21 it can be seen that the results are very similar to the results of OpenModelica (figure 4.19). Figure 4.22 shows the results using explicit Jacobi method. It is clear that the error is accumulating with the time, so the method is not stable for a problem like Lotka-Volterra system. The reason is hidden in equation 4.7.

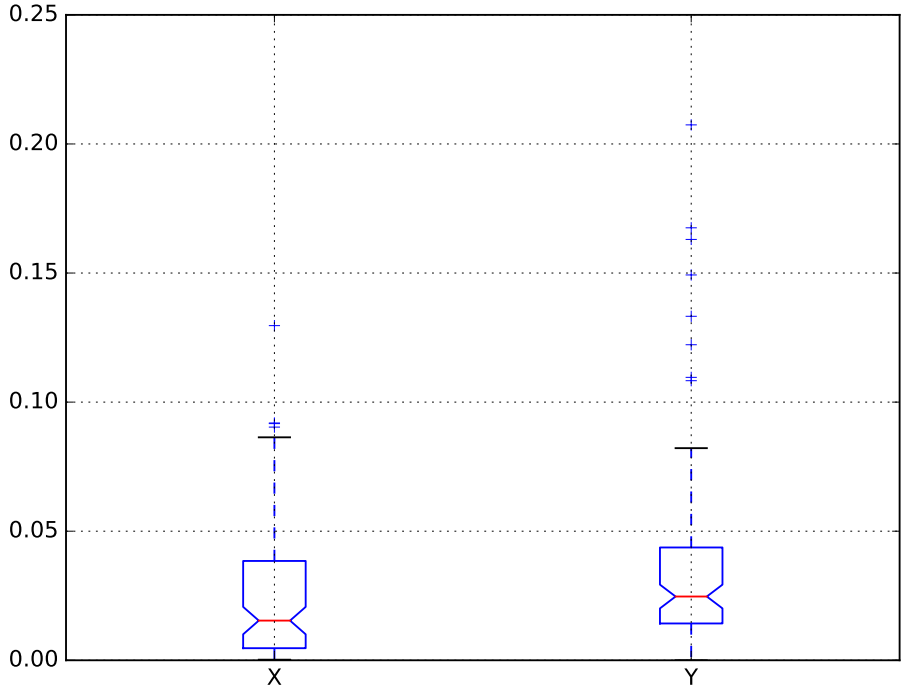
Equation 4.7 shows that the stability of the solution does not only depend on the method but also on the system[KS00]. For the detailed discussion of how the system and coupling method are interrelated to the stability of the solution it is suggest to read [BS10a] and [BS10b]. In brief if the system presented in equation 4.3 is reduced to a system with only two subcomponents, then it can be reduced to a form where solution of the system at communication point T_{n+1} is given by

$$\begin{pmatrix} \bar{z}_{n+1} \\ \bar{y}_{n+1} \end{pmatrix} = \left[\bar{\bar{I}} - \bar{\bar{\Omega}}_{n+1}\right]^{-1} \left[\bar{\bar{\Omega}}_n \begin{pmatrix} \bar{z}_n \\ \bar{y}_n \end{pmatrix} + \bar{\bar{\Omega}}_{n-1} \begin{pmatrix} \bar{z}_{n-1} \\ \bar{y}_{n-1} \end{pmatrix} + \ldots + \bar{\bar{\Omega}}_{n-p} \begin{pmatrix} \bar{z}_{n-p} \\ \bar{y}_{n-p} \end{pmatrix}\right] \tag{4.7}$$

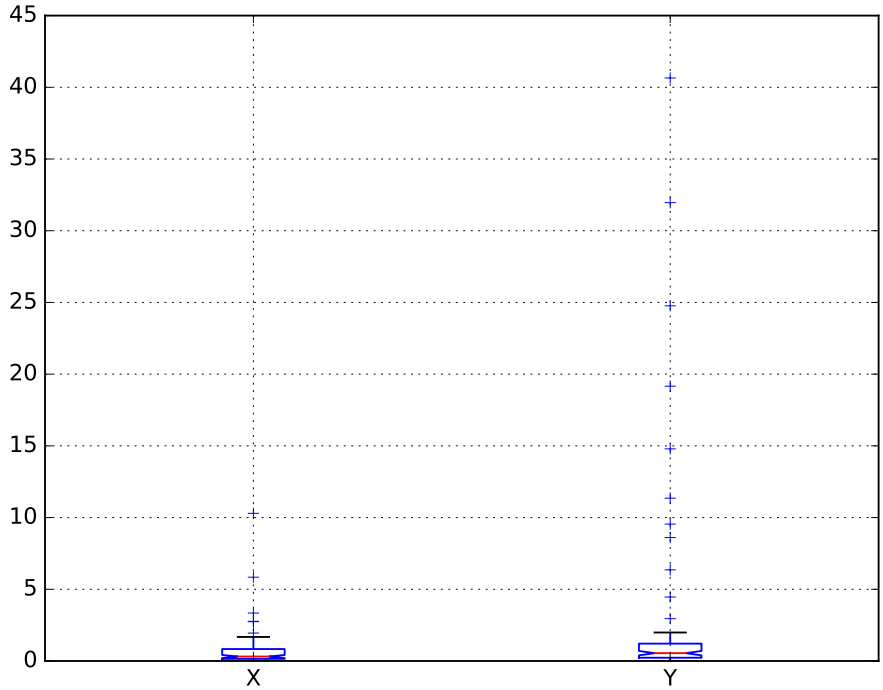
Here $p = max(k, j)$ where j represents the degree of interpolation for system 1 and k for system 2. $\bar{\bar{I}}$ is the identity matrix, while $\bar{\bar{\Omega}}$ is the coefficient matrix. The values of elements of $\bar{\bar{\Omega}}$ depend on the coupling method used, constraints of subsystems, and the coefficients of state variables. There are two components which depend on the system itself, that is "constraints" and "coefficients of state variables", while one component depending on the coupling method is *how inputs are connected to the outputs*. For example, in case of two subsystems s_1 and s_2, for the Jacobi method the inputs at time t_n are defined by the outputs of other subsystem at t_n. On the other hand, in case of Gauss-Seidel method, the inputs of s_1 at time t_n are the outputs of s_2 at t_n, but inputs of s_2 at t_n are outputs of s_1 at t_{n+1}. Conclusively, a coupling method is stable for a system if all the eigenvalues λ of $\bar{\bar{\Omega}}$ are in the unit circle i.e. $|\lambda(\bar{\bar{\Omega}})|_{max} < 1$.

It can be seen in figure 4.20a that the results generated from Gauss-Seidel method have a little difference form the reference solution given in figure 4.19. The median of the error for variable x lies around 0.015, which is close to 1% of the maximum value 1.1. For variable y the median lies around 0.02, which is also close to 1% of the maximum value 1.4. Looking at figure 4.20b, in fact there is no median of the error distribution. Most of the values are lying outside the quartiles. The reason is evident from figure 4.22, where the error is increasing with the passage of time. An abrupt peak in the solution of variable y can be seen near simulation time 50, in figure 4.22a. Similar smaller peaks can be seen in for variable x. In figure 4.22b it can be seen that the prominent egg like shape of the oscillator has vanished and the error is increasing at each cycle. In figure 4.21a, on the contrary, curves are very close to the ones shown in figure 4.19a. The prominent egg like shape in figure 4.21b is also present in the same way it was in figure 4.19b.

As the communication step size is fixed in the simulation of both Gauss-Seidel and Jacobi method, so there is no variability to demonstrate. Although, by comparing figure 4.19a and figure 4.21a less number of time steps in Gauss-Seidel method are evident. The comparison is not justified at he performance level due to the reasons discussed in chapter 6, but still it can be seen that OpenModelica uses a very small time step for solving the system. A smaller integration step size used by OpenModelica also indicates the stiffness of the system. The macro step size in Gauss-Seidel method was 0.5 which is clearly much larger than the integration step size of OpenModelica. Keeping in mind the stiffness of the system the results produced by the Gauss-Seidel method can be considered very good.

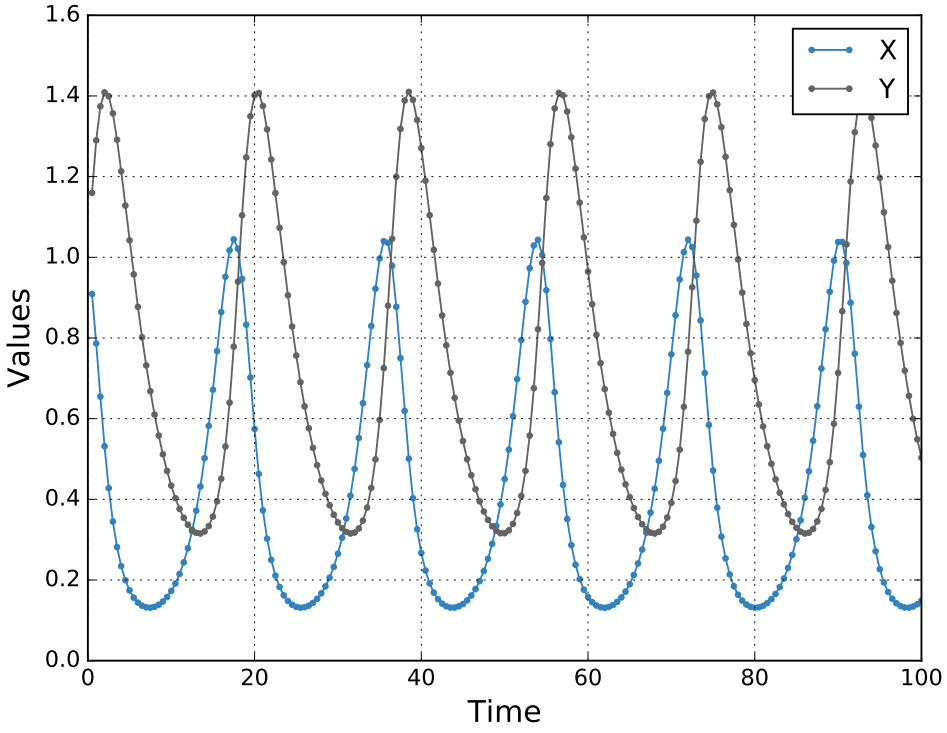


(a) Difference of results between Gauss-Seidel method and OpenModelica.

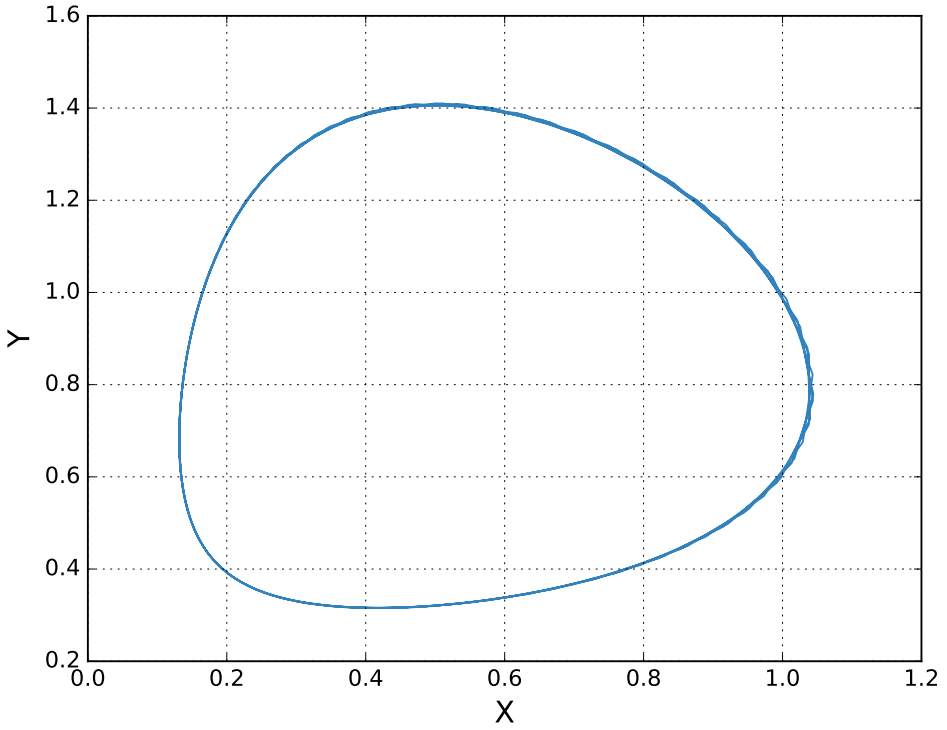


(b) Difference of results between Jacobi method and OpenModelica. Combining it with figure 4.22, clearly shows large amount of error in the values for variable *y*.

Figure 4.20: Cumulative differences of results between OpenModelica and other explicit methods, for "Lotka Volterra" system. The box plot shows how the values produced at each time step using explicit methods differ from the values of OpenModelica

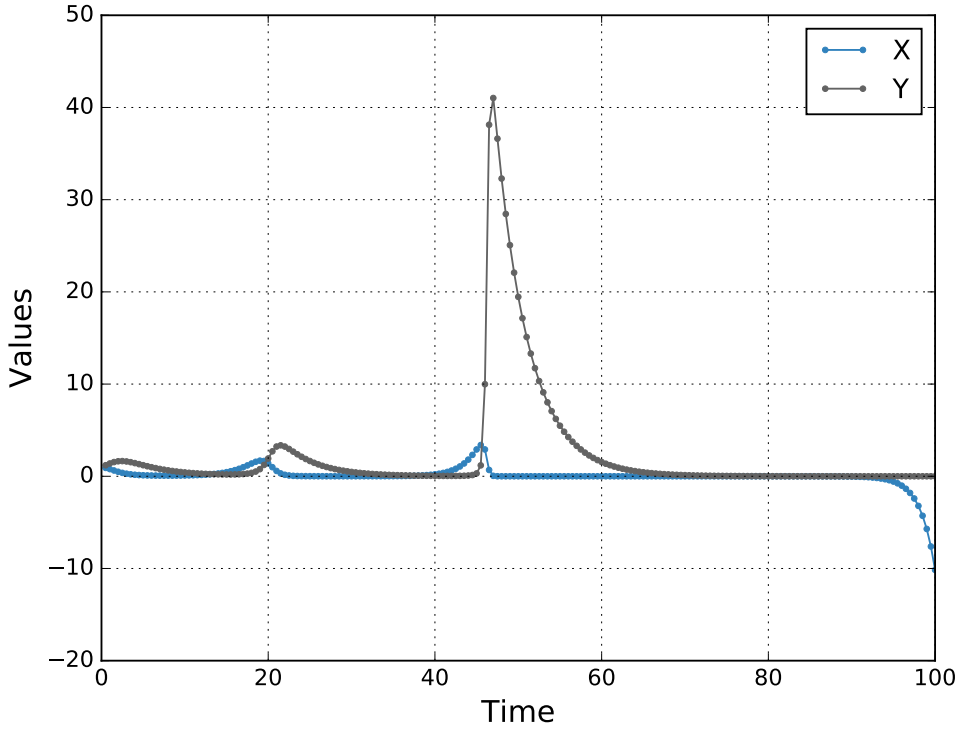


(a) Variables x and y plotted separately against simulation time.

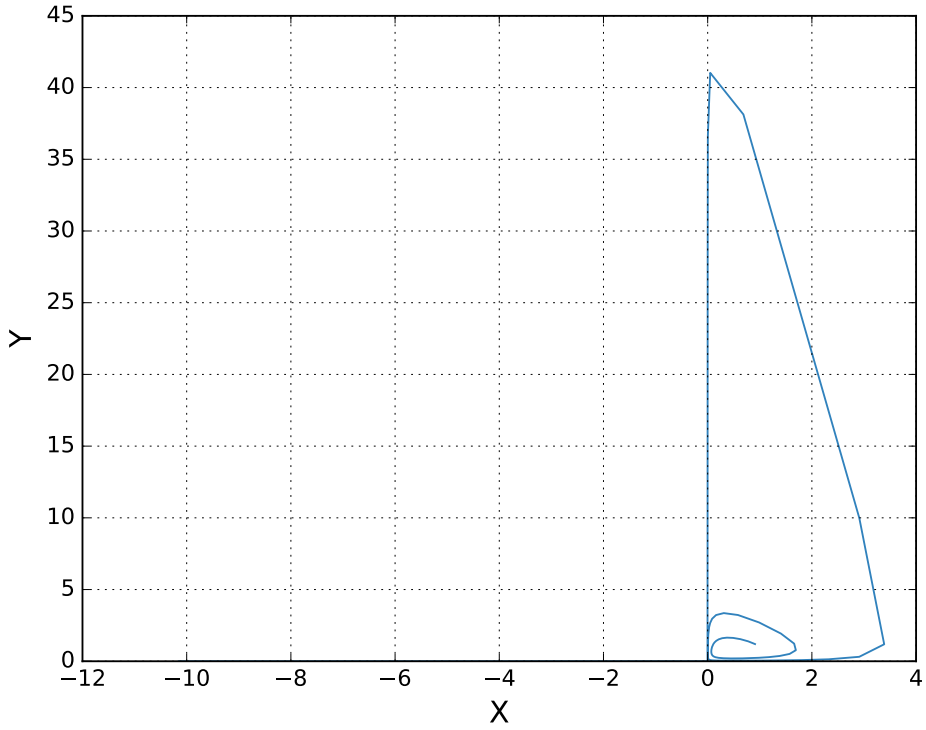


(b) Variables y plotted against variable x.

Figure 4.21: "Lotka-Volterra " system simulated using explicit Gauss-Seidel method. The "step size" is 0.5 here.



(a) Variables x and y plotted separately against simulation time.



(b) Variables y plotted against variable x.

Figure 4.22: "Lotka-Volterra" system simulated using explicit Jacobi method. The "step size" is 0.5 here.

5 Master-Slave Algorithms Using HLA and FMI

In the previous chapter, only explicit methods of solver coupling were discussed. In the present chapter, other forms of solver coupling techniques (see section 3.3.4) are discussed. Just like the previous chapter, the first few lines of each section introduce the mathematical concept behind the technique, then a distributed algorithm is designed. The mathematical scheme of implicit solver coupling discussed in section 5.2 has been proposed earlier, but the distributed algorithm based on that is developed by the author. The semi-implicit coupling technique discussed in section 5.3 has also been proposed earlier, but in a different way. Based on the previous concept the author further refines the concept specifically for FMI 1.0. Then a distributed algorithm is developed based on the refined technique. Strong coupling discussed in section 5.4 has been discussed in literature before. There can be various different choices of tools to be used with strong coupling, but the distributed algorithm developed here has not been presented before. Hybrid simulation algorithm discussed in section 5.5 is based on the ideas discussed in section 5.2. The algorithm developed in section 5.2 only works for coupling of continuous subsystems. Hybrid simulation algorithm further enhances the idea of section 5.2 and allows it to work for hybrid simulations, where coupling simulation components can have discrete valued state variables and outputs. The hybrid simulation algorithm is based on the idea of implicit coupling, but it has never been used for hybrid simulations before. The idea is first presented by the author. The last section (section 5.6) of the chapter discusses how different algorithms presented in the thesis are made part of a framework called SAHISim framework. The distributed algorithms presented in the thesis are not presented before in the literature. If there are any results that the author himself has published, then their references are provided at appropriate places.

5.1 Introduction

Previously discussed algorithms have the ability to work independently without the need of any regulator. The benefit of those algorithms is their ability to work independently. In this way single point failure can be avoided. Nevertheless, in tightly coupled simulations single point failure may not make much difference in many cases. As in order to proceed effectively it becomes essential for one federate to get an updated information from other federates. So if it is possible to get greater flexibility by using master-slave algorithms then it is completely logical and beneficial.

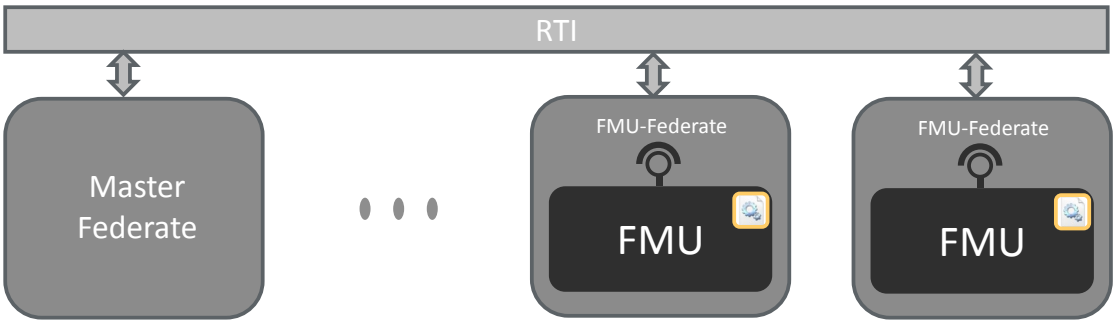
The master in these algorithms works like an orchestrator. Figure 5.1 shows how master and the RTI both are connected to all slaves. The RTI works as a medium for data and time synchronization, but master is the real orchestrator. Master drives all the slaves, guides them through different states and commands them to reach a common goal. Slaves on the other hand, are the work horses. Each one of them contains an FMU

inside, so they are the FMU-Federates in this topology. Master, on the contrary, does not contain any FMU, it is an executional form of different algorithms. Each slave fulfills the commands sent from the master, some of these commands require to take action on the FMU, like setting or getting state variables and input variables, and setting or getting time of the FMU. The internal integration of each FMU also takes place at the slave level. At "communication points" they share the data according to different schemes, orchestrated by the algorithm executing at master level.

Using master-slave topology, following algorithms are discussed in later sections

- **Fully Implicit Waveform Relaxation Algorithm**
- **Semi-implicit Algorithm**
- **Strong Coupling using the SUNDIAL, the HLA and the FMI**
- **Hybrid Simulation Algorithm**

Figure 5.1: Architecture of how FMUs, FMU-Federate, Master and the RTI fit in, in case of master-slave algorithms.

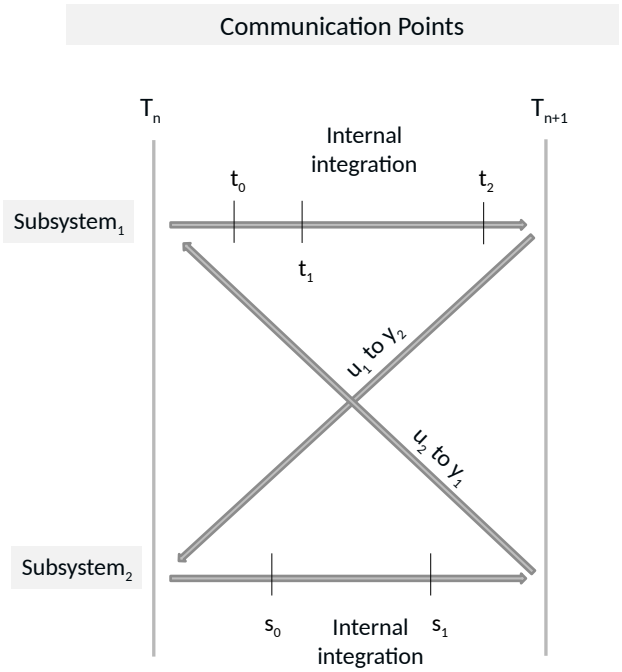


5.2 Distributed Algorithm Based on Fully Implicit Solver Coupling

The implicit solver coupling technique based on waveform relaxation was first presented in [LRSV82]. The idea is to start with a guess about the system states, which can be considered as a "predictor" step, and then rectify the guess in following fixed point iterations, which can be considered as the "corrector" steps. Figure 5.2 shows how the repetitive evaluations at a certain communication point (macro step) should work. It is worth noting that in the figure 5.2, T_n and T_{n+1} represent the communication points, while t_0, t_1, t_2 and s_0, s_1 represent the internal integration time steps. The length or number of internal integration steps is completely independent of each other and the length of communication step size $(T_{n+1} - T_n)$.

With respect to the algorithms discussed earlier, among other differences the Waveform Relaxation (WR) algorithm also differs in the strategy of data exchange. In comparison to Gauss-Seidel algorithm (algorithm 1) strategy of data exchange in WR algorithm is very robust. It does not imply any restrictions on the type of dependency relationship among FMU-Federates. Although, it is a fact that Gauss-Seidel method of data exchange can also be used in iterative methods like waveform relaxation algorithm [SBE+14], [LRSV82]. Nevertheless, the most robust approach is discussed here.

Figure 5.2: Waveform scheme for data exchange.



5.2.1 Description of the Algorithm

Looking deeply into algorithm 2 reveals the mechanisms of data exchange, time synchronization and convergence. Like the previous algorithm, here also there are three types of processors p_{rti}, which corresponds to the RTI, p_{master} corresponding to the master, and p_{slave} corresponding to the slave FMU-Federates.

It is beneficial to have a look at the code for p_{rti}. The p_{rti} process has to cater three types of messages. First type of message is the Time Advance Request Available (TARA) message, the other message is also similar which is a Time Advance Request (TAR) message. To see the difference among TARA and TAR services please have a look at the section 3.2.2.1. Both of these services cause the p_{rti} to deliver the updates to the invoking process and then granting it the time advance.

Second type of messages are related to the update of shared attributes. For these messages the RTI has a standard procedure which is followed in all algorithms. For readability, two variables are used to distinguish two different types of variables, $inputs^{j}_{slave}$ represent the input variables of j^{th} slave FMU-Federate, while $states^{j}_{slave}$ represent the state variables of j^{th} slave FMU-Federate. Similarly, p^{j}_{slave} is used to represent the j^{th} slave FMU-Federate.

Third type of messages sent from the p_{master} are destined to all "slave" processes. Although, the HLA RTI itself cannot distinguish between the master or the slave, but here it is assumed that the pseudo representative RTI process p_{rti} has this ability. The assumption does not make any change in the implementation, but only makes things more understandable and easy to read. While using the real RTI the exact same functionality can be easily achieved using, publisher subscriber model, different classes of FOM, and "interaction" classes for commands.

The "master" algorithm has two main parts, one is the "Main" procedure which embodies the main strategy for the orchestrator. Second, the message handling portion, which handles two types of messages. One

is the ⟨Update⟩ message, and the second is ⟨Time Grant⟩. The ⟨Update⟩ message is received in result of the updated shared state variables. Upon receiving ⟨Update⟩ message the value of the update is extracted from the message and then added to a list *updates*. The list *updates* contains the updates received from all FMU-Federates ("slaves"). For simplicity, it is assumed that each shared state variable is represented as a separate attribute in the Federation Object Model (FOM). When the attribute is updated, the master receives the ⟨Update⟩ message, along with the updated value of the shared state variable. The updated **value** is reflected into the *updates* list against its **name**. It should be kept in mind that whenever any federate sends any "time advance" request to the RTI, it first receives all the updates queued for that federate, then it receives the ⟨Time Grant⟩. Upon receiving ⟨Time Grant⟩ message, first the number of received updates in *updates* are counted and if they are equal to the number of "expected" updates then the "exceution state" (stored in $ExecState$ variable) is change to $UpdatesArrived$. Which tells the "Main" routine that all the updates from slaves have been received. If all the updates are not received then the $ExecState$ is set to $WaitForUpdates$, which means that the "Main" routine will wait for all the updates to arrive.

The most important aspect for the understanding of the following algorithms is to acknowledge that the state variables, outputs and inputs are "seemingly" represented using vectors, but individual values are "always" accessed using their **names**, instead of the indices of the vectors. The reason is that the HLA based implementation is only possible in this way. The HLA FOM has classes and their attributes. The FOM structure for the algorithms mentioned here can be designed in many different ways, the simplest one to understand is that each output or state variable is bound to a single attribute and the **name** of that attribute is bound to the **name** of the output or state variable of the FMU. The mapping of the FOM attribute and the respective FMU variable name can be saved in a map. Because the transformation from one to other is straightforward, in presented algorithms only one **name** is used to access any variable, instead of two. In reality, the vectors here are more like a "map", containing both the values and their names. So each element of the map is a pair, having a **name** and a **value**. If the name of any pair has to be accessed then it can be accessed using $name(x)$, where x is an element of map. Similarly, the value can be accessed using $value(x)$. In cases like evaluating a norm, only the values are of concern. A conversion from a map to vector of values is quite easy. If there is a map æ and it is required to extract all its values into a vector v, then all the values of æ can be copied into v using following

$$v \leftarrow [value(x) \mid x \in æ] \tag{5.1}$$

Similarly, if all the names have to be extracted in a set s then

$$s \leftarrow \{name(x) \mid x \in æ\} \tag{5.2}$$

The *norm*() function used in the following algorithms relies on the equation 5.1 to get the vector of values from a map and then get their difference.

The "Main" routine has one main loop starting from line 8 and ending at line 32. The loop ends when the simulation time exceeds the provided $simlationEndTime$. The loop starting at line 12 performs the convergence check. The line 12 has the norm function to check the difference between the state variables of current and previous fixed point iteration. If the difference is less than a certain tolerance value TOL then it is assumed that the fixed point iteration has converged. If the number of fixed point iterations are too much, even greater than a predefined value MAX, then it is supposed that convergence cannot be achieved in that specific case. There could be different reasons when the convergence is not achievable. One reason could be that the communication step size is too big, or the system is not convergent, or if the system is

Algorithm 2 Fully Implicit Waveform Relaxation Algorithm.

1: Code for p_{master}:
2: **Procedure** Main()
3: $time \leftarrow 0$
4: send ⟨Send Initial States, $time$⟩
5: GetUpdates()
6: $pStates \leftarrow pStates \cup updates$
7: $currStates \leftarrow pStates[-1]$
8: **while** $time \leq simulationEndTime$ **do**
9: $time \leftarrow time + step$
10: $iterationStates \leftarrow \{(name(x), \infty) \mid x \in updates\}$
11: $iterationCount \leftarrow 0$
12: **while** $norm(iterationStates - currStates) > TOL$ **do**
13: $iterationCount \leftarrow iterationCount + 1$
14: **if** $iterationCount > MAX$ **then**
15: throw exception "No convergence" and **terminate**
16: **end if**
17: send ⟨Advance Time, $time$⟩ to all "slave" processes.
18: send ⟨Send States, $time$⟩ to all "slave" processes.
19: GetUpdates()
20: $iterationStates \leftarrow currStates$
21: $currStates \leftarrow updates$
22: for each p^j_{slave} :
23: send ⟨Update States, $\{x \mid x \in pStates[-1] \wedge name(x) \in states^j_{slave}\}, time$⟩ to p^j_{slave}
24: for each p^j_{slave}:
25: send ⟨Update Inputs, $\{x \mid x \in currStates \wedge name(x) \in inputs^j_{slave}\}, time$⟩ to p^j_{slave}
26: **end while**
27: send ⟨Send States, $time$⟩ to all "slave" processes.
28: GetUpdates()
29: $pStates \leftarrow pStates \cup updates$
30: send ⟨End Iteration,$time$⟩ to all "slave" processes
31: send ⟨TAR,$time$⟩ to p_{rti}
32: **end while**
33: send ⟨End Simulation⟩ to all "slave" processes
34: **terminate**
35: **end**
36: **Procedure** GetUpdates()
37: $ExecState \leftarrow WaitForUpdates$
38: $updates \leftarrow \{(name(x), \infty) \mid x \in updates\}$
39: **while** $ExecState \neq UpdatesArrived$ **do**
40: send ⟨TARA,$time$⟩ to p_{rti}
41: **end while**
42: **end**

Algorithm 2 Fully Implicit Waveform Relaxation Algorithm (Continued).

43: upon receiving ⟨Time Grant, *time*⟩:
44: **if** $|updates| = NUMBER\_OF\_STATES$ **then**
45: $ExecState \leftarrow UpdatesArrived$
46: **end if**

47: upon receiving ⟨Update, *states*,*time*⟩:
48: $\forall x \in updates\, \forall y \in states\, ((name(x) = name(y)) \rightarrow (value(x) \leftarrow value(y))$.

49: Code for p_{slave}:
50: upon receiving any message:
51: store every message into *commandList*
52: upon receiving ⟨Time Grant, *time*⟩:
53: search for all "allowable" messages for the current state in *commandList*.
54: execute the action on the first message found.
55: send ⟨TARA,*time*⟩ to p_{rti} except in case of ⟨End Simulation⟩ and ⟨End Iteration⟩

56: action for ⟨Send Initial States, *time*⟩:
57: Initialize the FMU, and send the initial states to p_{master} in from of ⟨Update, *states*,*time*⟩
58: action for ⟨Advance Time, *time*⟩:
59: Integrate the FMU to *time*
60: action for ⟨Send States, *time*⟩:
61: send the state variables in form of ⟨Update,*states* ,*time*⟩ to p_{master}
62: action for ⟨Update States,*state*, *time*⟩:
63: set the time of FMU to *time*
64: set the respective state variable of FMU to *state*
65: action for ⟨Update Inputs,*input*, *time*⟩:
66: set the respective input variable of FMU to *input*
67: action for ⟨End Iteration, *time*⟩ :
68: send ⟨TAR, *time*⟩ to p_{rti}
69: action for ⟨End Simulation⟩:
70: **terminate**

71: Code for p_{rti}:
72: upon receiving ⟨TARA, *time*⟩ from p_i:
73: send all messages queued for p_i to p_i with time stamp $\leq$ *time*
74: upon confirmation of sending all updates send ⟨Time Grant⟩ to p_i
75: upon receiving ⟨TAR, *time*⟩ from p_i:
76: send all messages queued for p_i to p_i with time stamp $\leq$ *time*
77: upon confirmation of sending all updates send ⟨Time Grant⟩ to p_i
78: upon receiving ⟨Update,*states*, *time*⟩ from p_i:
79: enqueue *states* with time stamp *time*, for subscribing process p_{master}
80: upon receiving any other command ζ from p_{master}:
81: enqueue the command ζ for all subscribing "slave" processes.

convergent, the partition of variables is not proper to make it convergent [LRSV82]. In case there is no convergence achieved, a modeler should first try to minimize the macro time step. If that does not work then the modeler should have a look at the system and see whether the system is contractive in nature or not. To make the sure that system is contractive, there are few guidelines enumerated in [LRSV82].

It is important to understand the lines for updating $inputs$ and $states$. The command to update states is at line 23 and the command to update inputs it is at line 25. There is a set of sets $inputs$ which contains the names of all the inputs of all slave subsystems. Similarly, the names of state variables of all the slaves are stored in the set of sets $states$. The expression $inputs^{j}_{slave}$ gives the set of names of inputs of j^{th} slave. Similarly the expression $states^{j}_{slave}$ gives the name of states of j^{th} slave. At line 25 the statement says that a parameter to ⟨Update Inputs⟩ message is the set of inputs which are specific to j^{th} slave. Similar is the case of state variables. Again, all these notation are used for describing the algorithm completely, otherwise the actual implementation using the RTI and its Federation Object Model is a little different. There we need not make any separate sets of $inputs$ and $states$, instead separate classes and their unique attribute identifiers produce exactly the same result.

Another thing should also be kept in mind that here it is assumed that all the outputs are connected directly to their respective state variables. If that is not the case then a modification in the algorithm is required, which will be discussed later in the section 5.5. Because there is no conversion from state variables to the outputs, so outputs are not even mentioned in the algorithm, only state variables are considered.

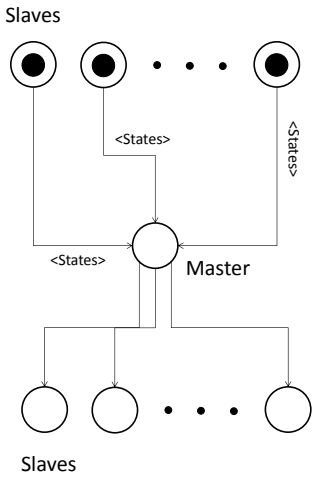
The procedure "GetUpdates()" (defined at line 36) does an important work. It is used when the master waits for the updates to arrive from all the FMU-Federates. It changes the execution state $ExecState$ of the master from $UpdatesArrived$ to $WaitForUpdates$, and waits until it turns back to $UpdatesArrived$. The change can only occur during the processing of ⟨Time Grant⟩ when all the updates are received. The condition is checked using the expression "$|updates| = NUMER\_OF\_STATES$" at line 44. The line counts the number of states whose values are not ∞, if their count is equal to the shared state variables $NUMBER\_OF\_STATES$, then this means that the values of shared state variables have been received. In that case the execution state is changed to $UpdatesArrived$.

Using line 17 and line 18 the master asks the slaves to move forward in time, this refers to the "internal integration" in figure 5.2. Using line 23 master moves the FMU-Federates back one step, and updates the state variables. Using line 25 it connects the outputs to inputs. Here for simplicity it is assumed as if p_{master} sends an update to all "slave" processes one by one. It is not the case in reality, using FOM of HLA "Input" and "State" classes can be defined. When an attribute of any class is updated, all subscribing federates automatically receive an update.

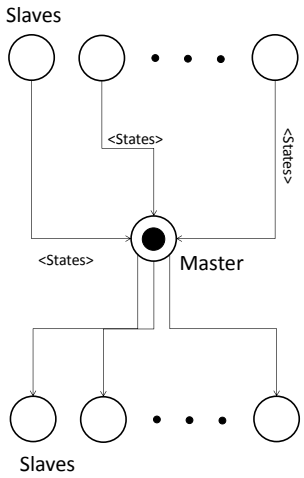
The variable $pStates$ is a list, $pStates[-1]$ gives the mapped vector (discussed earlier) of state variables stored at the last index of $pStates$. At line 29 new elements to the list are added, which are the state variables after the convergence. Using line 30 master tells all the FMU-Federates to close the internal integration of FMUs because the fixed point iteration has converged, upon receiving this command "slaves" send the ⟨TAR⟩ request to RTI. Using line 31 master also sends the ⟨TAR⟩ request to RTI, which ends one episode of execution. After sending TAR request with a parameter *time*, a federate cannot generate an update on that *time*, it can only generate updates after that *time*.

5.2.2 Slave For Implicit Waveform Relaxation Algorithm

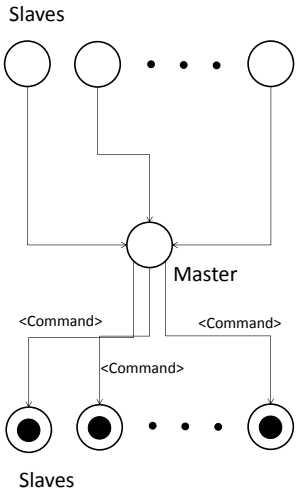
Although the real strategy of integration lies in the master algorithm, but obviously master algorithm can only work with the participation of the slave algorithm. The slave algorithm is quite straightforward.



(a) Master waiting for all salves to send the states.



(b) Master receiving states from all slaves one by one, sooner or later.



(c) Master sending other commands to slaves to continue simulation.

Figure 5.3: Petri net diagram for master-slave synchronization.

It goes into an infinite loop and waits for commands to be sent from the master. The slave works like a state machine, it changes its state based on the messages received from the master. It is important, therefore, to understand the commands (in form of messages) sent to, and processed by the slave process. In the messages when parameter *time* is not discussed explicitly then it means that it is only used for time stamping. Again it is reiterated that few lines of code appear to send messages directly to another process. In reality no process can communicate to another process directly, all communication has to happen through the RTI. Following are the commands sent from the master to a slave via the RTI.

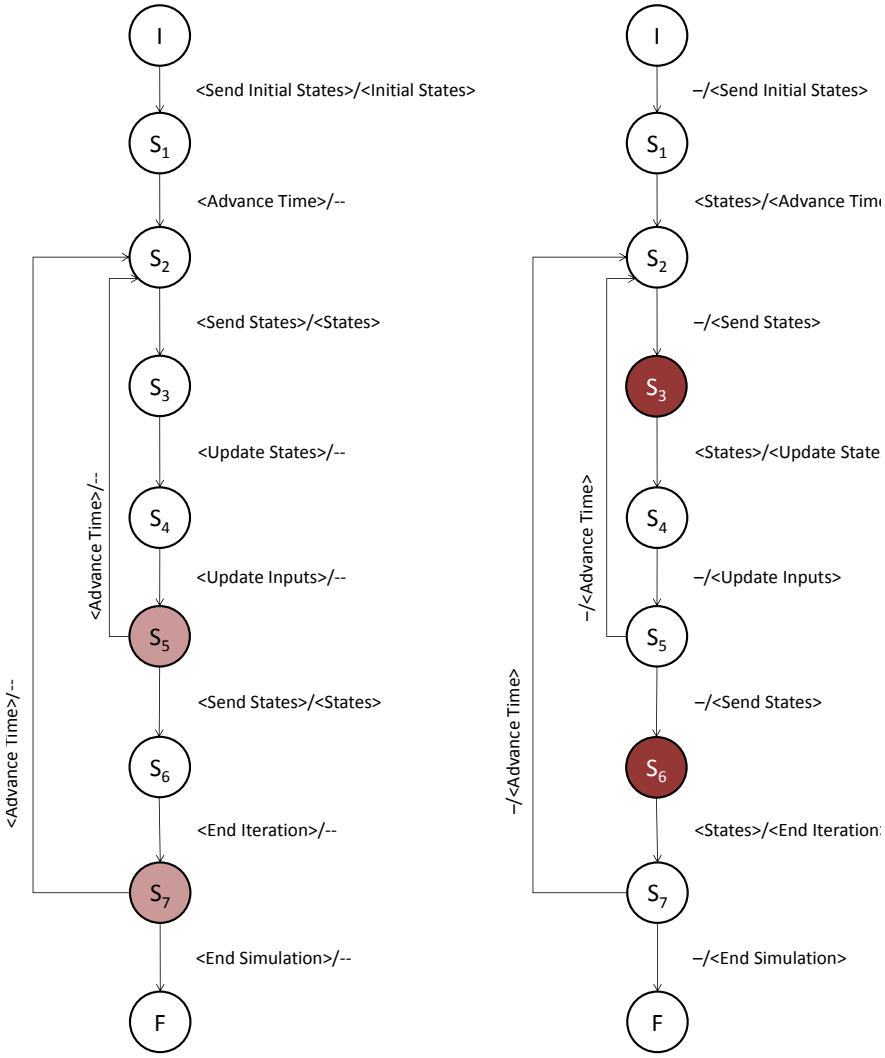
- ⟨**Send Initial States**⟩: The command is sent to "slave" processes to ask them to send the initial values of the state variables. The slave in response sends the values of state variables using the ⟨Update⟩ message.
- ⟨**Send States**⟩: The message is used to ask slaves to send their newly calculated values of state variables in the form of ⟨Update⟩ message.

- ⟨**Update Inputs**⟩: The message is sent to ask a slave to update its input variables to the values sent with the message.
- ⟨**Update States**⟩: The message is sent to ask a slave to update its state variables to the values sent with the message.
- ⟨**Advance Time**⟩: On receiving ⟨Advance Time, *time*⟩, FMU-Federate integrates the FMU to *time*, which is provided as parameter of the message.
- ⟨**End Iteration**⟩: The message is sent to inform the FMU-Federates that the fixed point iteration has converged, so they can close the internal integration procedure up to *time*, which is a parameter of the message.
- ⟨**End Simulation**⟩: The message asks an FMU-Federate to abandon all processing and go to **terminate** state.

The "Main" procedure in algorithm 2 when converted to a state machine takes the form of figure 5.4b. For more clarity input and output are written with every state transition like a Mealy machine. It is clear from the figure 5.4b that the master only receives inputs on states S_1, S_3 and S_6. Exactly opposite is the case of slave, which sends outputs only at states I, S_2 and S_5. We can say that for a slave except states I, S_2 and S_5, all other states are quiescent [AW04]. Looking at lines following the line 18 in algorithm 2, it reveals that the master keeps waiting for all the updates to come from all slaves, until then it does not send any more command to any slave. This causes the states S_1, S_3 and S_6 in master to act like "synchronization points". In terms of Petri net diagram the concept of synchronization point is shown in figure 5.3.

The discussion of "synchronization point" is very important because it ensures the correct execution of the algorithm. Suppose there was no synchronization point, in such a case there was no way to prove that the slaves are obeying the commands in the right order. As mentioned earlier in section 4.3.1 that the presented distributed model ensures "TSO" delivery, but it does not ensure that the messages will be received at the slave in the order they were sent. Messages with the same time stamp may change the order at reception. In such a situation if a slave, for example, applies the updated inputs before updating the state variables then the calculation will go completely wrong. To avoid such a situation it is essential to have synchronization points.

It should be clear in the mind of the reader that these "synchronization points" have nothing to do with the HLA "synchronization points". They are a completely different concept, and the use of HLA synchronization points in current scenario would be extravagant. The purpose of HLA synchronization points is to give control of regulation to a federate. A federate announces a synchronization point and waits until all others do not respond to it. In turn all other federates also have to wait for the signal form the announcing federate to begin their execution again. The procedure devised here is much cheaper than the HLA synchronization points, because it does not disturb the normal execution of the RTI. Using HLA synchronization points forces the RTI to push the complete federation into a special mode, until the federates are not synchronized. After the synchronization is achieved, the same state has to be restored by the RTI. The computational cost of performing the push and restore procedure is clearly higher than the presented technique.



(a) State machine for **slave** of waveform relaxation algorithm. Highlighted states are branching states.

(b) State machine for **master** of waveform relaxation algorithm. Highlighted states are synchronization points.

Figure 5.4: Master and slave state machines. State S_6 and the relative command is introduced just to guarantee correct synchronization, although without it every thing else could work properly.

5.2.3 Proof of the Correct Synchronized Execution

In order to prove that a slave always follows the correct execution path induced by the master, we denote the master state graph as $\mathcal{M}$ and slave state graph as $\mathcal{S}$ then

$$\mathcal{M} \preceq \mathcal{S} \tag{5.3}$$

Equation 5.3 means that $\mathcal{S}$ simulates $\mathcal{M}$, or in other words $\mathcal{M}$ and $\mathcal{S}$ have simulation preorder relationship [VG01]. Each move generated by the master $\mathcal{M}$ can be simulated by slave $\mathcal{S}$. From this fact it can be concluded that a slave always follows the master correctly until and unless the order of commands sent from the master to a slave, is changed during network communication.

Change in order only affects the execution when the slave is in a branching state. Branching states are highlighted as red in figure 5.4a. At a branching state (or branching point), there can be cases when a slave can go into a direction not intended by the master. Before proving that by using synchronization points such a situation can be avoided, few structures must be defined

- Σ is the set of all commands $\{\mu_1, \mu_2, \mu_3, \ldots, \mu_n\}$.
- Γ is the list where a slave stores all the commands sent to it. A slave removes a command from Γ when it executes it. At any time it may contain a limited number of commands induced by the master algorithm. So $\Gamma \subset \Sigma$ at any time of execution, with $|\Gamma \cap \Sigma| \geqslant 0$
- Λ is the set of all states $\{s_1, s_2, s_3, \ldots, s_n\}$ in the slave state graph.
- Λ^m is the set of all states $\{s_1^m, s_2^m, s_3^m, \ldots, s_n^m\}$ in the master state graph.
- $\Lambda_f \subset \Lambda$, contains all branching states in Λ.
- $\Lambda_f^m \subset \Lambda^m$, contains all branching states in Λ^m.
- Π is the set containing elements $\{\Pi_1, \Pi_2, \Pi_3, \ldots, \Pi_n\}$. An element Π_i is the set of all commands allowed at state s_i. So each $\Pi_i \subset \Sigma$, with $|\Pi_i \cap \Sigma| \geqslant 1$

The slave algorithm works in a way that it keeps accepting the commands and saves them in the list Γ. At each state s_i it follows the first command it finds in $\Gamma \cap \Pi_i$. In order to prove that the synchronization algorithm works perfectly, it is sufficient to prove that at any time during the execution of the algorithm, for the state s_i active at that time following is true

$$|\Gamma \cap \Pi_i| \leqslant 1 \tag{5.4}$$

In order to prove above statement, it should be noted that there are certain commands which work like synchronization commands for salves and the master. The state where the slave ends up in result of executing any of these commands is called as a "synchronization point" or synchronization state. The master also ends up at the similar state, in its own state graph, with a difference that it first gets into the state and then issues the command. At these points master waits for a response from all the slaves and it does not issue any more command until it has received a response from all of them. It is easy to observe that the condition given in equation 5.4 can only be violated at a branching state where $|\Pi_i| > 1$.

Lemma 1. *Condition given in equation 5.4 remains valid, if starting from any state in Λ_f^m and Λ_f respectively, master and slave have to go through a synchronization point in their state graphs, in order to reach any state— same or different—in Λ_f^m and Λ_f again.*

Proof. Suppose that the master has passed through a branching state s_f^m. The corresponding state of s_f^m in slave is s_f. By this, the master sends a command in Π_f to slave. As synchronization point s_s^m must follow it by definition, so master must wait for a response from all the salves at s_s^m. The corresponding state to s_s^m in slave is s_s. At s_s^m the master cannot send anymore commands until it receives all the responses. On the side of a slave, the list Γ now may or may not contain a command present in Π_f. In order to send a response back to the master, the slave has to pass through s_f and reach s_s, because by equation 5.3 slave simulates the master. To do this it must consume the command sent from the master. So essentially when a slave reaches at s_s, it must have consumed the command in Π_f sent from the master. This means that when a slave sends a response back to master, the list Γ of that slave does not contain any command in Π_f. The property holds for all branching states s_f and their respective set of commands Π_f, which proves that lemma 1 is true. □

The proof means that the condition given in equation 5.4 is entailed provided

1. Slave simulates the master and the master sends commands in correct order.
2. Conditions imposed by lemma 1 are valid in the state graphs of master and slave.

The above discussion proves that if the condition given in equation 5.4 remains valid at all times during the execution of the simulation, then there will be no problem of synchronization. The property is enforced by lemma 1. Looking at the slave state machine in figure 5.4a, it is clear that there are no two branching states reachable from each other without passing through a synchronization point. For example, if the state S_6 had been removed from figure 5.4a and figure 5.4b then a situation can arise that a slave could go from state S_5 to state S_2, when the intention had been to go through S_7 and then go to S_2. It is important to mention that the proof of synchronized execution is sufficient for any number of slaves, because for the sake of synchronization each slave is only dependent on the master. This is a benefit of master-slave topology.

5.2.4 Complexity of WR Algorithm

Time and space complexity analysis of distributed system is not straight forward [AW04]. In case of numerical algorithms it becomes even more complex. As the way numerical algorithms divide a problem into sub-problems is not straightforward. Each trajectory on time horizon is divided into steps, the points on the trajectory may not be equidistant, because the step size is normally not constant. Although, in aforementioned algorithms a constant step size is used, yet in later sections it will be discussed how it can be made variable. This makes time complexity analysis different from other distributed algorithms. Although a general model of complexity can be devised, but it does not tell anything how the problem "size" is related to the time of execution. Rather it can only tell something about how the time of execution is related to the "nature" of the problem. Message complexity is much more relevant in case of distributed numerical algorithms. As message communication takes large amount of execution time in distributed algorithms, so from message complexity, the time complexity can be derived in abstract terms.

Time complexity of distributed numerical algorithm is not comparable to a monolithic one, because nature of the problem is completely different. In a situation where a subsystem $\mathcal{S}_p$ of a system $\mathcal{S}$ is coupled with

another subsystem $\mathcal{S}_q$. If $\mathcal{S}_p$ is tried to solve with a monolithic solver, while receiving updates from $\mathcal{S}_q$, then the results will most probably be nothing less than absurd. A distributed solver on the other hand is robust. A subsystem $\mathcal{S}_p$ of a system $\mathcal{S}$, under some restrictions can be coupled with some other system $\tilde{\mathcal{S}}$. At current stage of research, a monolithic solver for a monolithic problem has much more choices and tools for optimization than any distributed solver. Hence, comparing a monolithic solver with a distributed one is not justifiable. Both address their own spectrum of problems. The topic is revisited later in section 6.1.

For message complexity, it should be realized that in master-slave configuration, a slave waits for the commands from the master indefinitely. Meaning, slave does not do anything unless the master asks it to. The commands from master have two types, one which do not require a response from slaves, second which do require a response. This is shown in figure 5.5, along with the topology how the master, slaves and the RTI are connected together. The same fact is clear from the figure 5.4, where master only seeks response from slaves when it asks them to send the state variables.

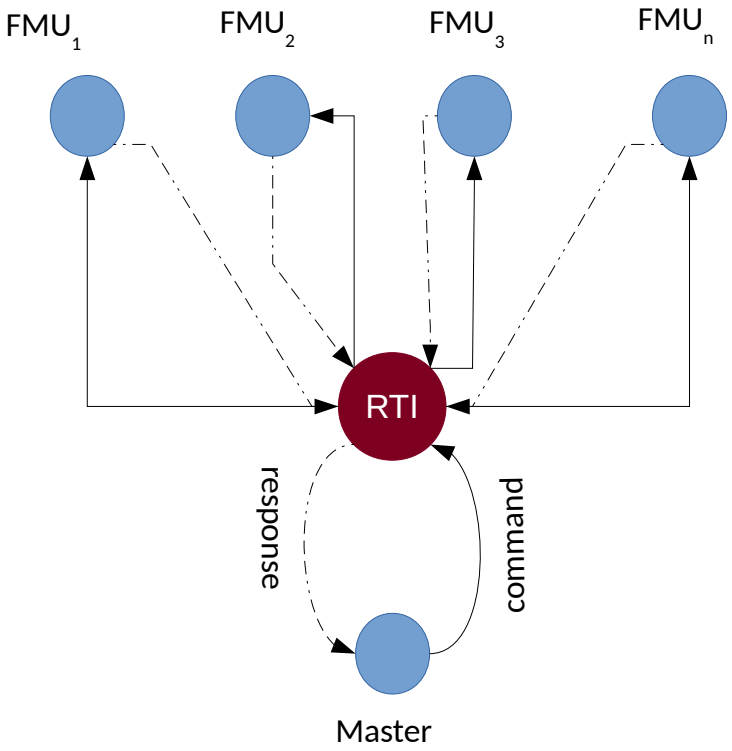


Figure 5.5: Figure shows how the master, slaves and the RTI are connected together. The dotted lines for responses show that for some commands master may not be expecting a response.

There are main three elements which are affecting the number of messages during algorithm execution. Number of subsystems or components n_c, the number of macro steps n_s, and the sum of number of fixed point iterations n_i per macro step ($sum_i = \sum_{i=0}^{n_s} n_i$). Looking at algorithm 2 the main simulation loop sends only three commands outside of the fixed point iteration. One of them (⟨Send States⟩) requires a response from all slaves. Inside the fixed point iteration it sends four commands, one of them requires a response. So the following expression can give the number of messages $\mathcal{M}_{wr}$ sent during the simulation .

$$\mathcal{M}_{wr}(n_s, sum_i, n_c) = (2n_s n_c + 2n_s n_c) + \sum_{i=0}^{n_s}(3n_i n_c + 2n_i n_c) = 4n_s n_c + \sum_{i=0}^{n_s} 5n_i n_c \tag{5.5}$$

Asymptotically speaking, if there is an upper bound I_{max} on number of fixed point iterations within any macro step, then equation 5.5 can be converted to following

$$\mathcal{M}_{wr}(n_s, I_{max}, n_c) = \mathcal{O}(n_s n_c + n_s I_{max} n_c) = \mathcal{O}(n_s I_{max} n_c) \tag{5.6}$$

Equation 5.6 can be further refined by observing that the number of components can be considered as initial parameter p_c. In practice their number may not achieve three digits. So it can be considered as a small constant multiple. Consequently, the message complexity can be reduced to

$$\mathcal{M}_{wr}(n_s, I_{max}, p_c) = \mathcal{O}(n_s I_{max} p_c) = \mathcal{O}(n_s) \tag{5.7}$$

From equation 5.7 it is clear that the most important factor which affects the performance of a distributed numerical algorithm is the number of macro steps, which is a direct consequence of macro step size. In most cases making the step size too big can cause the number of fixed point iterations to increase. So making step size just as big as possible may not be the best option. An intelligent step size control strategy has to be devised which increases the step size while keeping the fixed point iterations to minimum. The result is in perfect harmony with the result of performance evaluation of monolithic algorithms.

5.2.5 Case Study

The reasons why a system converges to fixed point and hence produces correct results while being solved with a Waveform Relaxation (WR) algorithm, can be seen in [LRSV82] and referenced literature. The proof is out of the scope of this work, but briefly it can be said that the system mentioned in equation 4.3 can a be transformed into a "canonical form" of WR algorithm [LRSV82], which is given as follows

$$\begin{aligned} \dot{x}^k &= f(x^k, x^{k-1}, \dot{x}^{k-1}, z^{k-1}, u) \\ z^k &= g(x^k, x^{k-1}, \dot{x}^{k-1}, z^{k-1}, u) \end{aligned} \tag{5.8}$$

Here f represents the differential part of the system wile g represent the algebraic constraints. It is proven in literature that the system is guaranteed to converge if (f, g) is Lipschitz continuous with respect to x and globally contractive with respect to $(\dot{x}, z)$.

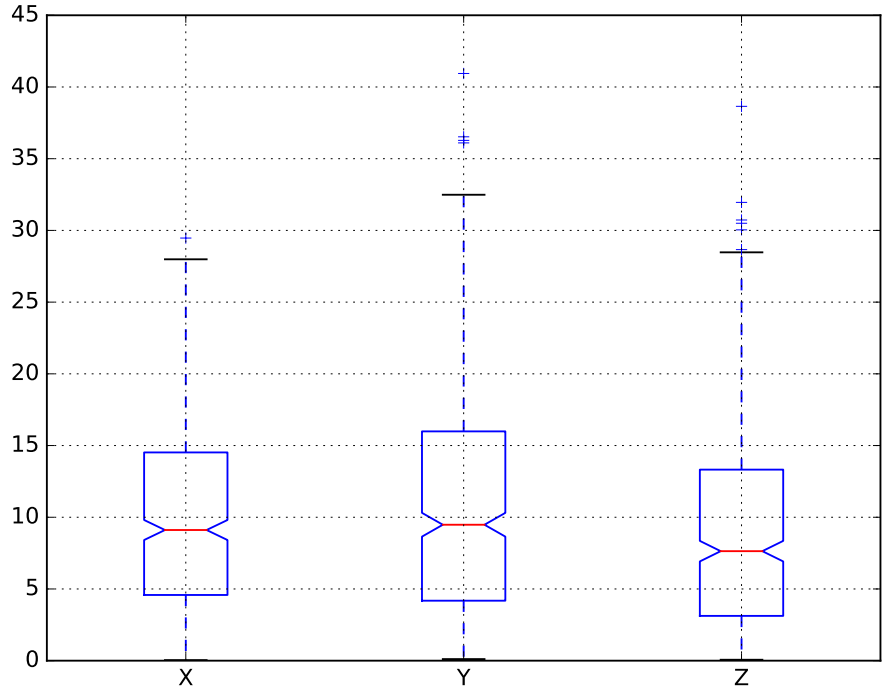


Figure 5.6: Difference of results between results of OpenModelica and WR algorithm for the problem of "Lorenz attractor".

A model is simulated using WR method, which is well renowned for its stiffness. It is also very popular for its chaotic behavior and strong coupling among state variables [GWR86]. It is called as "Lorenz attractor" and described as follows

$$\begin{aligned} \dot{x} &= \sigma(y - x) \\ \dot{y} &= x(\rho - z) - y \\ \dot{z} &= xy - \beta z \end{aligned} \tag{5.9}$$

The "Lorenz attractor" is a very good example of stability of WR method over others. Explicit Gauss-Seidel method cannot be used for the system due to circular dependency of variable y and z. Explicit Jacobi method does not give any results comparable to the standard solution even after reducing the simulation step size below 0.01. However WR method gives comparable results shown in figure 5.7. The step size had to be kept as small as 0.05 due to the stiff nature of the system. Later in section 5.5.2 a mechanism of steps size control is also discussed, but because it is not the main topic of the study so in depth discussion is avoided to prevent digression.

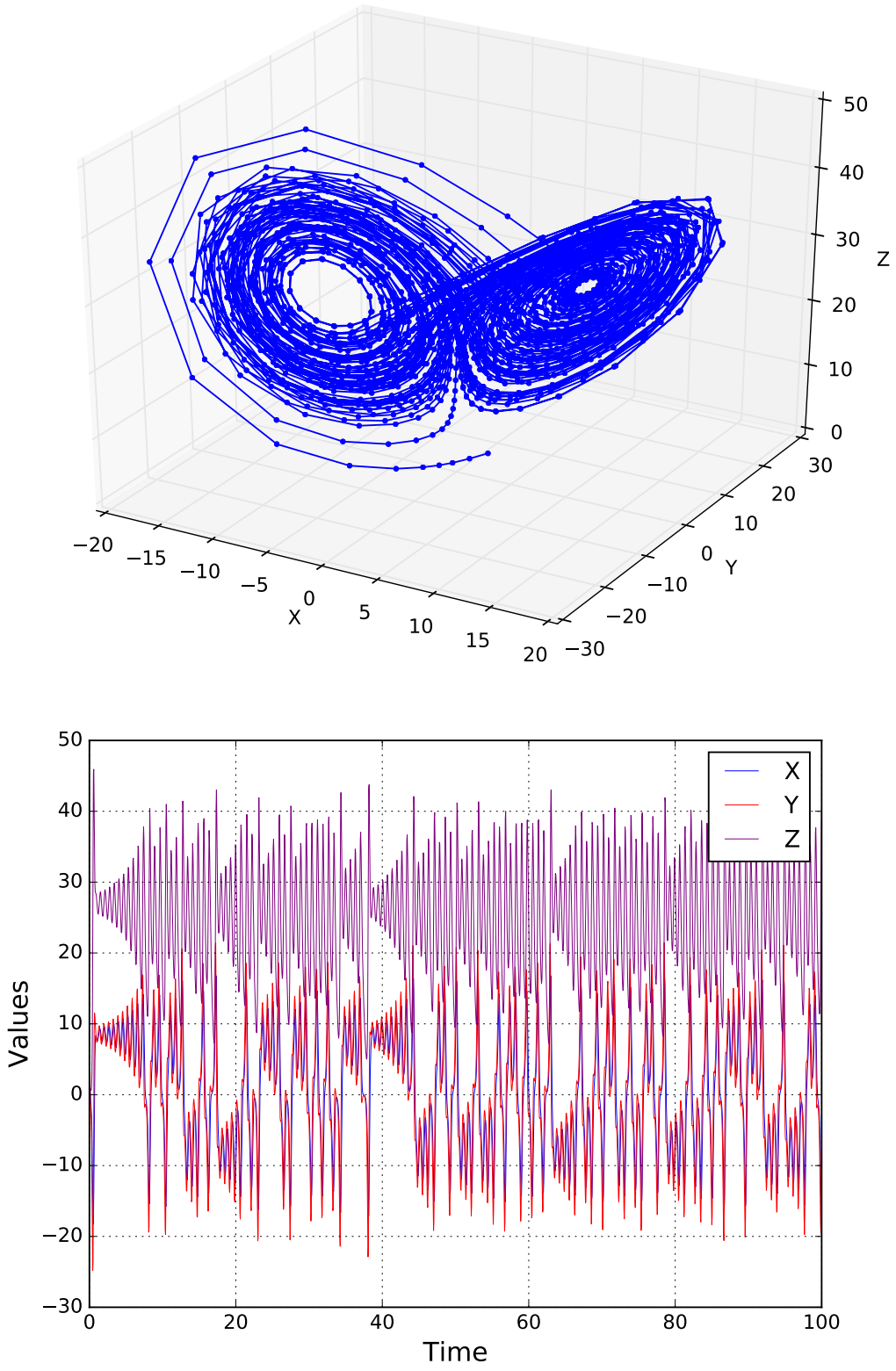


Figure 5.7: "Lorenz attractor" simulated using the waveform relaxation algorithm, the "step size" is 0.05 here.

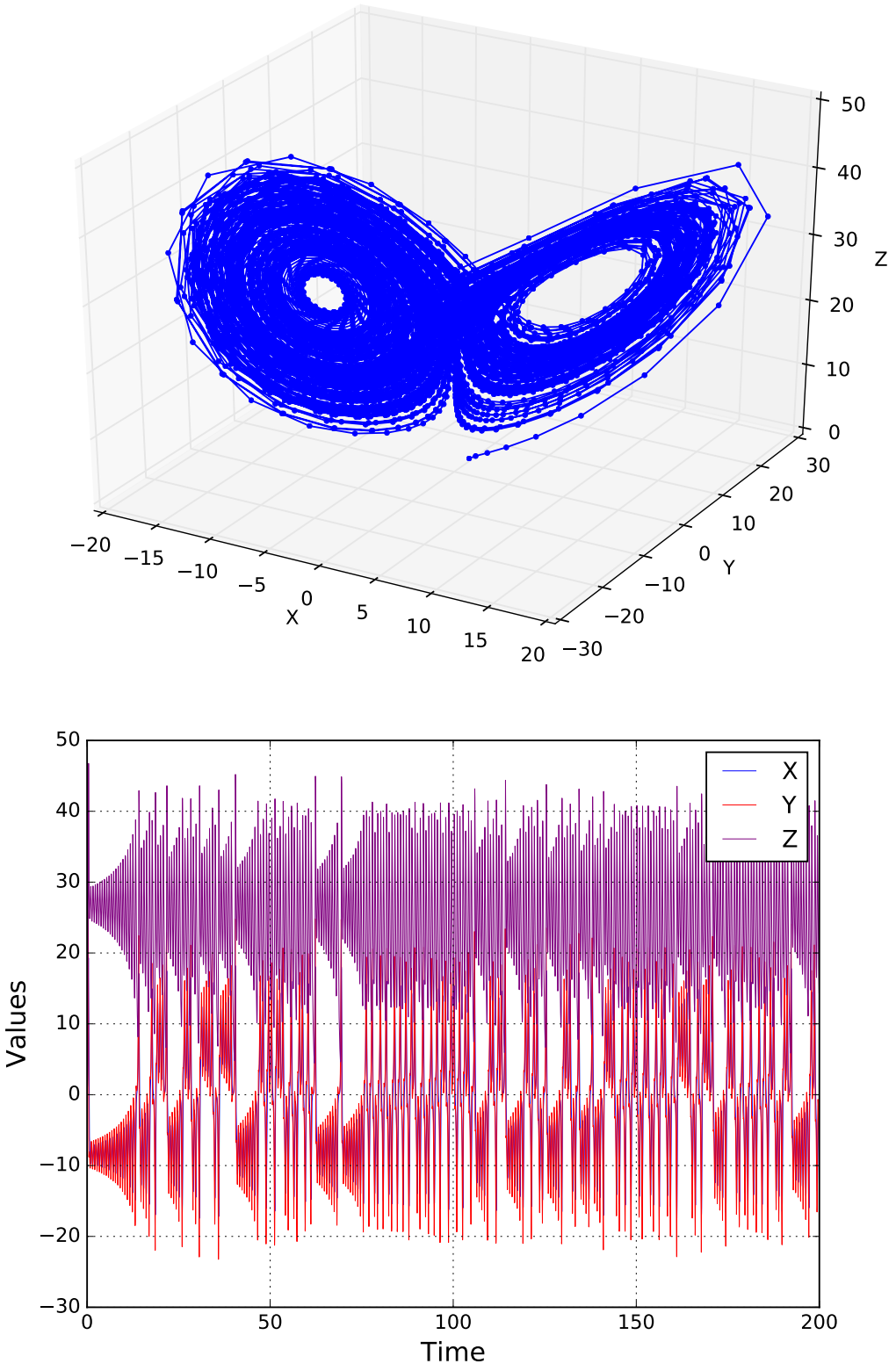


Figure 5.8: "Lorenz attractor" simulated using OpenModelica.

5.3 Distributed Algorithm Based on Semi-implicit Solver Coupling

A semi-implicit approach is different from a fully implicit approach in that it does not need to reevaluate the complete trajectory from "communication point" T_n to T_{n+1} [BS12]. If there are small internal integration points $t_0, t_1, t_2, \ldots, t_n$ for a sub-component F_1, then in order to perform a fixed point iteration which makes the solution to converge, it does not need to go through all the points $t_0, t_1, t_2, \ldots, t_n$. This makes it advantageous in cases where reevaluation of the complete macro time step is not possible.

The detailed mathematical description of the algorithm can be seen in [SBE+14], where it is named as Interface Jacobian-based Co-Simulation Algorithm (IJCSA). In brief, if only two systems $\mathcal{S}_1$ and $\mathcal{S}_2$ are considered, with respective inputs U_1, U_2 and outputs Y_1, Y_2 then $\mathcal{I}_1$ and $\mathcal{I}_2$ are their compatibility constraints defined as

$$\begin{aligned} \mathcal{I}_1(\mathcal{S}_1(U_1), \mathcal{S}_2(U_2), U_1, U_2) = 0 \\ \mathcal{I}_2(\mathcal{S}_1(U_1), \mathcal{S}_2(U_2), U_1, U_2) = 0 \end{aligned} \tag{5.10}$$

According to IJCSA method, the right hand side of the equation 5.10 are 0 only in the ideal condition, in reality they are non-zero values which can be named as $\mathcal{R}_1$ and $\mathcal{R}_2$. Values $\mathcal{R}_1$ and $\mathcal{R}_2$ are in fact residual to the Newton iteration applied to the composite system. Components involved in the Newton iteration are given below

$$\begin{bmatrix} \frac{\partial \mathcal{R}_1}{\partial U_1} & \frac{\partial \mathcal{R}_1}{\partial U_2} \\ \frac{\partial \mathcal{R}_2}{\partial U_1} & \frac{\partial \mathcal{R}_2}{\partial U_2} \end{bmatrix} \begin{bmatrix} \Delta U_1 \\ \Delta U_2 \end{bmatrix} = - \begin{bmatrix} \mathcal{R}_1 \\ \mathcal{R}_2 \end{bmatrix} \tag{5.11}$$

The system is ultimately converted to following, the details of the conversion can be seen in [SBE+14].

$$\underbrace{\begin{bmatrix} I & -\frac{\partial Y_2}{\partial U_2} \\ -\frac{\partial Y_1}{\partial U_1} & I \end{bmatrix}}_{\bar{\bar{\mathcal{J}}}} \begin{bmatrix} \Delta U_1 \\ \Delta U_2 \end{bmatrix} = - \begin{bmatrix} \mathcal{R}_1 \\ \mathcal{R}_2 \end{bmatrix} \tag{5.12}$$

Here $\frac{\partial Y_i}{\partial U_i}$ is the derivative of output with respect to input, and $\bar{\bar{\mathcal{J}}}$ is the Jacobian matrix. The method is useful using FMI 2.0, because it supports Jacobian information[BOAk+12]. In the absence of Jacobian information, either estimates to the Jacobian can be used, or a quasi-Newton scheme can be used in place of Newton iteration. Until the time of writing there is no open source simulation package which supports FMI 2.0, so the presented work is focusing on FMI 1.0, and the approach without Jacobian is presented here.

5.3.1 Quasi-Newton Methods Replacing Newton Method

To replace the need of Jacobian information in semi-implicit coupling the author has developed a technique to use quasi-Newton methods for fixed point iteration, in place of Newton's method for fixed point iteration. In order to distinguish the difference between a quasi-Newton method and Newton method, let us consider a quasi-Newton method presented by Broyden [Bro65]. To briefly cover how Broyden's method works, it must be compared to Newton's method. Newton's method defines a sequence x_k for the approximation of x^* by

$$\bar{x}_{k+1} = \bar{x}_k - \bar{\bar{\mathcal{J}}}_g^{-1}(\bar{x}_k) g(\bar{x}_k), \qquad k = 0, 1, 2, \ldots \tag{5.13}$$

The Broyden changes the equation 5.13 into following

$$\bar{x}_{k+1} = \bar{x}_k - \bar{\bar{\mathcal{B}}}_k^{-1}(\bar{x}_k)g(\bar{x}_k), \qquad k = 0, 1, 2, \ldots \tag{5.14}$$

The replacement of inverse Jacobian matrix $\bar{\bar{\mathcal{J}}}_g^{-1}$ is $\bar{\bar{\mathcal{B}}}_k^{-1}$, what we may call as Broyden Matrix. Transforming equation 5.13 into a co-simulation problem, and comparing with equation 5.12 we get following

$$\bar{\bar{\mathcal{J}}}\bar{x}_k = \bar{\mathcal{R}}_k \tag{5.15}$$

Replacing Jacobian matrix by Broyden matrix

$$\bar{\bar{\mathcal{B}}}_k\bar{x}_k = \bar{\mathcal{R}}_k \tag{5.16}$$

Equation 5.16 means that at each fixed point iteration Broyden matrix $\bar{\bar{\mathcal{B}}}$, inputs $\bar{x}$ and residual $\bar{\mathcal{R}}$ are updated simultaneously. The only known thing is the so called function $g(\bar{x})$ which gives predicted outputs given known inputs. The function $g(\bar{x})$ comprises of the output values of all the FMUs when their inputs are set to $\bar{x}$. The system is evaluated iteratively until the norm of residual becomes less than a tolerance value TOL.

Generalizing the problem, above statement shows that it is just the same as root finding problem of a known function $g(x)$, with unknown x. Besides Broyden there have been other techniques proposed to find the roots of a functions. All the proposed methods update an approximation to Jacobian Matrix or inverse Jacobian Matrix, along with the inputs $\bar{x}_k$ and residual $\bar{\mathcal{R}}_k$. Any of these methods can be used here, including Broyden's method. Another option is to use Newton Krylov method [KK04], which is proven to be good for large systems. Whatever method is used in practice, for generalization, in algorithm 3 it is abstractly represented as $\mathcal{X}$ at line 16. Similarly , the abstract representation of update of inputs x_k is represented by $\mathcal{Z}$ at line 17. The residual is calculated by the function *Residual* defined at line 26. The function is called at line 15 to pass on the calculated residual to the functions calculating estimate of Jacobian matrix and the inputs.

5.3.2 Description of the Algorithm

First, it is important to clarify an imminent ambiguity. While using algorithm 3 there seem to be little difference in inputs and outputs of subsystems, which is true because they only represent a different point of view to the same set of values. This is due to the reason that here only those outputs are of interest which are inputs to other subsystems. The outputs which are not connected to any other subsystem virtually play no role in convergence of the solution. Considering this situation with respect to equations 5.13 and 5.14, if the outputs of function g are defined as $\bar{y}_k = g(\bar{x}_k)$, then $\bar{x}_{k+1} = \bar{y}_k$. Meaning, the same outputs at macro time step T_n are going to be inputs for the first fixed point iteration at macro time step T_{n+1}, the only thing which has to be done is to create a mapping from outputs to inputs.

As a brief overview, the purpose of applying quasi-Newton iteration on a co-simulation problem, at a certain macro time step T_n, is to find those outputs of all subsystems which when fed as inputs to respective subsystems do not change the outputs more than a tolerance value TOL. Looking at the implementation of the concept in algorithm 3 (semi-implicit algorithm), it is evident that it is much the same as algorithm 2 (WR algorithm). Just like WR algorithm it has two loops, one, the main simulation loop which starts from line 7, the second is fixed point iteration loop which starts from line 14. The simulation loop is responsible for advancing time, and fixed point iteration loop is responsible for convergence. In section 5.2.3, by the

Algorithm 3 Semi-implicit Algorithm.

Code for p_{master}:
  Procedure Main()
    $time \leftarrow 0$
    send ⟨Send Initial Outputs, $time$⟩
    GetUpdates() // Defined in listing for algorithm 2
    $\bar{x}_k \leftarrow updates$
    while $time \leq simulationEndTime$ **do**
      $time \leftarrow time + step$
      send ⟨Advance Time, $time$⟩ to all "slave" processes.
      send ⟨Send Outputs, $time$⟩ to all "slave" processes.
      GetUpdates()
      $\bar{x}_k \leftarrow [value(x) \mid x \in updates]$
      $\bar{\mathcal{R}} \leftarrow \infty$
      while $\bar{\mathcal{R}} > TOL$
        $\bar{\mathcal{R}} \leftarrow Residual(\bar{x}_k, time)$
        $\bar{\bar{\mathcal{B}}}_k^{-1} = \mathcal{X}(\bar{\mathcal{R}}, \bar{x}_k, \bar{\bar{\mathcal{B}}}_k^{-1})$
        $\bar{x}_{k+1} = \mathcal{Z}(\bar{x}_k, \bar{\bar{\mathcal{B}}}_k^{-1})$
        $\bar{x}_k \leftarrow \bar{x}_{k+1}$
      end while
      send ⟨End Iteration, $time$⟩ to all "slave" processes.
      send ⟨TAR, $time$⟩ to p_{rti}
    end while
    send ⟨End Simulation⟩ to all "slave" processes
    terminate
  end

  Function $Residual(\bar{y}_k, time)$
    for each p^j_{slave}:
      send ⟨Update Inputs, $\{x \mid x \in \bar{y}_k \wedge name(x) \in inputs^j_{slave}\}, time$⟩ to p^j_{slave}
    send ⟨Recalculate, $time$⟩ to all "slave" processes
    send ⟨Send Outputs, $time$⟩ to all "slave" processes
    GetUpdates()
    $\bar{y}_{k+1} \leftarrow [value(x) \mid x \in updates]$
    return $\bar{y}_{k+1} - \bar{y}_k$
  end

  upon receiving ⟨Time Grant, $time$⟩:
    if $|updates| < NUMBER\_OF\_OUTPUTS$ **then**
      $ExecState \leftarrow UpdatesArrived$
    end if

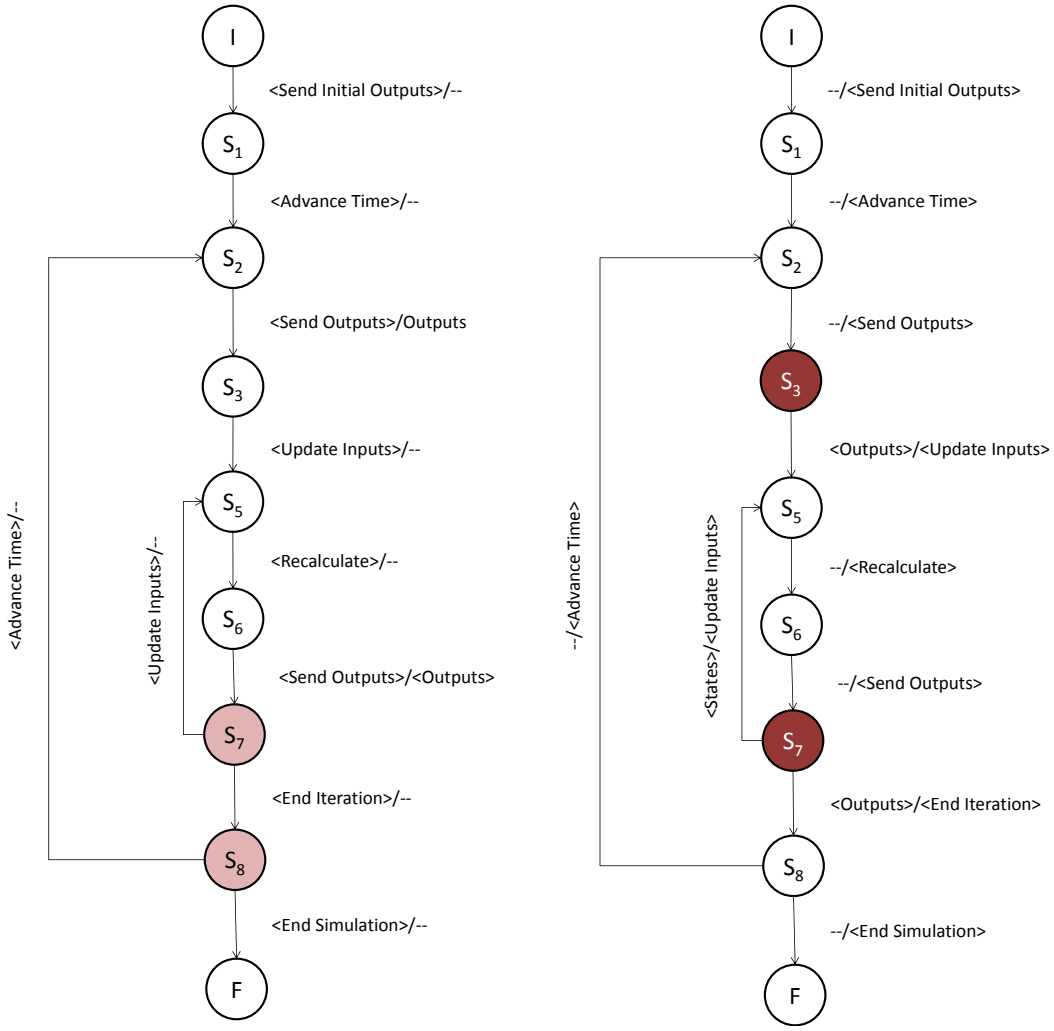
  upon receiving ⟨Update,$outputs$, $time$⟩:
    $\forall x \in updates\, \forall y \in outputs\, ((name(x) = name(y)) \rightarrow (value(x) \leftarrow value(y)))$.

Algorithm 3 Semi-implicit Algorithm (Continued).

Code for p_{slave}:
    upon receiving any message:
        store every message into *commandList*
    upon receiving ⟨Time Grant, *time*⟩:
        search for all "allowable" messages for the current state in *commandList*.
        execute the action on the first message found.
        send ⟨TARA,*time*⟩ to p_{rti} except in case of ⟨End Simulation⟩ and ⟨End Iteration⟩

    action for ⟨Send Initial Outputs, *time*⟩:
        Initialize the FMU, and send the initial outputs to p_{master} in from of ⟨Update,$outputs$, *time*⟩
    action for ⟨Advance Time, *time*⟩:
        Integrate the FMU to *time*
    action for ⟨Send Outputs, *time*⟩:
        send the outputs in form of ⟨Update,$outputs$, *time*⟩ to p_{master}
    action for ⟨Update Inputs,$input$, *time*⟩:
        set the respective input variable of FMU to $input$
    action for ⟨Recalculate, *time*⟩ :
        integrate the last integration step once again (with updated inputs).
    action for ⟨End Iteration, *time*⟩ :
        send ⟨TAR, *time*⟩ to p_{rti}
    action for ⟨End Simulation⟩:
        terminate

Code for p_{rti}:
    upon receiving ⟨TARA, *time*⟩ from p_i:
        send all messages queued for p_i to p_i with time stamp ≤ *time*
        upon confirmation of sending all updates send ⟨Time Grant⟩ to p_i
    upon receiving ⟨TAR, *time*⟩ from p_i:
        send all messages queued for p_i to p_i with time stamp ≤ *time*
        upon confirmation of sending all updates send ⟨Time Grant⟩ to p_i
    upon receiving ⟨Update,$outputs$, *time*⟩ from p_i:
        enqueue $outputs$ with time stamp *time*, for subscribing process p_{master}
    upon receiving any other command ζ from p_{master}:
        enqueue the command ζ for all subscribing "slave" processes.



(a) State machine for **slave** of semi-implicit algorithm **(b)** State machine for **master** of semi-implicit algorithm

Figure 5.9: Master and slave state machines. Slave state machine has branching states highlighted, while master state machine has synchronization states highlighted.

help of equation 5.3 and lemma 1 it is already proved that if master and slave have a simulation relationship and both necessarily pass through a synchronization state after passing through a branching state, then there is no possibility that the commands sent in order from master are not executed in the same order by slave. Looking at figure 5.9 it is clear that the same condition is valid in state machines of master and slave. So the execution of semi-implicit algorithm is also synchronized and will give the desired results.

The messages used by algorithm 3 are following.

- ⟨**Send Initial Outputs**⟩: The command is sent to "slave" processes to ask them to send the initial values of the outputs. The slave in response sends the values of outputs using the ⟨Update⟩ message.
- ⟨**Send Outputs**⟩: The message is used to ask slaves to send their newly calculated values of outputs in the form of ⟨Update⟩ message.
- ⟨**Update Inputs**⟩: The message is sent to ask a slave to update its input variables to the values sent with the message.

- ⟨**Recalcualte**⟩: When slave receives the message, it has already updated inputs, now it recalculates the outputs for that certain point in *time*. The *time* is sent as a parameter of the message.
- ⟨**Advance Time**⟩: On receiving the message, FMU-Federate integrates the FMU to *time*, which is provided as a parameter of the message.
- ⟨**End Iteration**⟩: The message is sent to inform the FMU-Federates that the fixed point iteration has converged, so they can close the internal integration procedure up to *time*, which is a parameter of the message.
- ⟨**End Simulation**⟩: The message asks an FMU-Federate to abandon all processing and go to **terminate** state.

5.3.3 Complexity of Semi-implicit Algorithm

Analysis of the message complexity of semi-implicit algorithm is very similar to complexity analysis of WR algorithm. Both algorithms have two loops, one the simulation loop, and second the fixed point iteration loop. The simulation loop in semi-implicit algorithm is just the same as WR algorithm, while the fixed point iteration loop is different because its implementation may vary, based on the method chosen for it. An in depth analysis of the complexity of different Jacobian free iterative methods is given in [KK04]. Based on the arguments presented in section 5.2.4 the message complexity of semi-implicit algorithm is also dependent on the number of step n_s and an upper bound on the number of fixed point iterations, i.e.

$$\mathcal{M}_{si}(n_s, I_{max}, n_c) = O(n_s I_{max} p_c) = O(n_s) \tag{5.17}$$

Here it is an added responsibility of the simulation engineer to use such quasi-Newton method for fixed point iteration which ensures to keep the upper bound on fixed point iterations I_{max} to minimum, while allowing to increase the step size to maximum.

5.3.4 Case Study

The same system described in equation 5.9 is used here to demonstrate the results. In case of semi-implicit algorithm it should be noted that the results can vary a lot based on the quasi-Newton scheme used for the fixed point iteration. For example in the given example if in place of Newton Krylov method, an older Broyden method is used, then the system does not solve at all after few macro steps. Nevertheless, describing the differences in iterative techniques is out of the scope of this document, any book on iterative methods can be consulted for this, such as [OR00]. In the presented example Newton Krylov method with LGMRES [KK04] is used for quasi-Newton iteration.

The reference solution for the "Lorenz attractor" is already given in figure 5.8. To avoid repetition it is not copied here. The error distribution, or the deviation from the reference solution can be seen in figure 5.10. Although the detailed comparison of the fully-implicit and semi-implicit method will be presented in section 6.4, but here the results of the simulated system can be compared by examining figure 5.7 and figure 5.11. The prominent shape of the "Lotenz attractor" can be seen in both the figures. Although, the individual curves plotted against the simulation time are hard to comprehend due to their rapid changing values, yet the overall shape of the curves is very much the same.

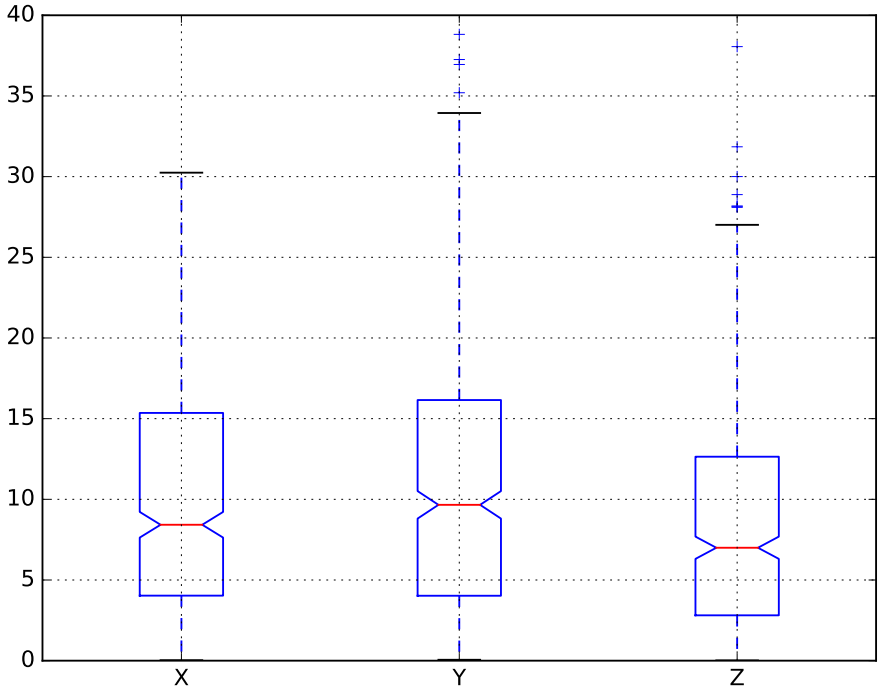


Figure 5.10: Difference of results between results of OpenModelica and semi-implicit algorithm for the problem of "Lorenz attractor".

To examine the error distribution of the results produced by semi-implicit and full-implicit algorithms with respect to the reference solution produced by OpenModelica, figure 5.6 and figure 5.10 have to be observed. Comparing both figures, it is clear that the median of error for variable x in figure 5.10 is slightly higher than the one in figure 5.6. The maximum error value is also slightly more than the one in figure 5.10. Both these indicators suggest that the semi-implicit algorithm has slightly more error than the fully-implicit algorithm, in the solution of variable x. The same situation can be seen for variable y. For variable z the situation is opposite. Semi-implicit algorithm has the median and maximum error value slightly less than the one of full-implicit algorithm. By and large, the differences in error distribution are not high, and it can be said that two methods can be used interchangeably to suite the requirements at hand. Further aspects of their comparison are delayed till section 6.4.

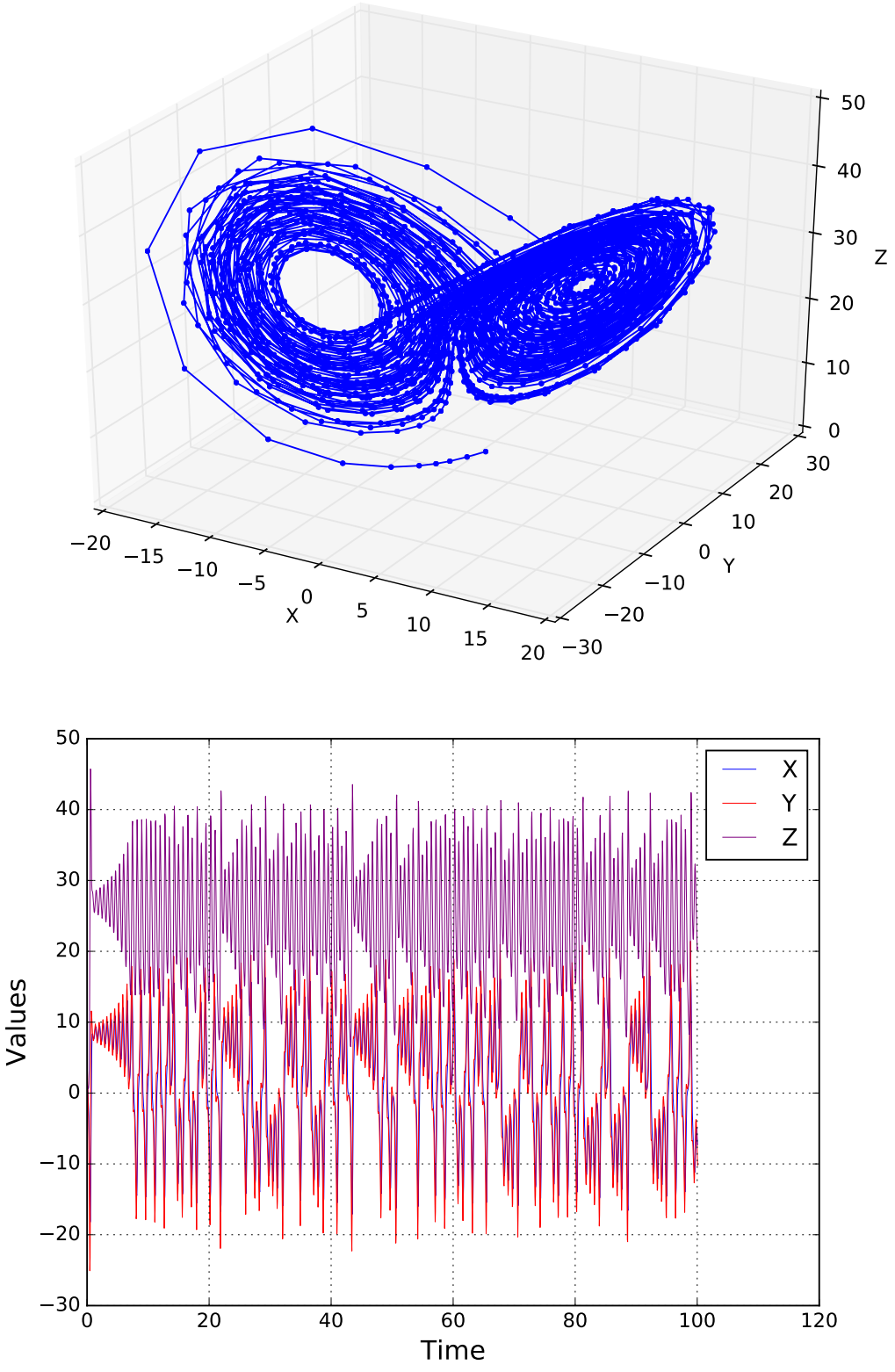


Figure 5.11: "Lorenz attractor" simulated using the semi-implicit algorithm, the "step size" is 0.05 here.

5.4 Distributed Algorithm for Strong Coupling Using SUNDIALS

As described earlier in section 3.3.4, research community divides coupling of models in two main categories, weak coupling and strong coupling. Aforementioned techniques were related to the weak coupling techniques. The present section describes a method to use HLA and FMI for strong coupling. In this type of coupling there is a single solver which solves the complete system. The only difference is that the system to be solved is distributed in different subsystems. For example suppose a model of a house hold consumption is developed using OpenModelica, and the model is exported in form of an FMU. While a model of power grid is developed using another simulation package, for example OpenDSS [1]. The model of power grid is also exported in the form of an FMU. Now to see the effect of load of n number of houses on the power grid, n instances of house FMU can be simulated with the FMU of power grid. The solver in this case can be any, but when already developed and tested solvers are available, it is better to use them. SUNDIALS [HBG+05] is one such solver, which is established as one of the efficient and accurate solvers in practice.

5.4.1 Description of the Algorithm

The algorithm 4 (strong coupling algorithm) is a little different from the master-slave algorithms described before. It has only one loop starting at line 8. The reason is quite obvious that the convergence is only an internal matter of SUNDIALS. The SUNDIALS solver is represented here by $\mathbb{SD}$ used at line 10. The solver function takes another function as a parameter represented here by $\mathcal{RHS}$. The SUNDIALS solver $\mathbb{SD}$ expects the $\mathcal{RHS}$ to evaluate the right hand side of the equation 5.18

$$\dot{\bar{y}} = f(t, \bar{y}), \qquad \bar{y}(t_0) = \bar{y}_0, \tag{5.18}$$

and return $\dot{y}$. For details of how different functions of SUNDIALS library work, have a look at [HBG+05]. The only challenge here is to evaluate $\mathcal{RHS}$ while the subsystems, or in other words FMUs, are residing in separate process spaces. The communication between solver code and the FMU is done using the gluing code called FMU-Federate. The definition of the function $\mathcal{RHS}$ starts from line 20. The numerical solution of the given system is completely hidden inside function $\mathbb{SD}$, but use of time stepping mechanism is still present at line 9. This may confuse the reader. The time stepping scheme here is only to tell $\mathbb{SD}$ when user code wants to get the updated values of the state variables vector $\bar{y}$. When the $\mathcal{RHS}$ function is evaluated depends completely on the internal implementation of the $\mathbb{SD}$ function. However, user can request to get the updated values at a point in time, which is what code at line 9 is doing.

Just like the previous master-slave algorithms here too, there is no need to prove that the correct order of execution is retained because of the correct placement of synchronizing states. The state digrams are shown in figure 5.12. The commands sent from the master to a slave are given below.

- ⟨**Send Initial States**⟩: The command is sent to "slave" processes to ask them to send the initial values of the state variables. Slaves in response send the values of state variables using the ⟨Update⟩ message.
- ⟨**Send States**⟩: The message is used to ask slaves to send their newly calculated values of state variables in the form of ⟨Update⟩ message.

[1] http://smartgrid.epri.com/SimulationTool.aspx

Algorithm 4 Strong Coupling Algorithm.

1: Code for p_{master}:
2: **Procedure** Main()
3: $time \leftarrow 0$
4: send ⟨Send Initial States, *time*⟩
5: GetUpdates() // Defined in listing for algorithm 2
6: $\bar{y} \leftarrow updates$
7: Initialize SUNDIALS
8: **while** $time \leq simulationEndTime$ **do**
9: $time \leftarrow time + step$
10: $\bar{y} = \mathbb{SD}(\mathcal{RHS}, time)$
11: send ⟨Send States, *time*⟩ to all "slave" processes.
12: GetUpdates()
13: $\bar{y} \leftarrow updates$
14: send ⟨End Iteration, *time*⟩ to all "slave" processes.
15: send ⟨TAR, *time*⟩ to p_{rti}
16: **end while**
17: send ⟨End Simulation⟩ to all "slave" processes
18: **terminate**
19: **end**

20: **Function** $\mathcal{RHS}(\bar{y}, time)$
21: for each p^j_{slave}:
22: send ⟨Update States, $\{x \mid x \in \bar{y} \wedge name(x) \in states^j_{slave}\}, time\rangle$ to p^j_{slave}
23: for each p^j_{slave}:
24: send ⟨Update Inputs, $\{x \mid x \in \bar{y} \wedge name(y) \in inputs^j_{slave}\}, time\rangle$ to p^j_{slave}
25: send ⟨Send Derivatives, *time*⟩ to all "slave" processes
26: GetUpdates()
27: $\dot{\bar{y}} \leftarrow updates$
28: return $\dot{\bar{y}}$
29: **end**

30: upon receiving ⟨Time Grant, *time*⟩:
31: **if** $|updates| = NUMBER\_OF\_STATES$ **then**
32: $ExecState \leftarrow UpdatesArrived$
33: **end if**

34: upon receiving ⟨Update,$\bar{y}$, *time*⟩:
35: $\forall x \in updates\, \forall y \in \bar{y}\, ((name(x) = name(y)) \rightarrow (value(x) \leftarrow value(y)))$.

Algorithm 4 Strong Coupling Algorithm (Continued).

Code for p_{slave}:
    upon receiving any message:
        store every message into *commandList*
    upon receiving ⟨Time Grant, *time*⟩:
        search for all "allowable" messages for the current state in *commandList*.
        execute the action on the first message found.
        send ⟨TARA,*time*⟩ to p_{rti} except in case of ⟨End Simulation⟩ and ⟨End Iteration⟩

    action for ⟨Send Initial States, *time*⟩:
        Initialize the FMU, and send the initial states to p_{master} in from of ⟨Update, *states*, *time*⟩
    action for ⟨Send States, *time*⟩:
        send the state variables in form of ⟨Update,*states*, *time*⟩ to p_{master}
    action for ⟨Send Derivatives, *time*⟩:
        send the state derivatives in form of ⟨Update,*derivatives*, *time*⟩ to p_{master}
    action for ⟨Update States,*state*, *time*⟩:
        set the respective state variable of FMU to *state*
    action for ⟨Update Inputs,*input*, *time*⟩:
        set the respective input variable of FMU to *input*
    action for ⟨End Iteration, *time*⟩ :
        send ⟨TAR, *time*⟩ to p_{rti}
    action for ⟨End Simulation⟩:
        terminate

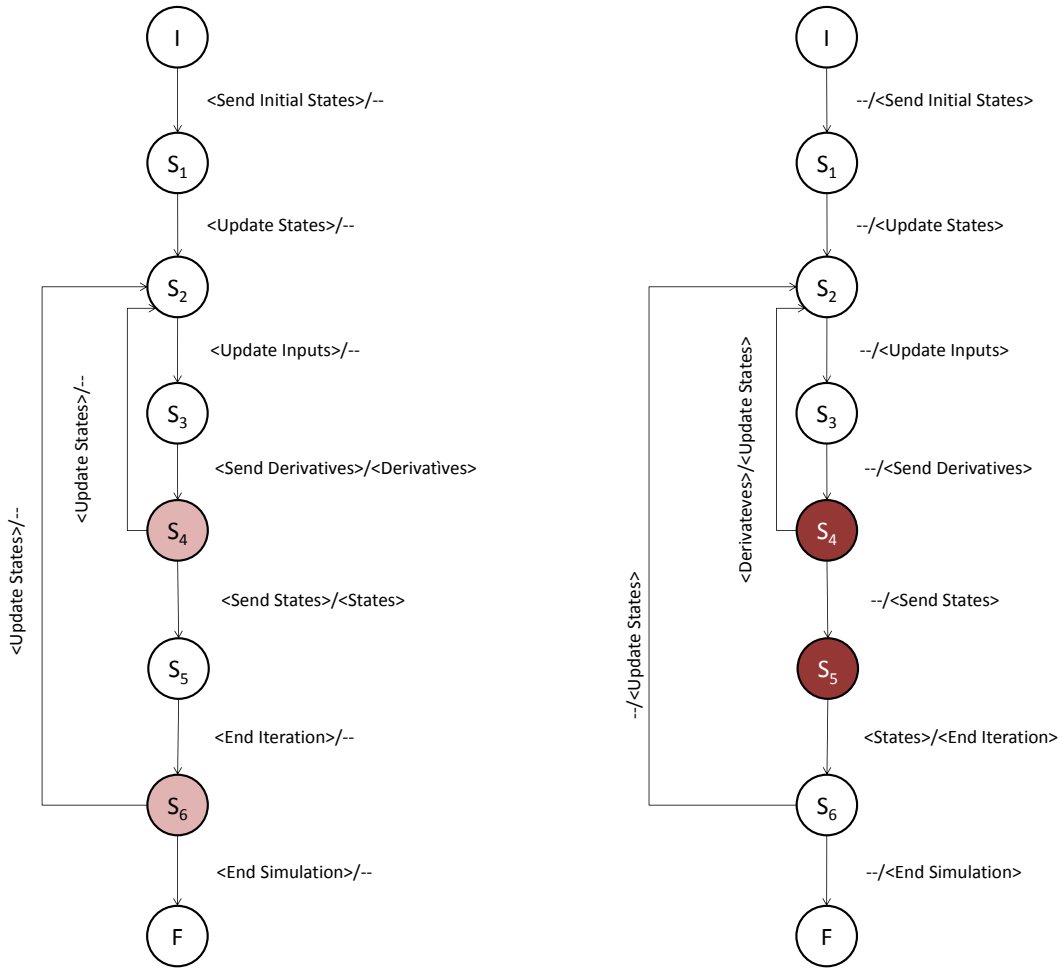
Code for p_{rti}:
    upon receiving ⟨TARA, *time*⟩ from p_i:
        send all messages queued for p_i to p_i with time stamp ≤ *time*
        upon confirmation of sending all updates send ⟨Time Grant⟩ to p_i
    upon receiving ⟨TAR, *time*⟩ from p_i:
        send all messages queued for p_i to p_i with time stamp ≤ *time*
        upon confirmation of sending all updates send ⟨Time Grant⟩ to p_i
    upon receiving ⟨Update,*y*, *time*⟩ from p_i:
        enqueue *y* with time stamp *time*, for subscribing processes p_{master}
    upon receiving any other command ζ from p_{master}:
        enqueue the command ζ for all subscribing "slave" processes.

- ⟨**Update Inputs**⟩: The message is sent to ask a slave to update its input variables to the values sent with the message.

- ⟨**Update States**⟩: The message is sent to ask a slave to update its state variables to the values sent with the message.

- ⟨**End Iteration**⟩: The message is sent to inform an FMU-Federate that one macro time step of the simulation is completed, so the FMU-Federate can accept the current state variables as the final one for the current macro time step.

- ⟨**End Simulation**⟩: The message asks an FMU-Federate to abandon all processing and go to **terminate** state.



(a) State machine for **slave** of strong coupling algorithm **(b)** State machine for **master** of strong coupling algorithm

Figure 5.12: Master and slave state machines. Slave state machine has branching states highlighted, while master state machine has synchronization states highlighted.

5.4.2 Complexity of Strong Coupling Algorithm

Algorithm 4 (strong coupling algorithm) may look similar to WR algorithm and semi-implicit algorithm, but in reality it is very different. The main simulation loop which starts at line 8, is not playing any part whatsoever in setting the macro step size, although it may look like. The reality is that the step size is completely under control of internal algorithms of SUNDIALS. The only thing set at line 9 is the next time of getting the updated values of state variables. So in reality it is not prudent to enumerate number of iterations in order to count the number of messages during the algorithm execution. The only way to count the message complexity of strong coupling algorithm is to realize how SUNDIALS as a solver works.

Let us suppose that $\Im$ is an advanced solver, which in present case is SUNDIALS. Looking at equation 5.18, every advanced solver $\Im$ has to evaluate the right hand side of equation 5.18 repeatedly, in order to evaluate the values of state variables, namely $\bar{y}$. In many cases the function f is not evaluated with strictly increasing values of independent parameter t[2]. After evaluating the function f at certain points in time and analyzing the rate of change, the solver $\Im$ performs the sensitivity analysis, whose purpose, among other things, is to find the optimal step size for solving the system. In modern solvers, the process of solving the system and sensitivity analysis goes hand in hand [SH05]. Due to this, modern solvers are good in identifying stiff portions of the system and vary the step size accordingly. Due to the involvement of sensitivity analysis, step size itself does not guarantee anything about the number of times the function f is evaluated during a simulation. Sometimes it is evaluated only to judge the stiffness of the system alone. Keeping all these things in mind it is hard to associate the number of steps with number of messages as it was done in WR algorithm or semi-implicit algorithm. What can be done is to associate the number of function evaluations with the number of messages.

Suppose the number of function calls during the solution of a system are n_f. The main simulation loop only contribute to get the updated values, suppose it has n_s iterations, with constant c_1 number of messages sent per iterations. Inspecting the $\mathcal{RHS}$ function it is clear that there are no loops inside this function so the messages sent during the execution of $\mathcal{RHS}$ function can be a constant number c_2. The message complexity of strong coupling is given below

$$\mathcal{M}_{sc}(n_s, n_f) = c_1 n_s + c_2 n_f = O\left(n_s + n_f\right) \tag{5.19}$$

Equation 5.19 does not choses anyone between n_s and n_f to be larger than the other, because it is not possible to generalize. For stiff systems like the one discussed in case study (section 5.4.3), n_f can be much larger than n_s, if the step size *step* in the algorithm is not chosen to be extremely small. While keeping the same value of *step*, if the system to be evaluated is a smooth one then n_f can be much less than n_s. The value of n_s also depends on how long the simulation have to be executed. It is also not possible to generalize the association of n_s and n_f among themselves. The nature of a system can vary during the time. At one point in time, if a system is stiff, then at another, it can be smooth [WH91].

5.4.3 Case Study

The solving capabilities of SUNDIALS solver do not need any proof. The solver has been used in many industrial and research projects. The purpose here is to demonstrate that the solver also works well in a distributed setting described here. The same example is chosen to demonstrate the results, which was used for WR algorithm i.e. “Lorenz attractor”. The description of the system can be seen in equation 5.9. The reference solution produced by OpenModelica can be seen in figure 5.8.

[2]The independent parameter t can be something other than the time, but in case of simulations it is mostly the time.

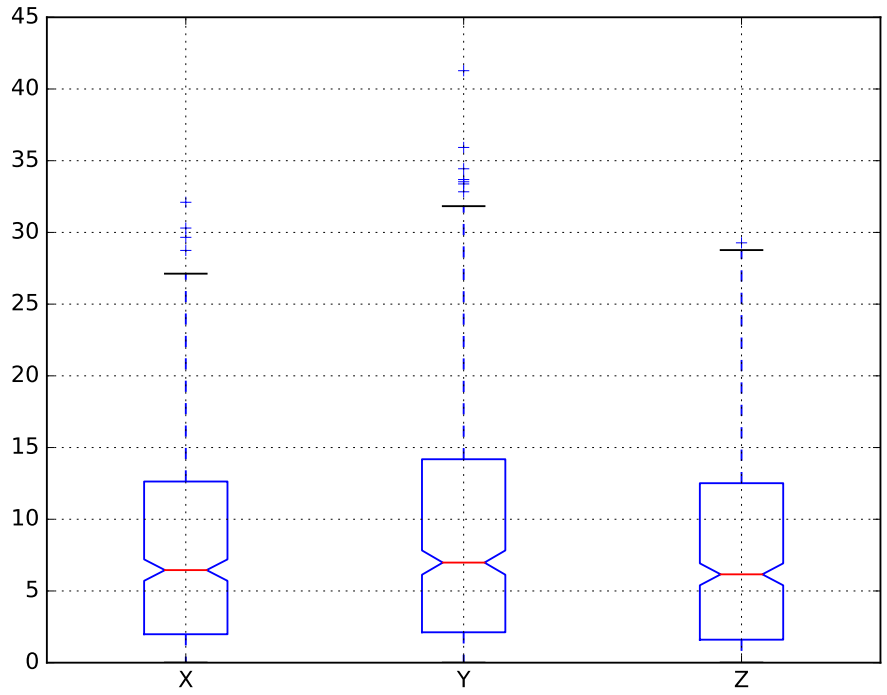


Figure 5.13: Difference of results between results of OpenModelica and strong coupling algorithm for the problem of "Lorenz attractor".

To see the error distribution of the solution produced by SUNDIALS with that of reference solution using OpenModelica, figure 5.13 has to be examined. Further comparing the figure 5.13 with figure 5.6 and figure 5.10, it is revealed that the error distribution is not much different in fully-implicit, semi-implicit and strong coupling algorithms. As SUNDIALS is a very advanced solver so it can be seen that the medians of errors for all variables x, y and z are less than the other two (semi-implicit and fully-implicit). The maximum error values are also much less. The difference is not too large though, especially not large enough to render the other two solutions (semi-implicit and fully-implicit) as useless. The application domain of strong coupling algorithm is completely difference from the semi-implicit and fully-implicit algorithms. The strong coupling algorithm is only applicable where the solvers are not distributed and the centrally located solver has a complete control over the distributed models. Semi-implicit and fully-implicit algorithms have distributed solvers, in contrast. The comparison of these methods is not justified, due to the vast differences in their application domains.

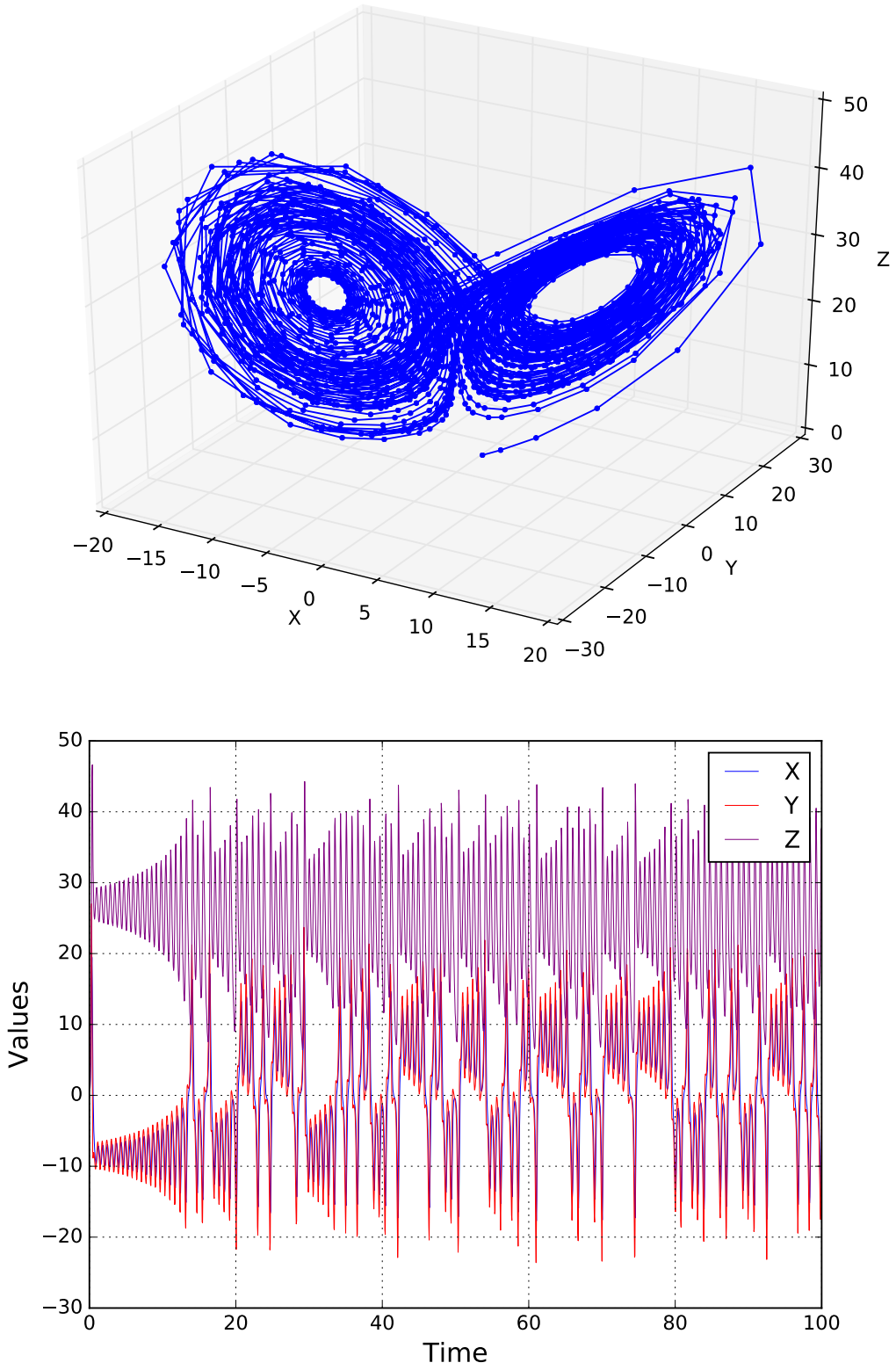


Figure 5.14: "Lorenz attractor" simulated using the strong coupling of FMUs, solved by SUNDIALS and connected by HLA, the "step size" is internally controlled by SUNDIALS.

5.5 Distributed Hybrid Co-Simulation Algorithm

Previous algorithms targeted solution of different continuous systems. Complex systems, though, mostly comprise of continuous and discrete parts. Especially, Cyber Physical Systems (CPS) are inherently hybrid in nature. The Information and Communication Technology (ICT) part is normally modeled as a discrete event simulation. Due to these reasons it is essential to look into hybrid simulations, using distributed components. As discussed in chapter 2 smart grid simulators try to combine one continuous and one discrete event simulator. In real life the problems may be more complex than that. For example, a complex energy system may have electric power and thermal energy interconnected with each other, along with the ICT portion for intelligent management and control. Most simulators designed for thermal energy simulation are not designed to model an electric grid. In such a scenario it is most beneficial to model and simulate the relevant parts in suitable simulation package and then produce unified results.

For these reasons, in the current section the WR algorithm is modified in a way to accommodate discrete events. As it is seen in section 5.2 that theoretically WR algorithm can accommodate any number of continuous systems to be coupled with each other, as far as they satisfy the contraction conditions described earlier. Similarly, when expanding the algorithm to hybrid systems, there can be any number of continuous and discrete subsystems. Only continuous subsystems have to fulfill the contraction conditions.

5.5.1 Description of the Algorithm

Listing 5 describes the hybrid simulation algorithm. One important difference of this algorithm from previous ones is, its inputs and outputs when communicated among processors have three elements rather than two. In section 5.2.1 it was discussed that inputs and outputs are communicated among processors in form of list of pairs (map), and each element has two parts, a **name** and **value**. Here each element has another part which is **time**, which is represented as $time(x)$, where x is an element of a list of inputs or outputs. It is important to mention the **time** of the updated values, because in case of discrete variables, it is important to know when their state has changed.

The main idea of hybrid simulation algorithm revolves around WR algorithm. If the lines from 19 to 24 are removed from the algorithm 5 then it will just be another form of a WR algorithm. Lines from 19 to 24 make the portion which is responsible for detecting and taking care of a discrete event. The discrete event itself is identified at the master level, when a slave sends a negative value as the update time of a state variable. The statement is listed at line 19.

As it was discussed in section 5.2 that WR algorithm assumed that there is no change from state variables to the outputs of a subsystem, which was a limitation. That limitation is removed in algorithm 5. Algorithm 5 does not passes on the inputs and state variables to the subsystems, rather the subsystems exchange the relevant outputs when master commands them to do so. This is done using the ⟨Share Data⟩ command. When a slave receives this command, it waits for the inputs from other slaves and sends its outputs to subscribing slaves. Similarly, now to go back one macro step T_{n-1} there is no need to tell each slave the values of their state variables, as each one can store them themselves from the last successful iteration. So in algorithm 5 ⟨Rewind⟩ command fulfills this job. There is no need to send explicit message to ask slaves to send their updated state variables or outputs. When the ⟨Advance Time⟩ command is received, the slave knows that after advancing the time to the desired value, it has to send its updated outputs back to the master. Because ⟨Advance Time⟩ command replaces ⟨Send States⟩ command so it becomes a synchronizing command. As figure 5.18 shows that the state machine for the hybrid algorithm is much more complex than previous algorithms, so the synchronization of the master and slave processes is also more

challenging. At few stages the synchronization has to be enforced. For this a special command ⟨Synch⟩ is introduced which asks the slaves to send their updated outputs. In fact, the only purpose is to introduce a synchronization point between two branching states as the requirement is stated in section 5.2.3.

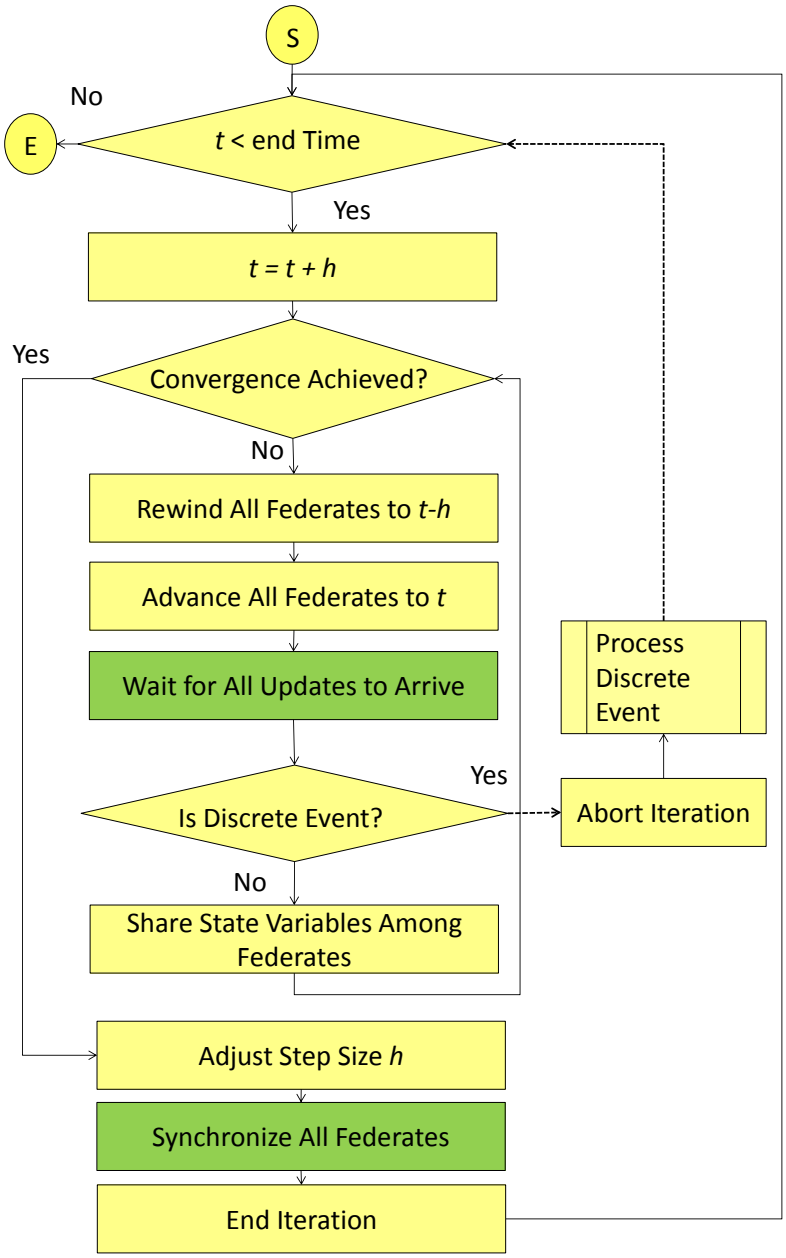


Figure 5.15: Main flow of hybrid simulation algorithm, excluding discrete event processing. The green states show the synchronization points.

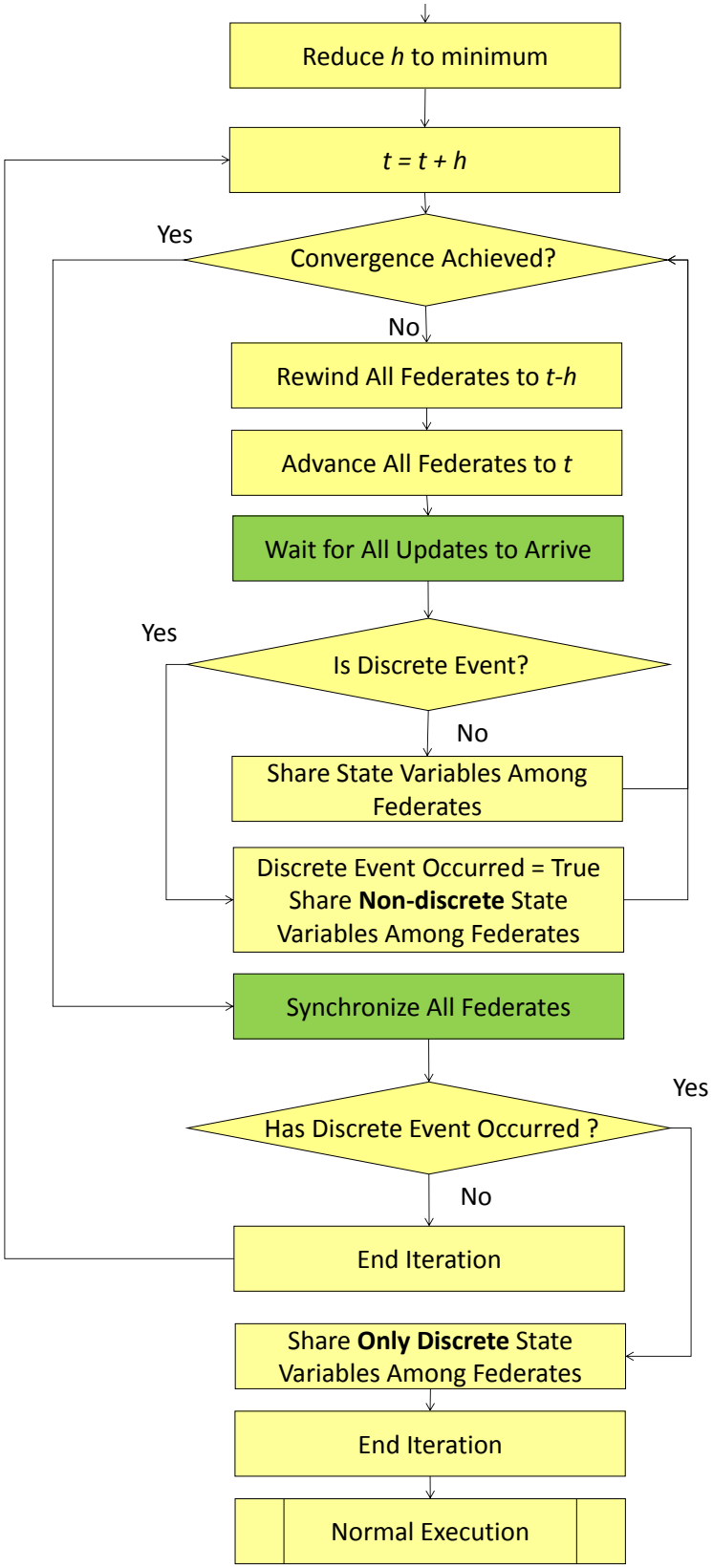


Figure 5.16: Discrete event processing of hybrid simulation algorithm, in presence of efficient event detection it can be called as stage-2 of event processing.

Algorithm 5 Hybrid Simulation Algorithm.

```
1:  Code for p_master:
2:      Procedure Main()
3:          currStates ← {(name(x), ∞) | x ∈ updates}
4:          while time ≤ simulationEndTime do
5:              time ← time + step
6:              discEvent ← False
7:              iterationStates ← {(name(x), ∞) | x ∈ updates}
8:              iterationCount ← 0
9:              while norm(iterationStates − currStates) > TOL do
10:                 iterationCount ← iterationCount + 1
11:                 if iterationCount > MAX then
12:                     throw exception "No convergence" and terminate
13:                 end if
14:                 send ⟨Rewind, time − step⟩ to all "slave" processes.
15:                 send ⟨Advance Time, time⟩ to all "slave" processes.
16:                 GetUpdates()            // Defined in listing for algorithm 2
17:                 iterationStates ← currStates
18:                 currStates ← updates
19:                 if ∃y | y ∈ updates ∧ time(y) < 0 then
20:                     discEvent ← True
21:                     time ← time − step
22:                     send ⟨Abort Iteration, time⟩ to all "slaves"
23:                     ProcessDiscreteEvent()
24:                 else
25:                     send ⟨Share Data, time⟩ to all "slaves"
26:                 end if
27:             end while
28:             if discEvent = False then
29:                 send ⟨Synch, time⟩ to all "slave" processes.
30:                 GetUpdates()
31:                 currStates ← updates
32:                 send ⟨End Iteration,time⟩ to all "slave" processes
33:                 send ⟨TAR,time⟩ to p_rti
34:             end if
35:         end while
36:         send ⟨End Simulation⟩ to all "slave" processes
37:         terminate
38:     end
```

Algorithm 5 Hybrid Simulation Algorithm (Continued).

```
39:  Procedure ProcessDiscreteEvent()
40:      eventOccured ← False
41:      while eventOccured = False do
42:          time ← time + δ
43:          iterationStates ← {(name(x), ∞) | x ∈ updates}
44:          while norm(iterationStates − currStates) > TOL do
45:              send ⟨Rewind, time − δ⟩ to all "slave" processes.
46:              send ⟨Advance Time, time⟩ to all "slave" processes.
47:              GetUpdates()
48:              iterationStates ← currStates
49:              currStates ← updates
50:              if ∃y | y ∈ updates ∧ time(y) < 0 then
51:                  eventOccured ← True
52:                  send ⟨Share Data Non-discrete, time⟩ to all "slaves"
53:              else
54:                  send ⟨Share Data, time⟩ to all "slaves"
55:              end if
56:          end while
57:          send ⟨Synch, time⟩ to all "slaves"
58:          GetUpdates()
59:          currStates ← updates
60:          if eventOccured = False then
61:              send ⟨End Iteration, time⟩ to all "slaves"
62:              send ⟨TAR,time⟩ to p_rti
63:          end if
64:      end while
65:      send ⟨Share Data Only Discrete, time⟩ to all "slaves"
66:      send ⟨End Iteration, time⟩ to all "slaves"
67:      send ⟨TAR,time⟩ to p_rti
68:  end

69:  upon receiving ⟨Time Grant, time⟩:
70:      if |updates| = NUMBER_OF_OUTPUTS then
71:          ExecState ← UpdatesArrived
72:      end if
73:  upon receiving ⟨Update, outputs,time⟩:
74:      ∀x ∈ updates ∀y ∈ outputs
             ((name(x) = name(y)) → (value(x) ← value(y) ∧ time(x) ← time(y))).
```

Algorithm 5 Hybrid Simulation Algorithm (Continued).

75: Code for p_{slave}:
76: upon receiving any command message
77: store every message into *commandList*
78: upon receiving ⟨Update Slave,*outputs*, *time*⟩:
79: $\forall x \in updates\, \forall y \in outputs\, ((name(x) = name(y)) \rightarrow (value(x) \leftarrow value(y))$.
80: upon receiving ⟨Time Grant, *time*⟩:
81: **if** $mode = ShareData$ **then**
82: **if** $|updates| = NUMBER\_OF\_FMU\_INPUTS$ **then**
83: $mode \leftarrow Execute$
84: **end if**
85: **else if** $mode = Execute$ **then**
86: search for all "allowable" messages for the current state in *commandList*.
87: execute the action on the first message found.
88: send ⟨TARA,*time*⟩ to p_{rti} except in case of ⟨End Simulation⟩ and ⟨End Iteration⟩
89: **end if**
90: action for ⟨Advance Time, *time*⟩:
91: Integrate the FMU to *time*
92: for each continuous output x of the FMU
93: send ⟨Update,$(y \mid value(y) = value(x) \wedge name(y) = name(x) \wedge time(y) = time)$,*time*⟩ to p_{master}
94: for each discrete output d of the FMU
95: **if** value of d has changed **then**
96: find the time of change (discrete event) and store in e
97: $e = -1 \times e$
98: **else**
99: $e \leftarrow time$
100: **end if**
101: send ⟨Update,$(y \mid value(y) = value(d) \wedge name(y) = name(d) \wedge time(y) = e)$,*time*⟩ to p_{master}
102: action for ⟨Rewind,*time*⟩:
103: set the time of FMU to *time*
104: set the *states* of FMU to the last saved *states* at *time*, if not saved then set to initial states.
105: action for ⟨Abort,*time*⟩:
106: set the time of FMU to *time*
107: set the *states* of FMU to the last saved *states* at *time*, if not saved then set to initial states
108: set the *inputs* of FMU to the last saved *inputs* at *time*, if not saved then set to initial values
109: action for ⟨Synch, *time*⟩:
110: send the *outputs* in form of ⟨Update,*outputs* ,*time*⟩ to p_{master}
111: action for ⟨Share Data, *time*⟩:
112: send the *outputs* in form of ⟨Update Slave,*outputs* ,*time*⟩ to all subscribing "slave" processes
113: $mode \leftarrow ShareData$
114: $updates \leftarrow \{(name(x), \infty) \mid x \in updates\}$
115: **while** $mode \neq Execute$ **do**
116: send ⟨TARA,*time*⟩ to p_{rti}
117: **end while**
118: Set the *inputs* of the FMU to the *updates* received

Algorithm 5 Hybrid Simulation Algorithm (Continued).

action for ⟨Share Data Non-discrete, *time*⟩:
    like ⟨Share Data⟩, send and receive only continuous valued *outputs*
action for ⟨Share Data Only Discrete, *time*⟩:
    like ⟨Share Data⟩, send and receive only discrete valued *outputs*
action for ⟨End Iteration, *time*⟩ :
    send ⟨TAR, *time*⟩ to p_{rti}
    save the *states* of FMU to be used for ⟨Rewind⟩ and ⟨Abort⟩ messages
    save the *inputs* of FMU to be used for ⟨Abort⟩ message
action for ⟨End Simulation⟩:
    terminate
Code for p_{rti}:
upon receiving ⟨TARA, *time*⟩ from p_i:
    send all messages queued for p_i to p_i with time stamp ≤ *time*
    upon confirmation of sending all updates send ⟨Time Grant⟩ to p_i
upon receiving ⟨TAR, *time*⟩ from p_i:
    send all messages queued for p_i to p_i with time stamp ≤ *time*
    upon confirmation of sending all updates send ⟨Time Grant⟩ to p_i
upon receiving ⟨Update,*outputs*, *time*⟩ from p_i:
    enqueue *outputs* with time stamp *time*, for subscribing process p_{master}
upon receiving ⟨Update Slave,*outputs*, *time*⟩ from p_i:
    enqueue *outputs* with time stamp *time*, for all subscribing slave processes p_j, where $i \neq j$
upon receiving any other command ζ from p_{master}:
    enqueue the command ζ for all subscribing "slave" processes.

Due to the relative complexity of the algorithm it was thought to illustrate the algorithm using flow chart diagrams [AGDCP15]. Figure 5.15 shows the main execution part of the hybrid algorithm, while figure 5.16 shows the part where discrete event is tackled. The green colored states show the synchronization points of the algorithm

The main idea behind tackling discrete event is to find the precise timing of the discrete event and then allow the discrete state variables to change their state and be communicated to other slaves. The process shown in figure 5.16 can be made faster by introducing stage wise finding of precise time of discrete event. In the first stage, shown in figure 5.17 the step size is fractionally reduced whenever the discrete event occurs during the iteration. Once the point is found closest to the discrete event where there is no discrete event, the execution is shifted to stage 2, which is already described in figure 5.16. Process of finding the discrete event in a stage wise manner, as shown in figure 5.17, is not part of the listing 5. It is omitted to let the listing remain easily understandable.

5.5.2 Communication Step Size Control

Step size control offers many advantages in any numerical integration algorithm. Implemented correctly, it can significantly enhance the performance of the algorithm. Here too, the communication step size control offers many advantages. Most importantly, in a distributed simulation more communication steps mean more communication, which means lesser performance. So increasing the communication step size to the maximum where the solution remains valid is very beneficial.

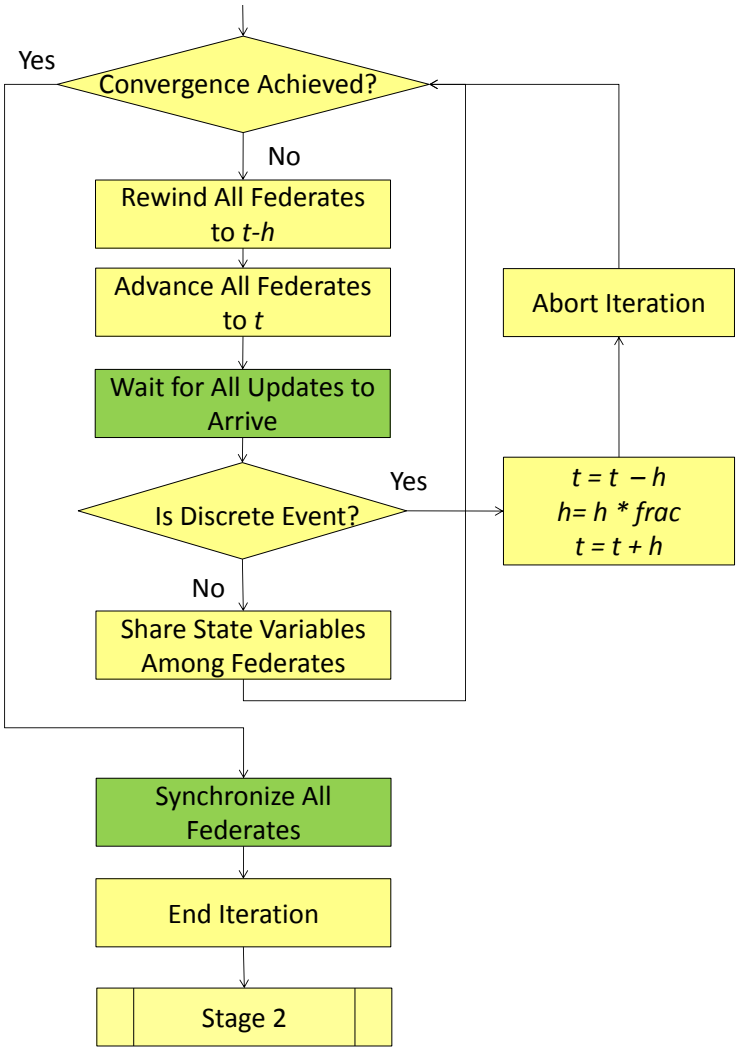


Figure 5.17: Stage-1 of discrete event processing of hybrid simulation algorithm. The purpose is to find the exact timing of event more quickly.

Looking at figure 5.2, it is easy to understand that separating the ODEs means that some or all of the state variables in a subsystem are going to grow without the knowledge of all other subsystems. Mathematically speaking, suppose there is a system given in equation 5.20

$$\dot{y} = f(y, p) \tag{5.20}$$

The state variable vector y contains n state variables $y = y_1, y_2, y_3, \ldots, y_n$. To perform the numerical integration of the system, if an implicit method is used, then the Jacobian of the system will be $n \times n$ matrix, containing partial derivatives of all the state variables with respect to each of them. Partitioning the system in two (equation 5.21) means that the Jacobian of each subsystem is also reduced to some degree. If $\hat{y} = y_1, y_2, y_3, \ldots, y_i$ and $\tilde{y} = y_{i+1}, y_{i+2}, y_{i+3}, \ldots, y_n$, then this means that state variables in $\hat{y}$ are being evaluated without their partial derivatives with respect to $y_{i+1}, y_{i+2}, y_{i+3}, \ldots, y_n$. Similar is the case of $\tilde{y}$. This causes divergence in the solution. If the divergence remains in a realm where the system remains defined, then it is possible to recover the error through fixed point iteration. If not, then this means that the gap between two communication steps is too large.

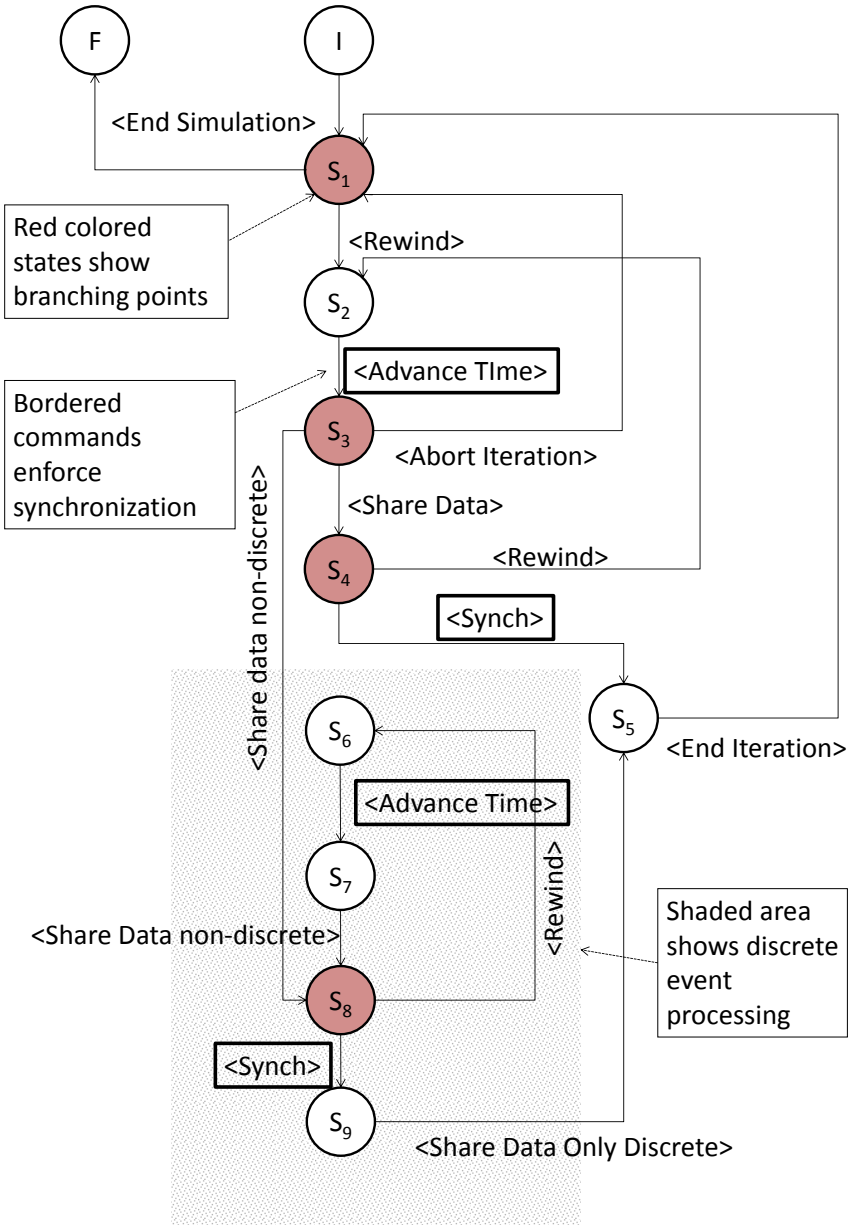


Figure 5.18: The slave state machine of hybrid simulation algorithm.

$$\begin{aligned} \dot{\hat{y}} &= \hat{f}(\hat{y}, \hat{p}) \\ \dot{\tilde{y}} &= \tilde{f}(\tilde{y}, \tilde{p}) \end{aligned} \tag{5.21}$$

Following the idea of divergence, apart from error tolerance, there is an additional parameter introduced, which is called as "divergence tolerance" tol_d. This is tolerance for the error caused by divergence. If the state variable vector, as a result of initial guess at the start of WR iteration, is y_i, and at the end of WR iteration after convergence is y_f, then the error e_d caused by divergence is given in equation 5.22.

$$e_d = |y_f - y_i|_2 \tag{5.22}$$

At the end of each WR iteration the communication step size is either increased or decreased by some percent, based on the fact that $e_d + \tau_0 \|y_f - y_i\|_{max} < tol_d$ or $e_d + \tau_0 \|y_f - y_i\|_{max} > tol_d$. Here τ_0 is a small positive value used for normalization. During processing of discrete event, the communication step size is intermediately reduced to minimum. After the discrete event, communication step size takes some time to recover its value. At that moment the mechanics of communication step size control become evident. Figure 5.19 shows the phenomenon by zooming into that situation.

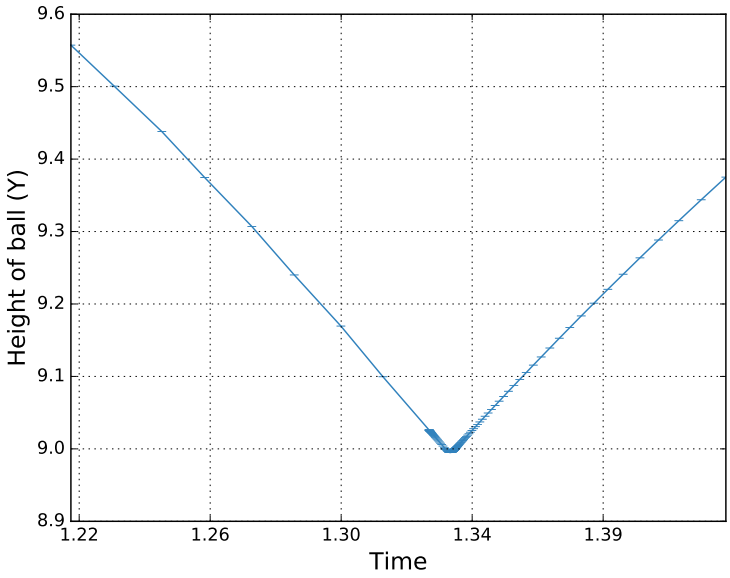


Figure 5.19: Variation of communication step size during processing of discrete event.

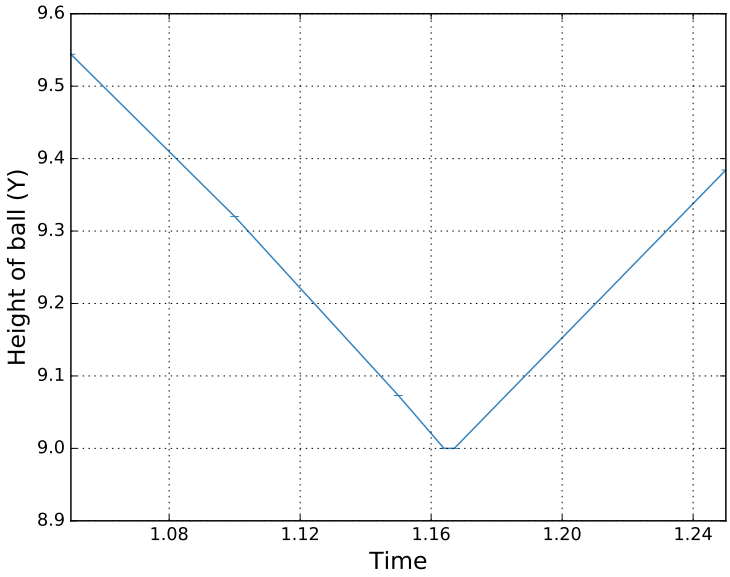


Figure 5.20: OpenModelica treatment of discrete event.

Algorithm 6 Discrete State Decisions

```
1: begin
2:     if (y < stair) then
3:         contact ← 1
4:     else if (y > stair) then
5:         contact ← 0
6:     end
7:     if (x − N + 1 + stair > 0) then
8:         stair ← stair − 1
9:     end
10: end
```

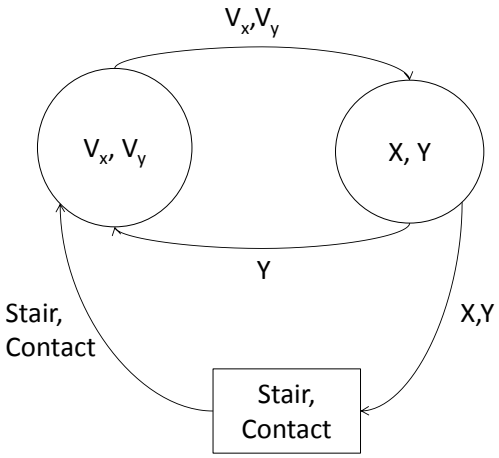


Figure 5.21: Division and interdependence of different subsystems. Arrows show information flow. Square element shows discrete component while circular show continuous.

5.5.3 Case Study

To test the algorithm, a test system is used for simulation. It is first simulated using the OpenModelica. The results are compared with the hybrid simulation algorithm presented earlier. The system is very popular hybrid system i.e. a ball being dropped from a height on stairs, namely a "bouncing ball on stairs". The system is given by the system of equations 5.23. The discrete part is given by the algorithm 6. Figure 5.21 shows how different FMU-Federates are associated with each other, via their state variables.

$$\begin{aligned} \dot{x} &= v_x \\ \dot{y} &= v_y \\ \dot{v_x} &= -c_0 v_x \\ \dot{v_y} &= -g - c_1 v_y - contact((y - stair)c_2 + c_3 v_y) \end{aligned} \tag{5.23}$$

Here g is the gravitational constant, while c_0,c_1, c_2 and c_3 are few constants facilitating the phenomena of friction, air resistance, damping and mass of the ball. The variables *stair* and *contact* represent discrete variables. The variable *contact* shows that the ball is in contact with the floor or not. When *contact* = 1 the system shifts its behavior immediately at that point. It is sometimes called as "behavioral state change".

The variable *stair* shows that on which step of the stair ball is currently bouncing. Initially, its value is $\mathcal{N}$ in algorithm 6.

For the presented run, the value of "divergence tolerance" was $tol_d = 1 \times 10^{-3}$. Although, the value is relatively large, using a smaller value makes results more accurate, but that causes more communication steps and hence performance deteriorates. This is an area which needs further refinement in the algorithm.

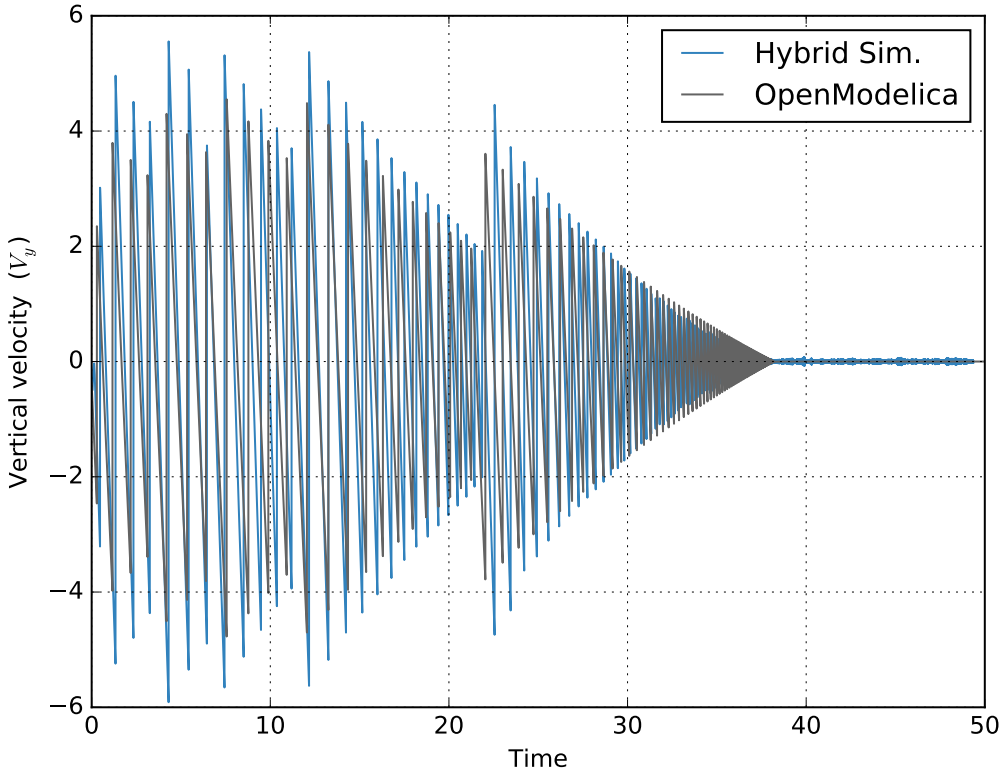


Figure 5.22: Comparative results produced by hybrid simulation algorithm and OpenModelica for variable v_y in equation 5.23. The variable v_y represents the vertical velocity of the ball.

Figures [5.22 - 5.26] show the results produced by OpenModelica and hybrid simulation algorithm, for the presented example. Especially examining figure 5.22 it can be seen that the vertical velocity of the ball changes abruptly when the ball is in contact with the ground. This is a point when the line in figure 5.22 moves upward. Its decrement is a lot slower than its abrupt increment and change in direction. The situation is specifically challenging for a solver of a partial system, because at this point the subsystem under the influence of external discrete events behaves abruptly. The fact that under such influence the WR iteration is able to converge back to the legitimate solution is appreciable. Figure 5.23 and figure 5.24 should be examined in conjunction with figure 5.22. In figure 5.23 the bouncing behavior of the ball can be seen, plotted over the simulation time, while in figure 5.24 the plot shows how the trajectory of the ball would be in real life, as the ball position is plotted on X-Y plan. It is also clear from these figure that there are some differences in the results. The difference between results are obvious due to the completely different treatment of events in OpenModelica DASSL algorithm. Figure 5.20 shows how OpenModelica cuts the contact dynamics out, and converts the system into a piecewise continuous systems.

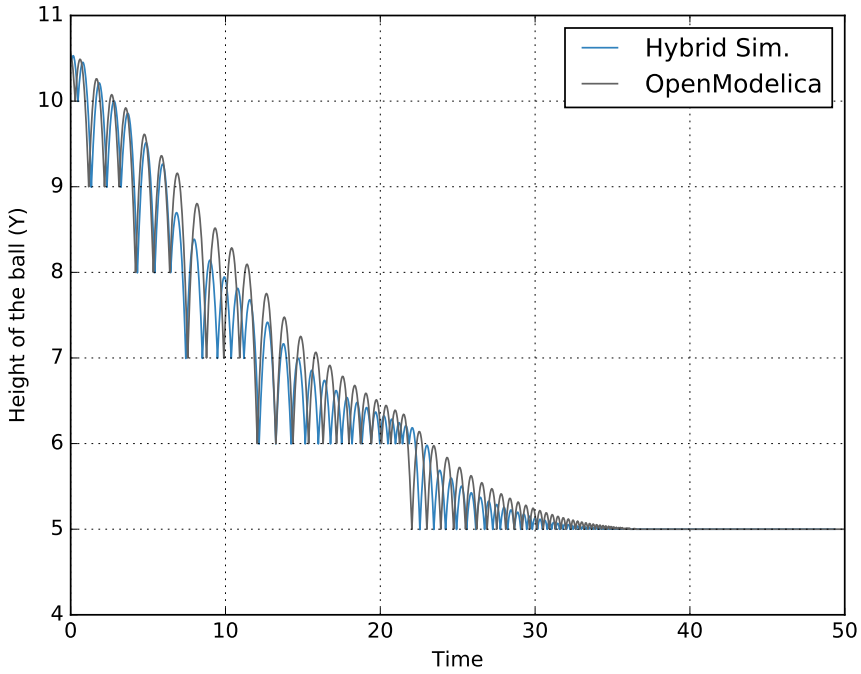


Figure 5.23: Comparative results produced by hybrid simulation algorithm and OpenModelica for variable y in equation 5.23. The variable y represents the height of the ball.

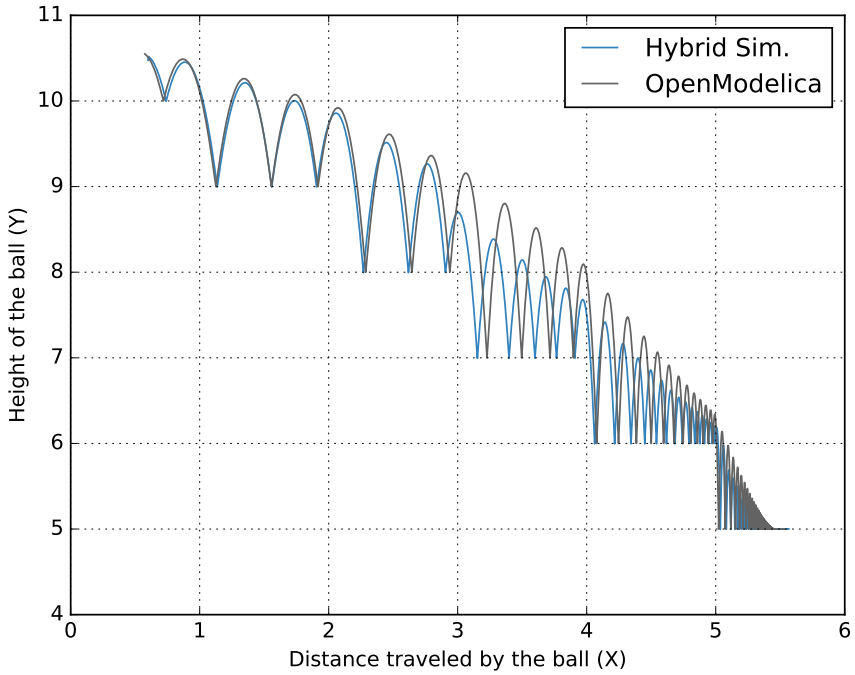


Figure 5.24: Comparative results produced by hybrid simulation algorithm and OpenModelica for both variables x and y in equation 5.23.

For example, Lundvall et al. [LFB08] describe how OpenModelica changes a system of DAEs into a hybrid system of DAEs, separating continuous part and a discrete part. This and many other simplification methods are examples of those advantages which monolithic simulation packages have. A distributed algorithm with current state of technologies cannot simplify the system as such. Most importantly, this type of simplification is something which a modeler may not wish to apply in complex simulations. One purpose of developing complex simulations is to understand any phenomenon which is difficult to experiment in real life. In order to compensate the empirical evidence of working of a theory about a complex phenomenon, modeler tries to verify the theory by simulating the real life phenomenon. In such situation, mostly, there is not a "complete" mathematical description of the phenomenon. The only thing, in mathematical terms, a modeler knows are parts of the system. To be able to see how these parts interact and evolve with each other, a modeler takes the help of a distributed simulation.

Figure 5.25 and figure 5.26 show the variables related to horizontal movement of the ball. Figure 5.25 shows how the horizontal velocity of the ball slowly decreases and reaches close to 0, which causes the ball to stop at a certain point in time. There is some amount of error in the results produced by hybrid simulation algorithm. In figure 5.26 it is shown how the ball covers the distance over time. As the initial value of the variable x was already given greater than 0, so the graph starts from 0.57. Slowly it covers the distance and as the horizontal velocity comes close to 0, it stops moving further and the graph on the figure 5.26 becomes horizontal.

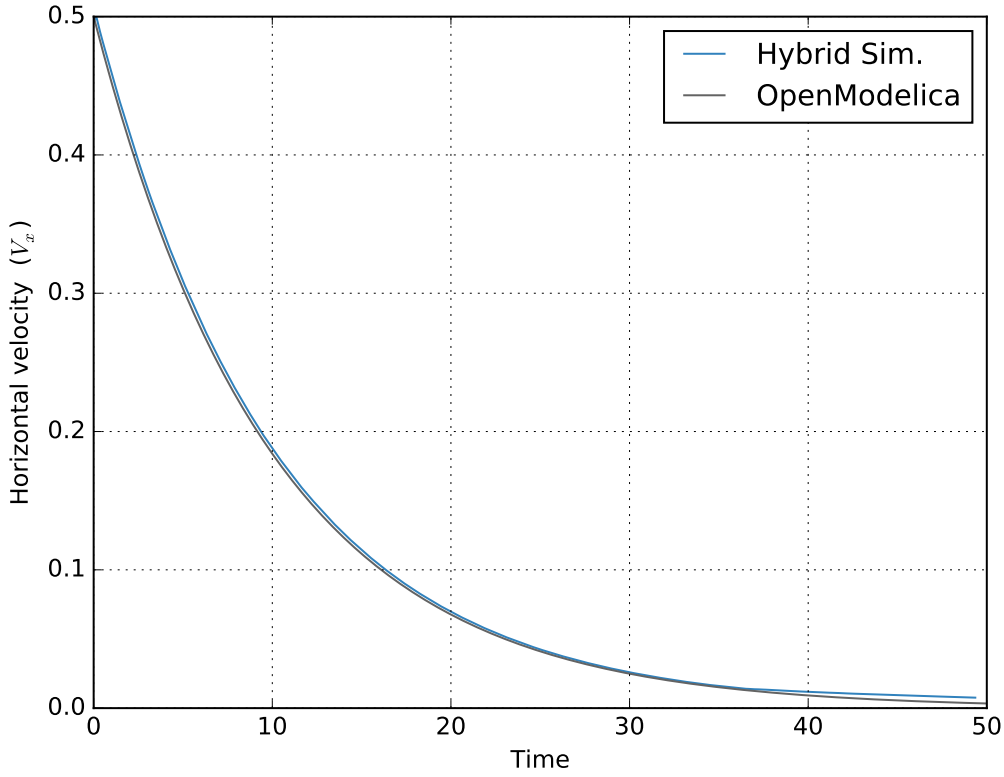


Figure 5.25: Comparative results produced by hybrid simulation algorithm and OpenModelica for variable v_x in equation 5.23. The variable v_x represents the horizontal velocity of the ball.

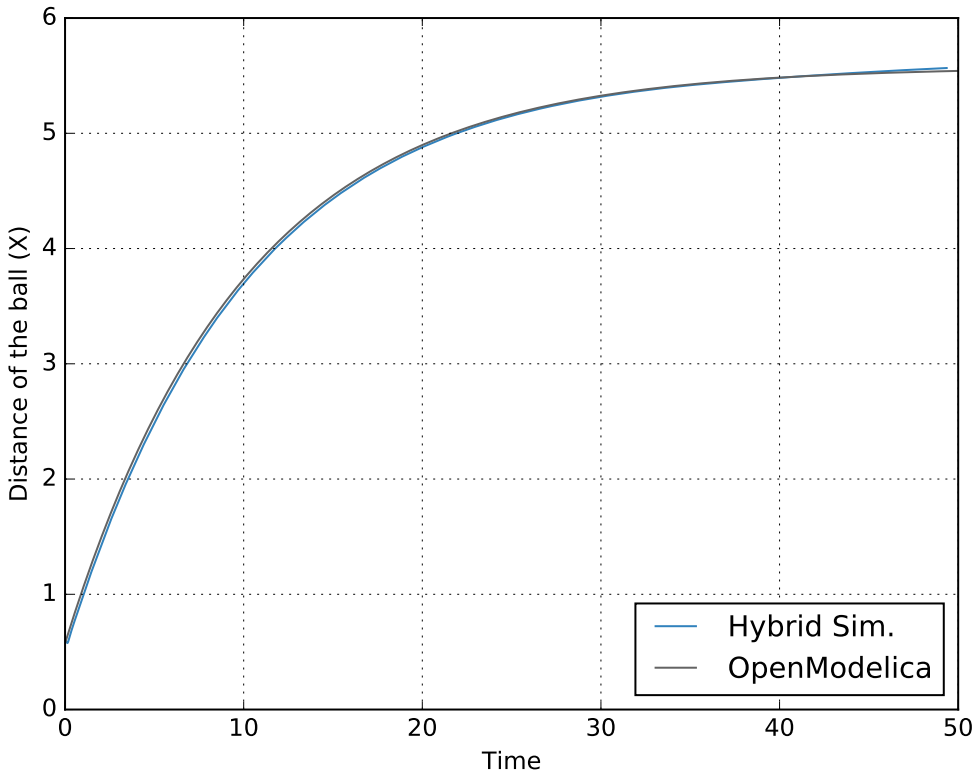


Figure 5.26: Comparative results produced by hybrid simulation algorithm and OpenModelica for variable x in equation 5.23. The variable x represents the distance traveled by the ball.

Focusing on the specific type of simulation methodology, the presented hybrid simulation algorithm, does not simplify the phenomenon of bouncing ball on stairs. It models parts of the system, discrete and continuous, separately, and lets them evolve with each other governed by an algorithm. The results presented here show the success of the algorithm, as they are so close to the results obtained by a monolithic simulator (OpenModelica), that it can be easily deduced that the algorithm is able to simulate an unknown physical phenomena successfully. The purpose of using the word "unknown" is to emphasize the fact that the "complete" model of physical (or cyber physical) phenomenon is not known, and the modeler, by providing mathematical description of the parts of the system, is relying on the algorithm to accurately simulate the parts of the system as a whole. This is also a reason of emphasizing the fact (section 5.2.4) that comparing a distributed simulation algorithm with a monolithic, or even a parallel simulation algorithm, is not justified. Both address their own specific realm of problems.

5.6 The SAHISim Framework

In order to enable a wider usage of the capabilities offered both by the HLA and the FMI, a framework is developed which enables a simulation engineer to orchestrate a distributed and interoperable simulation federation. The framework is named as Standardized Architecture for Hybrid Interoperability of Simulations (SAHISim). It is standardized because it enforces the simulations to conform to HLA or FMI. It enables hybrid simulation by implementing hybrid simulation algorithm presented in section 5.5,

secondly, the simulation standards—HLA and FMI— allow hybrid simulation, so the framework also supports hybrid simulations.

Current section should be considered as an extension to the previously presented section 3.3. In section 3.3 reader did not had a clear picture of the presented work, so it was a bit too early to go into the implementation details of the framework itself.

5.6.1 The FMU-Federate

An FMU-Federate can be considered as the worker of the simulation. Each FMU-Federate contains a sub-model of the complete system. During previous discussions, FMU-Federate has also been called as subsystem, which is what it represents in abstract terms. Figure 5.27 shows different components of an FMU-Federate. It shows that there are two main components of an FMU-Federate

- **HLA Messenger**: A portion which communicates with other FMU-Federates through the RTI, and the RTI itself for time synchronization. The main services provided by this portion are **time synchronization** and **data sharing**. Apart from this, it initializes the FMU-Federate and makes it part of the FMI-Federation. Simultaneous initialization of subcomponents is important in distributed simulation. The functionality is achieved by synchronization services provided by the HLA.
- **FMU Controller**: At the level of FMU-Federate or in abstract terms at the level of subcomponents, FMU controller is the part which directs the solver to solve an FMU according to the directions passed onto it. This part also know how inputs or outputs of the FMU are connected to the attributes of the FOM. The information is passed on to it in form of parameters. The executing program takes few parameters which include following information. One parameter tells FMU controller the federation to get registered to. Another parameter of type map tells which attributes of the FOM should be bound to which inputs. Another parameter tells which outputs should published to which FOM attributes. The publishing and subscription is performed automatically. In case of standalone algorithms FMU controller also includes the simulation algorithms like the Jacobi algorithm and the Gauss-Seidel algorithm. In case of master-slave algorithm it only follows the directions sent by the master.

The HLA is a complex standards requiring deep knowledge of it to be able to use it. The abstraction created in form of FMU-Federate simplifies its use. Modeler does not need to know the tiny details of HLA specifications. Using time synchronization services of HLA , sending and receiving timely updates itself is not a straightforward task, all these things are abstracted away in the from of FMU-Federate.

There are some intricacies involved in developing a generic solution using the FMI standard. FMU-Federate simplifies its usage too. The most important aspect is to use a solver to solve an FMU. There are many open source solvers available, so they can be used in conjunction, SUNDIALS is just one example. Although, there are few problems in using SUNDIALS. The most important one appears when an algorithm requires an FMU-Federate to rewind its solution to a specific time. It is so important that in FMI 2.0 the functionality of rewinding the FMU to a previous state is made part of the specification. It is found hard to solve get the functionality using SUNDIALS. One way could be that SUNDIALS could allow the solved solution to be discarded till a certain time. Alternatively, it could provide the facility to go back in time, but both things are not possible with SUNDIALS at the time of writing. Another functionality that can be very useful is to be able to set the past states of the solver. For example, suppose there is a system S to be simulated comprising of subsystems S_1 and S_2. When S_1 and S_2 are solved

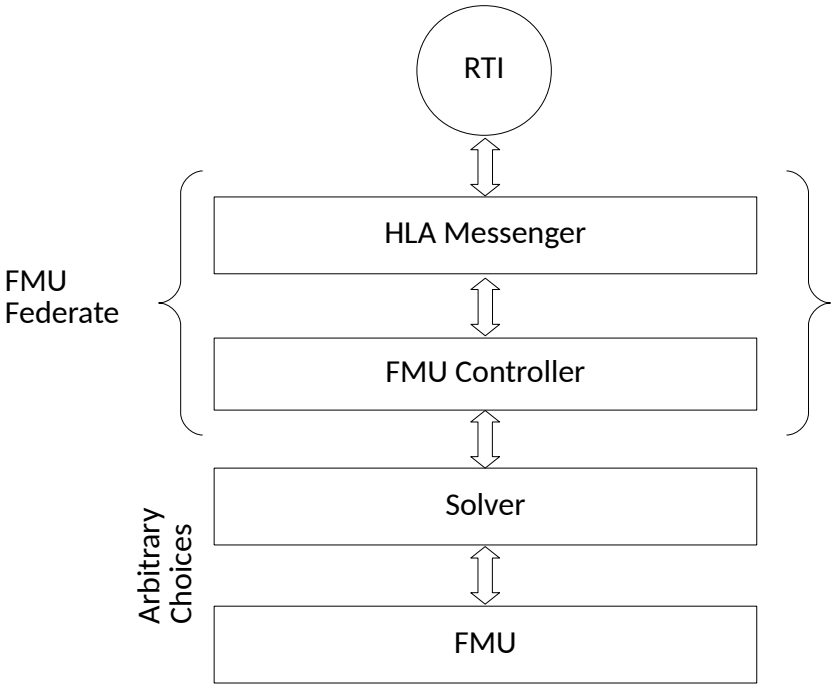


Figure 5.27: Components of an FMU-Federate.

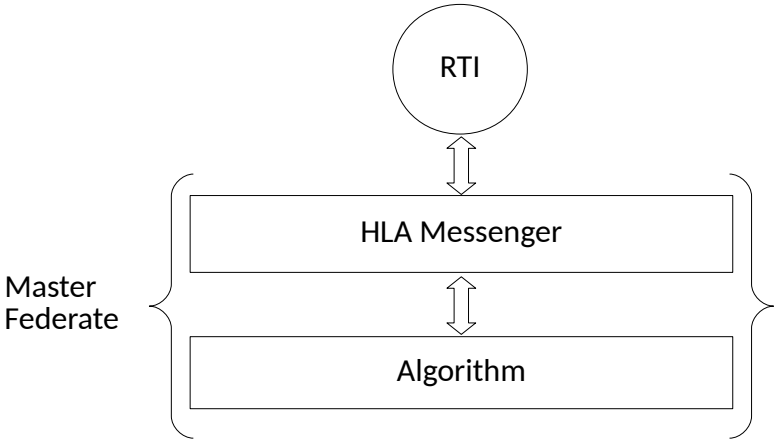


Figure 5.28: Components of a master federate.

for time step T_n separately using any solver, their results are not correct, until they go through some WR iterations. Once the results are achieved by convergence, after that S_1 or S_2 are to be solved for T_{n+1}. If the solver is using a multi step method, like Backward Difference Formula (BDF), to solve S_1 or S_2, then the solver may use a local vector of past states, which may not be the correct one. Instead the solver should use the vector of past states which were accepted as the legal states of the system after convergence. In such a situation one may like to alter the vector of past states manually and would like the solver to progress based on the provided states. A modification in SUNDIALS providing these functionalities can allow the modeler to use SUNDIALS in place of any other solver in SAHISim framework.

5.6.2 The Master Federate

The master federate is only used for the master-slave algorithms. It contains the so called master algorithm. It directs the slaves or other FMU-Federates in a way that the collective simulation produces correct results. Figure 5.28 shows a symbolic diagram of a master federate.

As shown in figure 5.28 there are two parts of a master federate. One portion communicates with the RTI and other FMU-Federates, using the RTI. The other portion contains the master algorithm. The HLA messenger performs the tasks of time synchronization and data sharing. The algorithm portion contains the strategy for simulation. In previously described algorithms the main focus was on this portion. Typically, the HLA messenger part is common for all master-slave algorithms described earlier, while the algorithm itself varies a great deal, as can be seen in previous sections.

Most important parts of the algorithm part are, the strategy for achieving convergence, and macro step size control. Algorithms described in previous sections are not the last word in distributed simulation algorithms. There are many other algorithms which can be implemented using the SAHISim framework. Specially FMI 2.0 has opened many new opportunities of developing efficient algorithms for distributed simulations. A researcher trying to come up with a new distributed simulation algorithm can use SAHISim framework for research and experimentation. A researcher wishing to do so has to do following main tasks.

- After a researcher has come up with the abstract strategy of distributed simulation, and if the strategy is a master-slave strategy. Then the master portion should encode the strategy in terms of execution loops and the commands sent in those loops. A typical simulation strategy must contain two loops, one for simulation and second for convergence.
- A corresponding slave state machine has to be developed after developing a master strategy. The SAHISim framework is also helpful in developing a slave state machine, as it provides the basic skeleton of a state slave machine.
- The researcher should also look for the situations where the synchronization of federates can go wrong. The easiest way is to see where the conditions described in section 5.2.3 are violated. To avoid such situations “synchronizing commands” can be introduced, as shown in previous sections.

6 Evaluation and Comparison

Previous chapters introduced six different distributed algorithms. In the present chapter their mutual comparison and comparison to other solutions in the field, are presented. While comparing the presented algorithms one thought comes to mind reflexively. Why and how the presented algorithms are better than the numerical algorithms presented earlier by researchers? It is emphasized once again that distributed algorithms for simulator coupling address their own unique set of problems. The problems they solve are just not solvable using monolithic numerical algorithms. Comparing the presented algorithms, and other algorithms for simulator coupling, with general purpose numerical methods of solving system of DAEs, is not appropriate. Sections 1.1.4, 5.2.4 and 5.5.3 have already covered the related differences of a distributed coupling algorithm and a monolithic or a parallel numerical algorithm.

In brief, one of the most important purpose a distributed algorithm serves is to allow the modeler to simulate a system which is not completely known as a system, rather only parts of the system are known, along with the interdependencies to other subsystems. Distributed algorithms allow the modeler to experiment with the complete system by evolving the subsystems in conjunction. In contrast, a monolithic or a parallel algorithm for solving system of DAEs must know the complete system, in some cases even the constraints applied on the system. Due to the complete knowledge of the system, a monolithic algorithm can transform the system of DAEs into a much simpler to solve system of ODEs, or even a single ODE. A distributed algorithm, on the other hand, cannot do this. If there is any simplification possible, it must be applied at a subsystem level, which may make a subsystem easier to solve, but does not make any difference to the whole system. This is another reason why the performance comparison between a monolithic and a distributed algorithm does not make much sense.

Before the presented work, the algorithms for solver coupling were only discussed in mathematics community on an abstract level. Here it is presented how these algorithms can take a form of a distributed algorithm. According to the knowledge of author it is the first attempt to explain the distributed nature of these algorithms. The SAHISim framework is also first of its kind. SAHISim framework allows all those simulation packages to share their models which conform to either HLA or FMI. There is no framework developed before which allows subsystems designed in so many different simulation packages to simulate together. Even if the standards are not kept, there is no example of a distributed simulation framework which embodies state of the art distributed numerical algorithms.

6.1 Comparison with FMI-Based Synchronous and Monolithic Algorithms

Distribution in nature offers an opportunity for parallelism, similarly, the distributed algorithms presented earlier allow subsystems to execute in parallel, but in a limited way. A simulation subsystem has to keep

up with other subsystems mainly to allow time synchronization. The presented algorithms are designed to allow the limited parallelism a simulation can offer. It may sound counter intuitive, but parallelism does not necessarily offer performance improvements. There are certain conditions that must be fulfilled to get performance improvements [Gus88]. Similarly, in a simulation environment distribution affects performance in a certain way, which is investigated in current section.

The time taken for a solver of a partitioned system can be divided into three parts, the integration part, the communication part and the constant processing time at master to arrange the results. Due to the matter under discussion, assume that the partitioned system S is simulated using FMUs as subsystems. For the sake of comparison, suppose that there is an algorithm which solves system S by loading FMUs in single process space. Because algorithm itself and FMUs reside in single process space, so no parallelism is possible. The algorithm necessarily ends up being a serial or synchronous algorithm. Now the time T_s taken by the synchronous algorithm for one macro time step is given by equation 6.1

$$T_s = S_c + \sum_{i=1}^{n} (I_i + C_{s_i}) \tag{6.1}$$

Where n is the number of FMUs, I_i is the cumulative integration time taken by FMU_i and C_{s_i} is the cumulative time spent on communication between master and FMU_i. As FMUs are not hosted in a separate process, and are linked statically or dynamically, the time C_{s_i} will be very small. S_c is the computation time taken by master to arrange the integration results and send commands. In case of monolithic execution the time S_c can be considered as 0.

Now examine the case of distributed simulation. The factors of time remain the same just the composition changes to equation 6.2. Here I_i is the integration time taken by FMU_i and C_{p_i} is the time spent on communication between master and FMU_i.

$$T_p = \max_{i=1}^{n} (I_i + C_{p_i}) + P_c \tag{6.2}$$

In place of the term S_c, here the term P_c is used. Which is computation time of a parallel algorithm for arranging the results and sending commands to the FMUs. The total number of macro time steps $\tilde{N}$ are considered equal in both the serial and distributed algorithms, so $\tilde{T}_s = \tilde{N}.T_s$ is the total time taken by the serial execution, and $\tilde{T}_p = \tilde{N}.T_p$ is the total time spent on the distributed execution. For simplicity, the difference between the terms S_c and P_c can be neglected.

It is clear from above equations that the performance gain due to distributed execution will only be substantial if the term $\max_{i=1}^{n}(C_{p_i})$ is minimum. In case of local linking the communication cost C_{s_i} is likely to be very small as compared to the communication cost C_{p_i} in case of distributed execution. In case of serial execution the sum of integration time of all FMUs $\sum_{i=1}^{n} I_i$ can be very large, especially if each represents a large subsystem. If each FMU_i is simulated by a separate simulation package, then the additional cost of context switching on a monolithic architecture could also make performance very slow. On the contrary, if the subsystems are very small and have to perform very few calculations for one integration step, then overhead of context switching will be negligible. Moreover, in such a scenario the complete cost T_s may remain far less than the communication cost $\max_{i=1}^{n}(C_{p_i})$ of the distributed execution. In other words, the distributed execution will produce better results if subsystems represent very large systems, who have to perform plenty of processing for each integration time step, or mathematically

$$\sum_{i=1}^{n}(I_i) > \max_{i=1}^{n}(I_i + C_{p_i}) \tag{6.3}$$

In equation 6.3 left hand side represents the a simulation where FMUs are locally linked to the master algorithm. The communication cost is negligible as it is just passing of parameters to the functions calls, so it is not included in the equation. The term on right hand side is not separable, since we are considering the maximum of sum of integration and communication time. Though, for the sake of bounded analysis, lets assume there is an FMU FMU_m which takes the maximum time I_{max} to solve one step. Suppose that the time taken for communication by FMU_m is $C_{p_{max}}$, and it is also an upper bound to other FMUs. Putting these terms in equation 6.3 results in equation 6.4

$$\sum_{i=1}^{n}(I_i) > I_{max} + C_{p_{max}} \tag{6.4}$$

From this equation, it is easy to see the fact that asymptotically right hand side of equation 6.4 is just a constant, while left hand side strictly keeps growing as the value of n grows. Taking average case analysis, and considering I_{avg} to be the average time taken by FMUs for integration of one macro time step, equation 6.4 becomes

$$\sum_{i=1}^{n}(I_{avg}) > I_{max} + C_{p_{max}} \implies n.I_{avg} > I_{max} + C_{p_{max}} \tag{6.5}$$

There must be some value n_0 of n for which equation 6.5 is true. In other words, if the number of FMUs is greater than a certain value n_0, then the distributed algorithm will always perform better than the synchronous one. The value n_0 in the special case under consideration is

$$n_0 = \left\lceil \frac{I_{max} + C_{p_{max}}}{I_{avg}} \right\rceil \tag{6.6}$$

Comparing it to equation 6.3, the value n_0 for a general case is

$$n_0 \approx \left\lceil \frac{\max_{i=1}^{n}(I_i + C_{p_i})}{I_{avg}} \right\rceil \tag{6.7}$$

Sign $\approx$ is used to show approximation, because in this equation few factors have been neglected, such as communication time of FMUs in case of serial algorithm, and the constant processing times of both distributed and serial algorithms. Nevertheless, the equation forms the basis to define the relationship between different factors of performance constraints in a distributed simulation. Equation 6.7 shows the benefit of using asynchronous and parallel distributed algorithm in a situation where more than one subsystems are part of the simulation, and each part must be simulated separately, due to different reasons.

6.2 Comparison with General Purpose Solvers

Equation 6.7 is not based on a comparison between a general purpose DAE analyzer and a distributed solver. Although, from equation 6.7 some interesting results can be deduced. Suppose there is a monolithic simulator which takes $\tilde{N}$ time steps to solve the system S. Now suppose a distributed algorithm also solves the system S in the same number of macro time steps i.e. $\tilde{N}$. If the average time taken by the monolithic system to solve one time step is I^m_{avg}, then a distributed algorithm will perform faster than the monolithic one if equation 6.8 holds.

$$\tilde{N}.I^m_{avg} > \sum_{j=1}^{\tilde{N}} \max_{i=1}^{n}(I^j_i + C^j_{p_i}) \tag{6.8}$$

It is reminded that $\tilde{N}$ is the number of steps and n is the number of FMUs. Equation 6.8 means that in order to make distributed algorithm perform faster, the sum of time spent by all FMUs for each macro step should be less than the total time taken by a monolithic algorithm, to solve system S. In real world problems there is no way to relate the terms I^m_{avg} and I^j_i with each other. It is far from realistic to assume that average time (I^m_{avg}) taken by a monolithic algorithm to solve one time step will always be greater than the time (I^j_i) taken to solve one subsystem S_i, for j^{th} macro time step. As monolithic solvers simplify the system before simulation and due to that the solution may become faster than its subsystem. Secondly, in real life it is hardly possible that a monolithic algorithm and a distributed one have the same number of steps ($\tilde{N}$), for a given problem. So in essence equation 6.8 proves the point once again that it is not appropriate to compare a monolithic solver with a distributed one.

6.3 Evaluation of Explicit Algorithms

In section 4.3.3 an example was presented which showed the benefit of using Gauss-Seidel method, here some more examples are given. The results are compared to OpenModelica [59]. The systems which are chosen for simulation are popular in simulation scientific community for their stiff behavior. The point of presenting such systems is that the analysis of the system and simulation is easy when it does not have too many variables, yet it is complex or stiff enough to establish the fact that if such systems are simulated well enough, then the smooth systems may be simulated much more easily.

The first model was described earlier in section 4.3.3 named "Lotka-Volterra equation" or "Predator-Prey equation". Another example is called as the "Van der Pol osialcillator". Its coupled differential equation form is described below

$$\begin{aligned} \dot{x} &= \mu(x - \frac{1}{3}x^3 - y) \\ \dot{y} &= \frac{1}{\mu}x \end{aligned} \tag{6.9}$$

The parameter μ is chosen to be $\mu = 1.53$. Third example is simple oscillator described by the equation 6.10

$$\begin{aligned} \dot{x} &= -y \\ \dot{y} &= x \end{aligned} \tag{6.10}$$

It was argued in section 4.3.3 that for "Predator-Prey equation" results of Jacobi method are not valid. On the contrary for "Van der Pol" oscillator, the figures 6.1, 6.2 and 6.3 suggest that the results for both algorithms are just the same. The error in the case of Jacobi method just like Gauss-Seidel method, is not increasing with time, and with a step size of 0.3 the solution does not deviate much.

For the system described in equation 6.10, with Jacobi method the solution is not stable and the error accumulates with the time (figure 6.6), while for Gauss-Seidel method results are similar to that of Open-Modelica results (figures 6.4 and 6.5). Moreover, figure 6.7 shows that by reducing the step size of the Jacobi method for unsuitable systems (Lotka-Volterra and Simple oscillator) the error also reduces. So it may be concluded that the systems are zero-stable for Jacobi method but are not stable otherwise. For details of zero stability and general stability conditions of a coupled system, reader is directed to [SL14b], [Arn10] and [BS10a].

It is a good point to mentioned that the in all algorithms mentioned above, especially in explicit methods, there is a special type of extrapolation used, which is called as constant extrapolation [Arn10]. Take the example of two coupled systems S_1 and S_2 discussed above, being solved by Jacobi method. After completing the integration for a communication step T_n, S_1 and S_2 move forward to solve the system at T_{n+1}. The input variables of S_1 and S_2 remain constant during the internal integration from T_n to T_{n+1}. An alternative method could be to use a linear extrapolation and change the input variables during internal integration. Extending the idea further, a polynomial extrapolation could also be used. Similarly in Gauss-Seidel method the "source" used the input values from the previous time step T_n and does not change them to calculate its outputs at T_{n+1}. Here too, instead of using constant input values linear or polynomial extrapolation could be used. In order to use polynomial extrapolation there should be some polynomial expression of growth of input variables. Otherwise, polynomial approximation methods can be used, to approximate a polynomial from past values. For linear extrapolation only past two values are needed to extrapolate the next one. However, how accurate is such an approximation and what are the results on accuracy and stability, is a completely separate topic of study. It was therefore emphasized in section 5.6 that there has been a need of a framework like SAHISim, using which a modeler can experiment with all these parameters with ease.

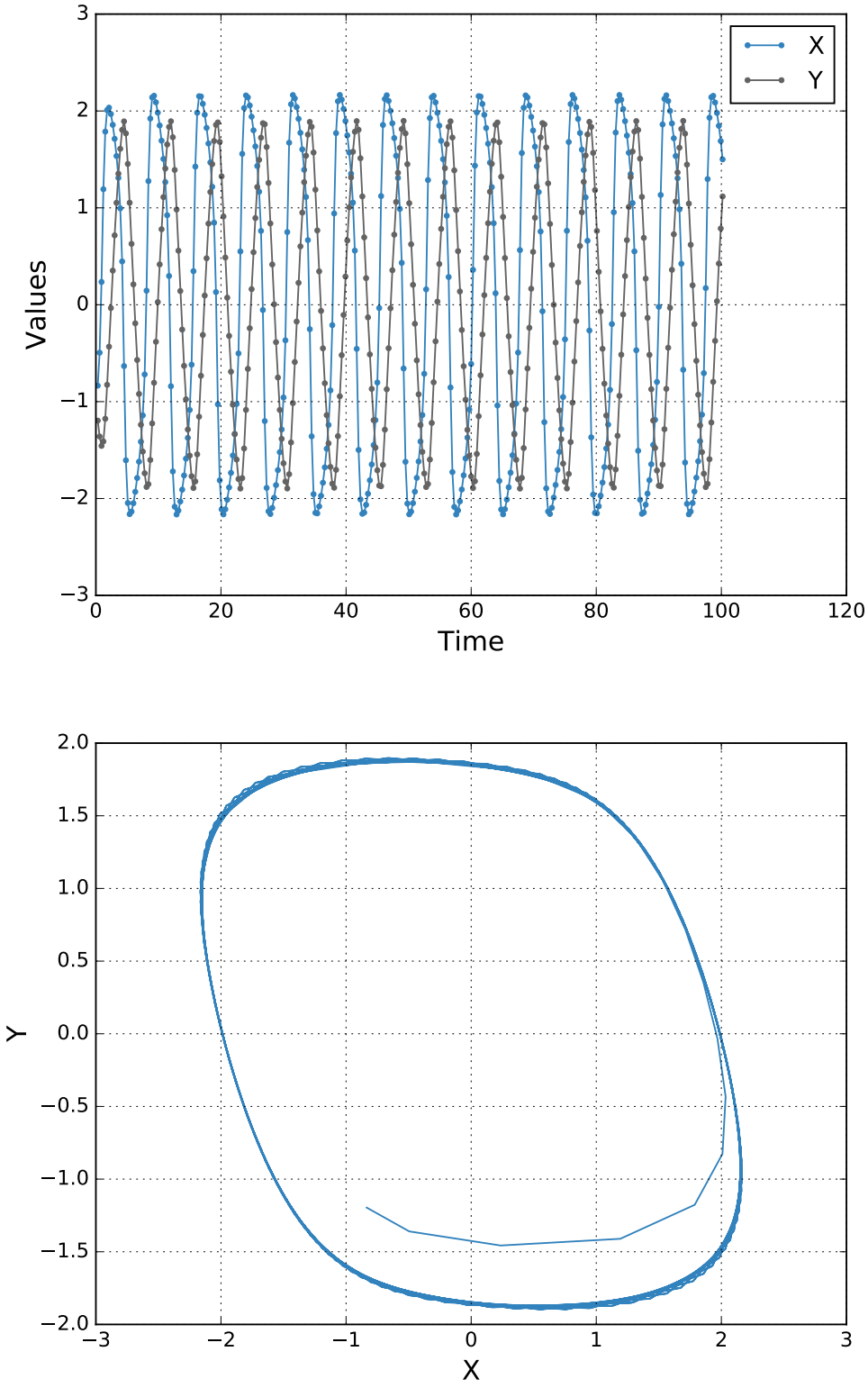


Figure 6.1: "Van der Pol" oscillator simulated using explicit Jacobi method. The "step size" is 0.3 here.

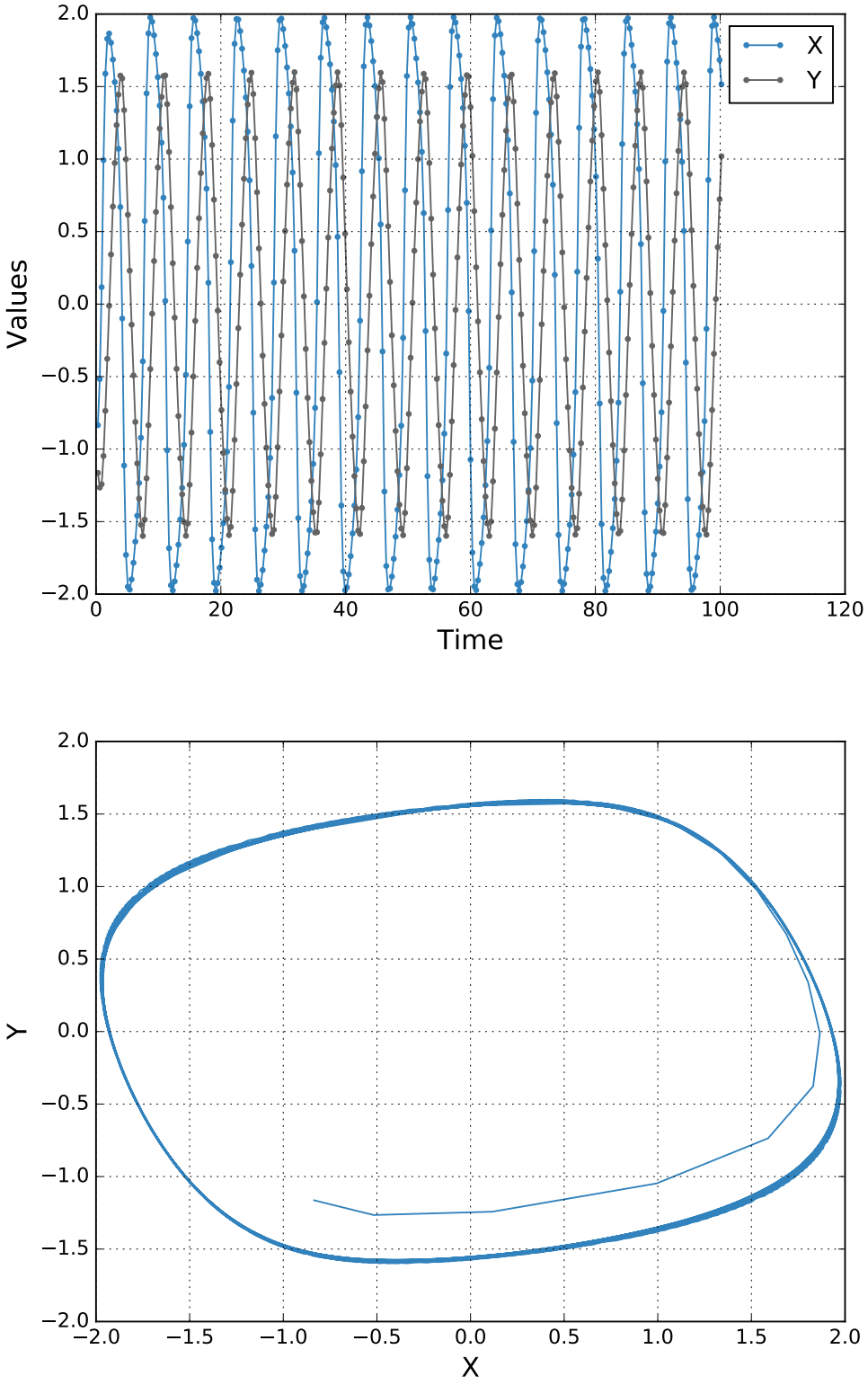


Figure 6.2: "Van der Pol" oscillator simulated using explicit Gauss-Seidel method. The "step size" is 0.3 here.

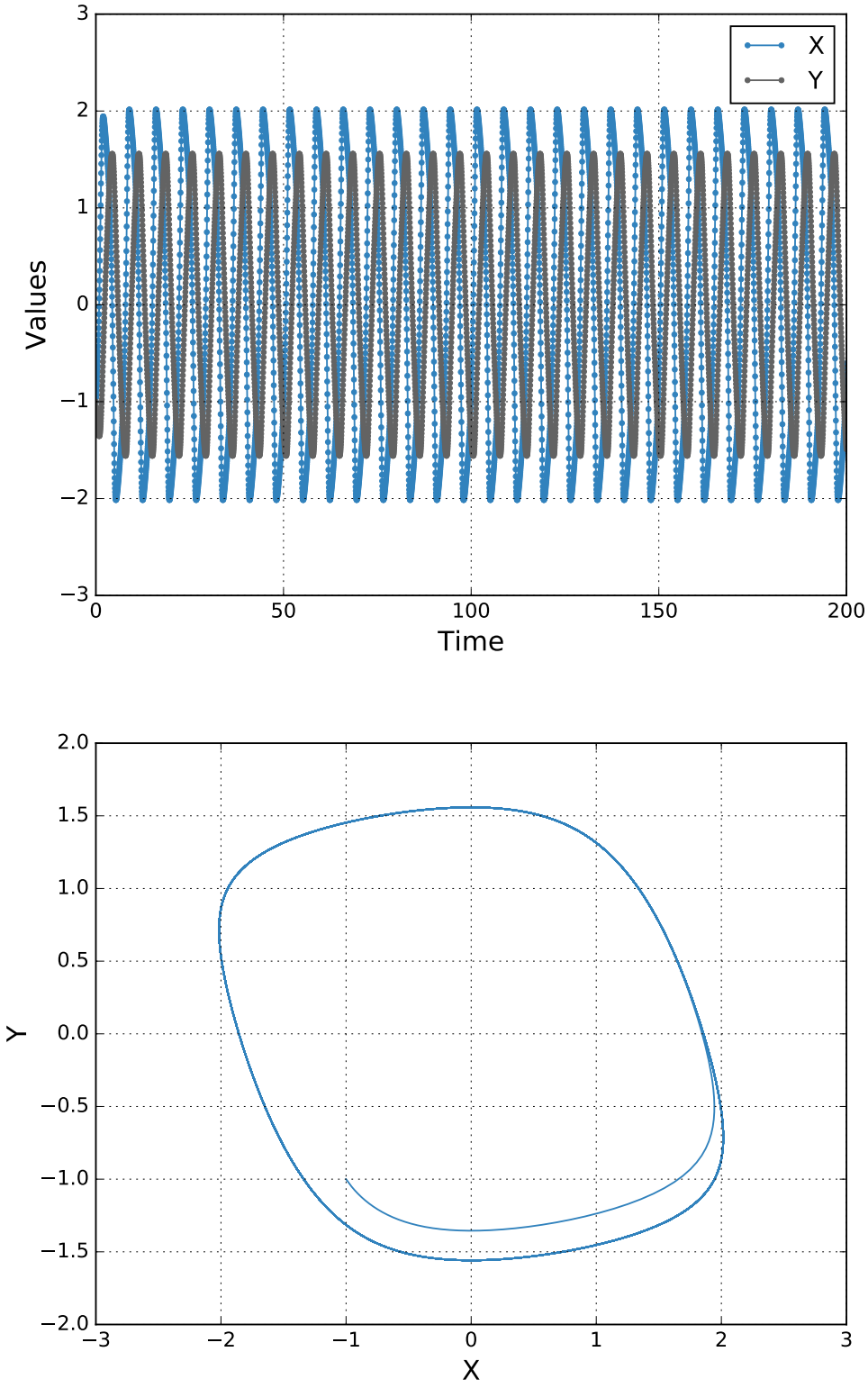


Figure 6.3: "Van der Pol" system simulated using OpenModelica. The "step size" is internally controlled by OpenModelica.

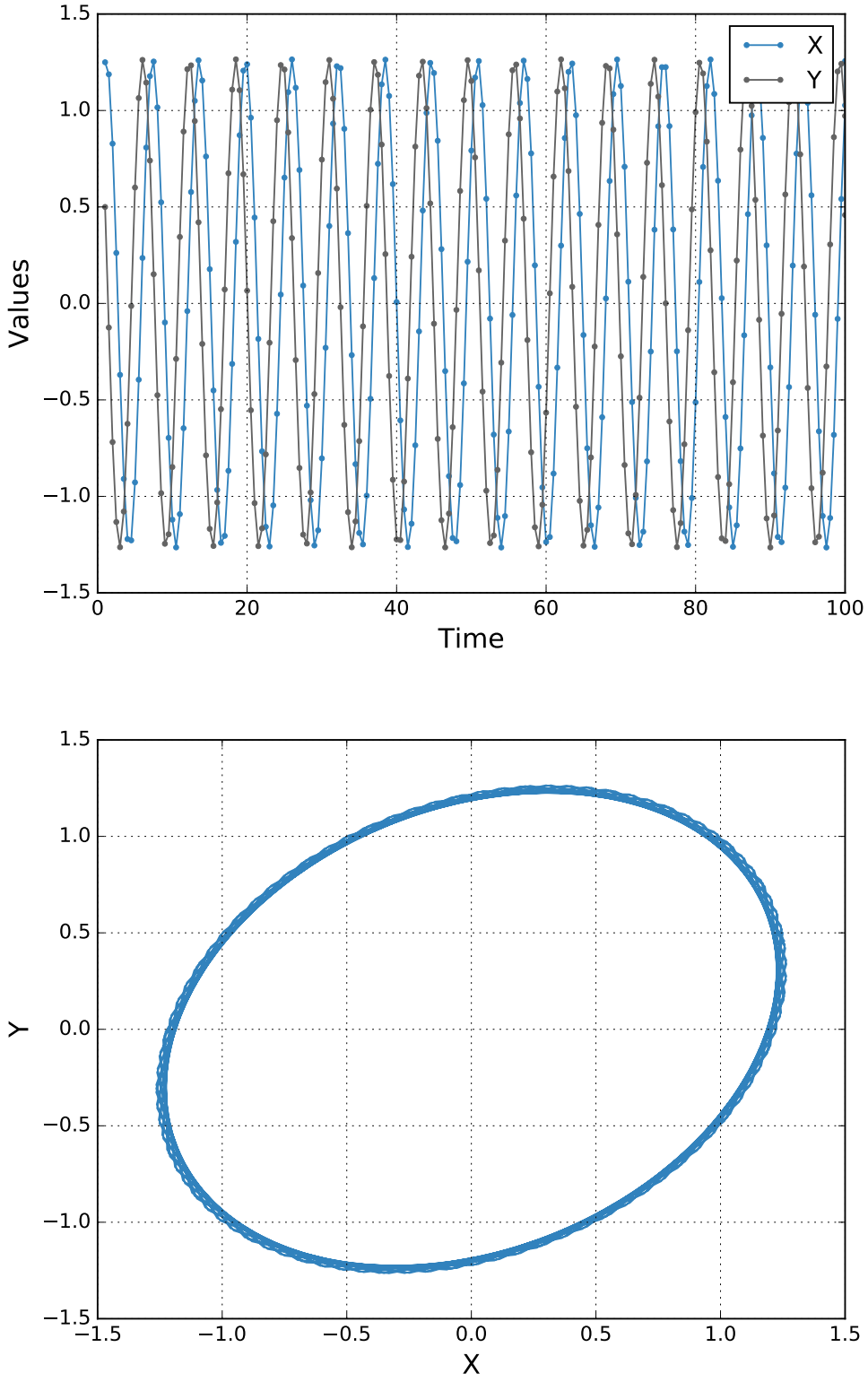


Figure 6.4: "Simple oscillator" simulated using explicit Gauss-Seidel method. The "step size" is 0.5 here.

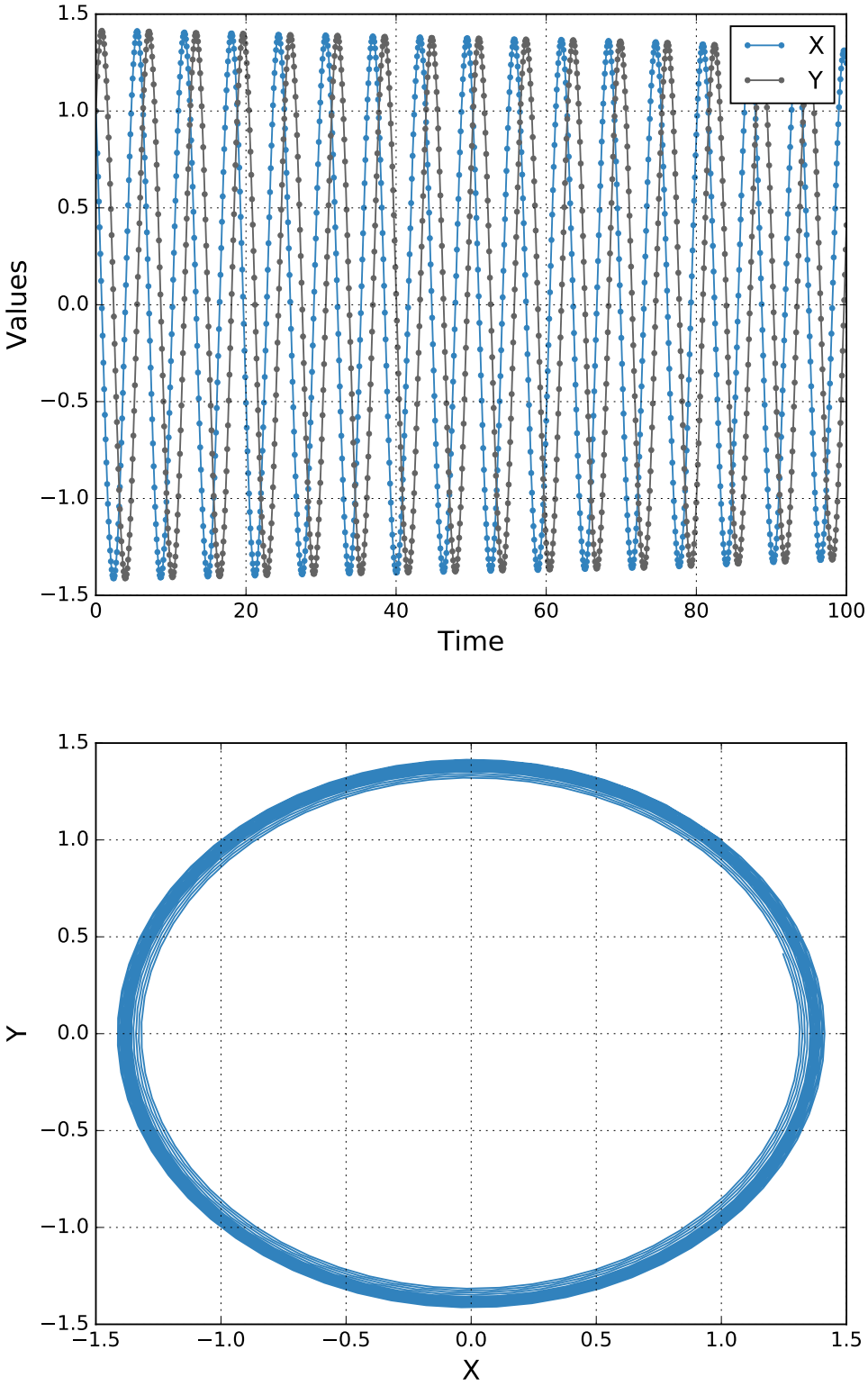


Figure 6.5: "Simple oscillator" simulated using OpenModelica. The "step size" is internally controlled by OpenModelica.

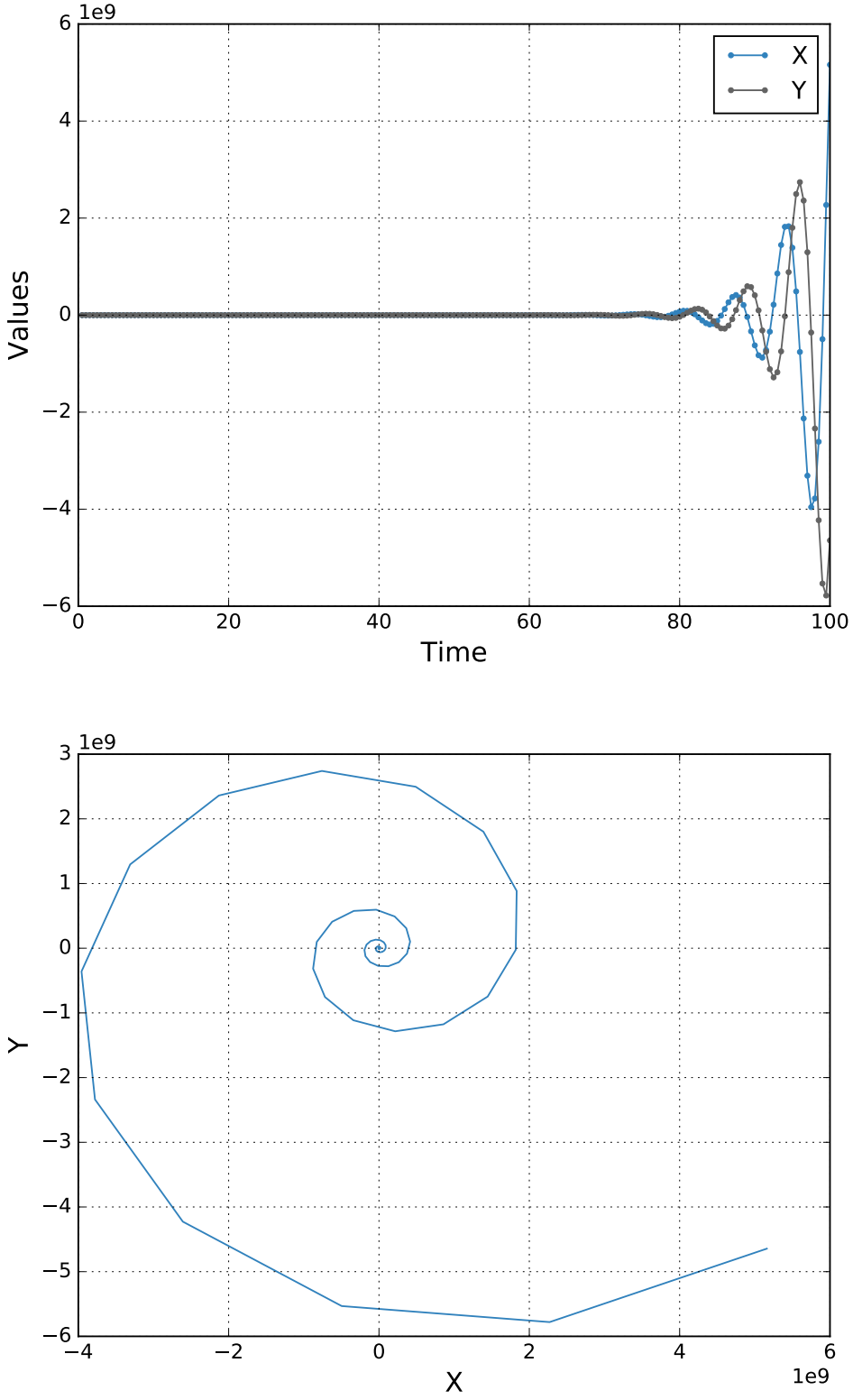
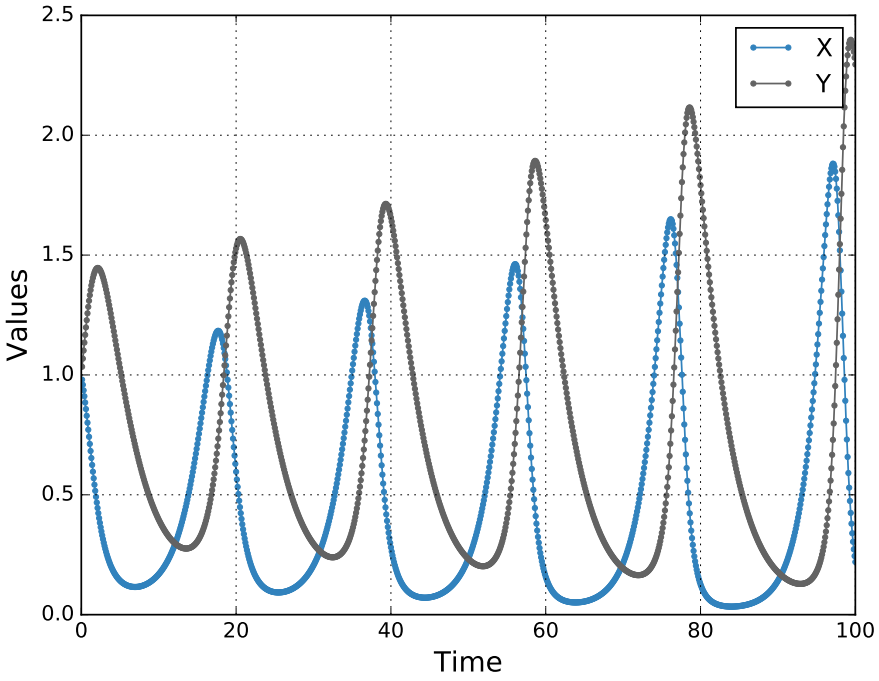
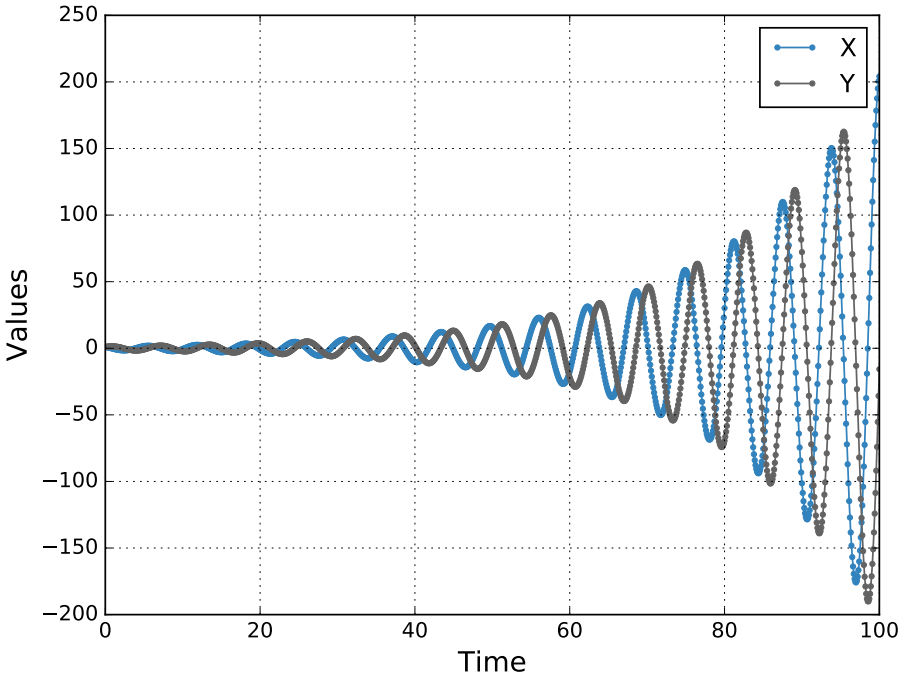


Figure 6.6: “Simple oscillator” simulated using explicit Jacobi method. The “step size” is 0.5 here.



(a) "Lotka-Volterra" system simulated using explicit Jacobi method, with reduced step size 0.1.



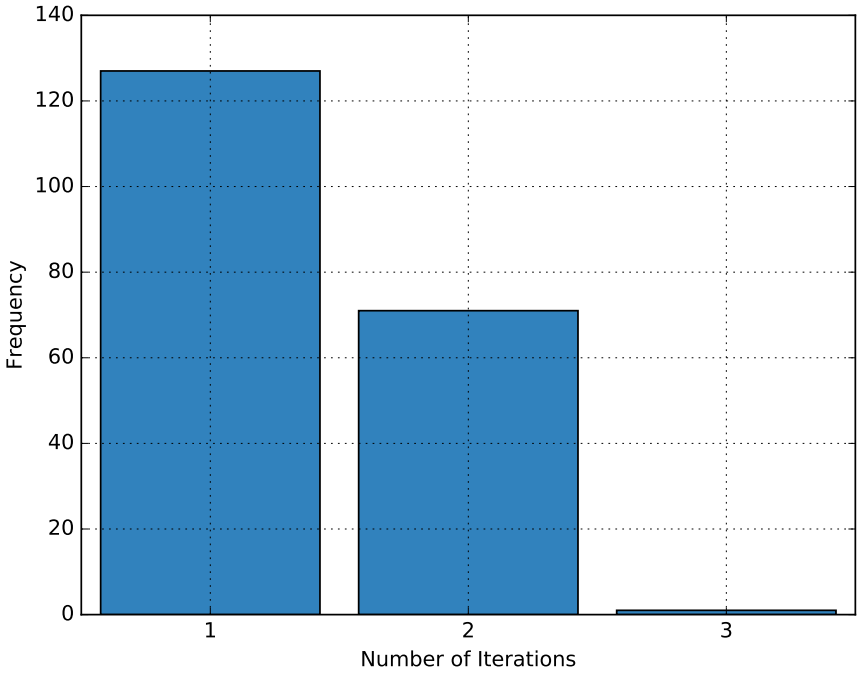
(b) "Simple oscillator" simulated using explicit Jacobi method, with reduced step size 0.1.

Figure 6.7: "Lotka-Volterra" and "Simple oscillator" simulated using explicit Jacobi method. The "step size" is reduced to 0.1 here, which also reduces the error.

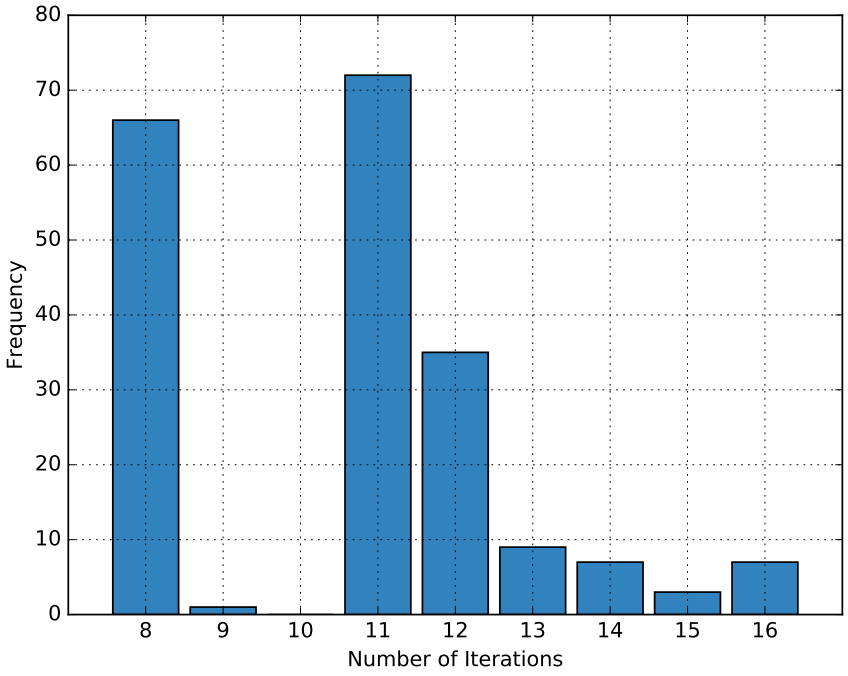
6.4 Evaluation of Implicit and Semi-Implicit Algorithms

Theoretical differences of implicit (waveform relaxation) algorithm and semi-implicit algorithm have been discussed in sections 5.2 and 5.3. Results presented in section 5.2.5 and 5.3.4 also show similar proximity from OpenModelica results. Theoretical aspects of message complexities have been discussed in sections 5.2.4 and 5.3.3. Despite their similarities there are some differences from practical point of view

- Semi-implicit algorithm in case of FMI 1.0, where there is no Jacobian information available, cannot produce any results, if there is no quasi-Newton method used. There is no way that a coupled system can converge without making any difference to inputs of its subsystems. Semi-implicit algorithm calculates residual with the help of a system Jacobian or any other quasi-Newton method, and tries to minimize the residual by changing the inputs to subsystems. WR algorithm, on the other hand, does not need any external algorithm or routine for adjusting inputs to subsystems. WR algorithm relies on the structure of the problem to correct the error by iterative calculations. In practical terms WR algorithm is more easily usable because it does not need any additional implementation for convergence of the system.
- In a situation where the macro time step is much larger than the internal integration step of subsystems, the number of function evaluations in case of semi-implicit algorithm may be less than the number of function evaluations done by WR algorithm. The reason is semi-implicit algorithm only adjusts the result using only the last internal integration time step. While WR algorithm integrates through all previous internal integration time steps, starting from last macro time step. Although the conclusion is not straightforward because number of function evaluations also depend on the number of fixed point iterations. If fixed point iterations of semi-implicit algorithm are more than WR algorithm then the advantage mentioned above can be lost.
- This has been noted by researchers that in absence of Jacobian information the convergence to the result become much slower [CK06]. Because WR algorithm relies on the structure of the problem, so its convergence rate is observed to be better. Figures 6.8, 6.9 and 6.10 show its empirical evidence.

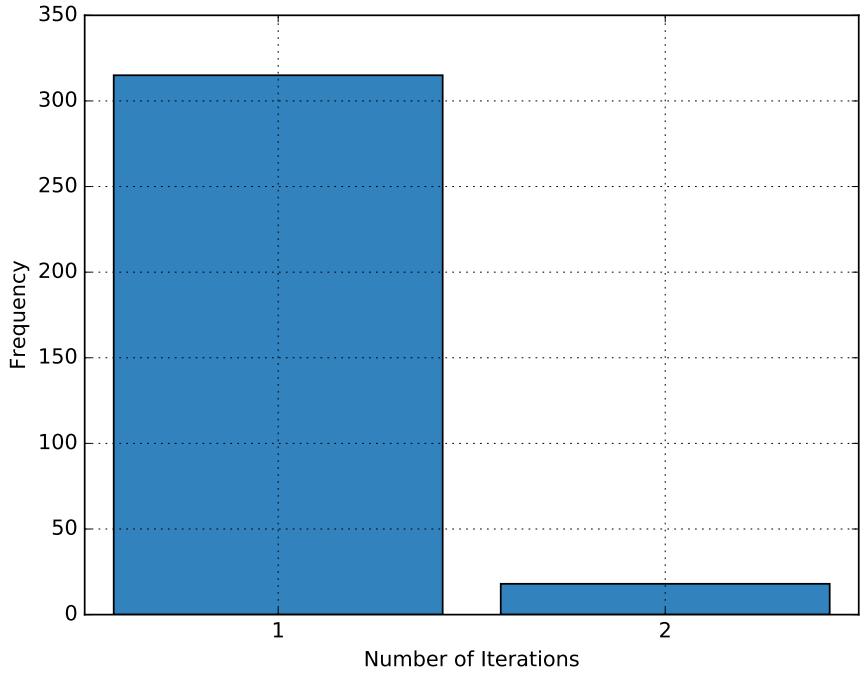


(a) Fixed point iterations in WR algorithm.

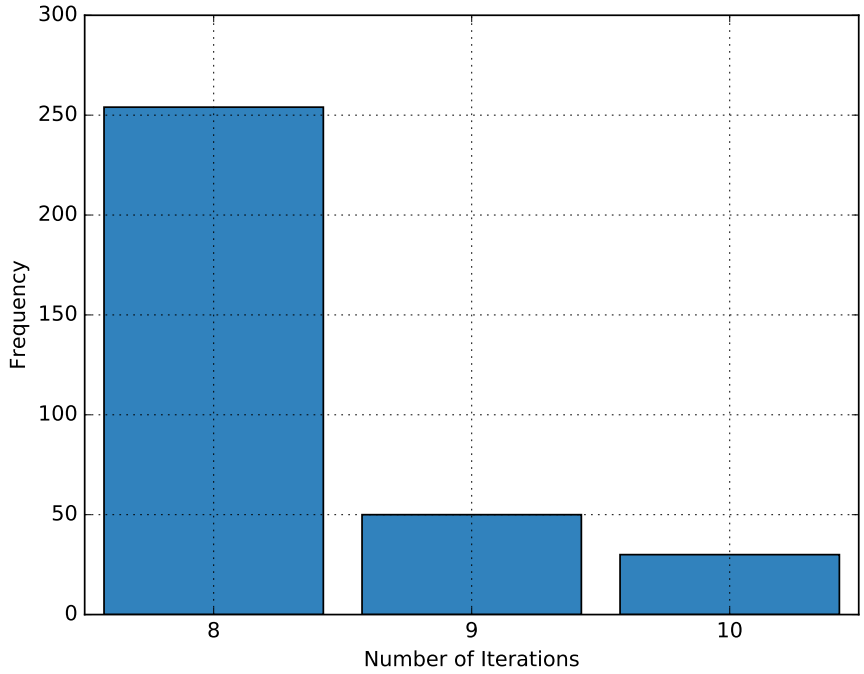


(b) Fixed point iterations in semi-implicit algorithm

Figure 6.8: Histogram of fixed point iterations while solving "Lotka-Volterra" system.

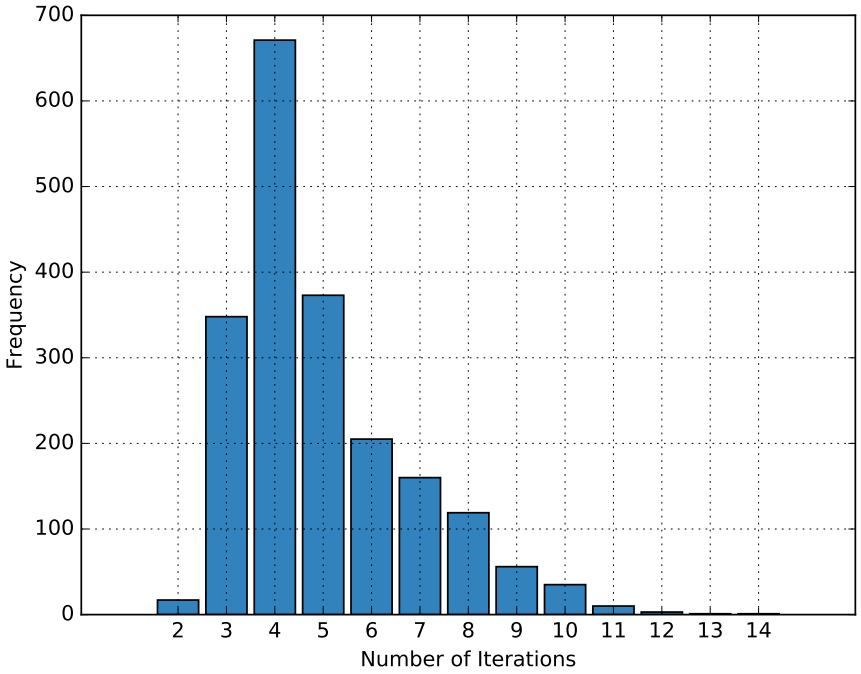


(a) Fixed point iterations in WR algorithm.

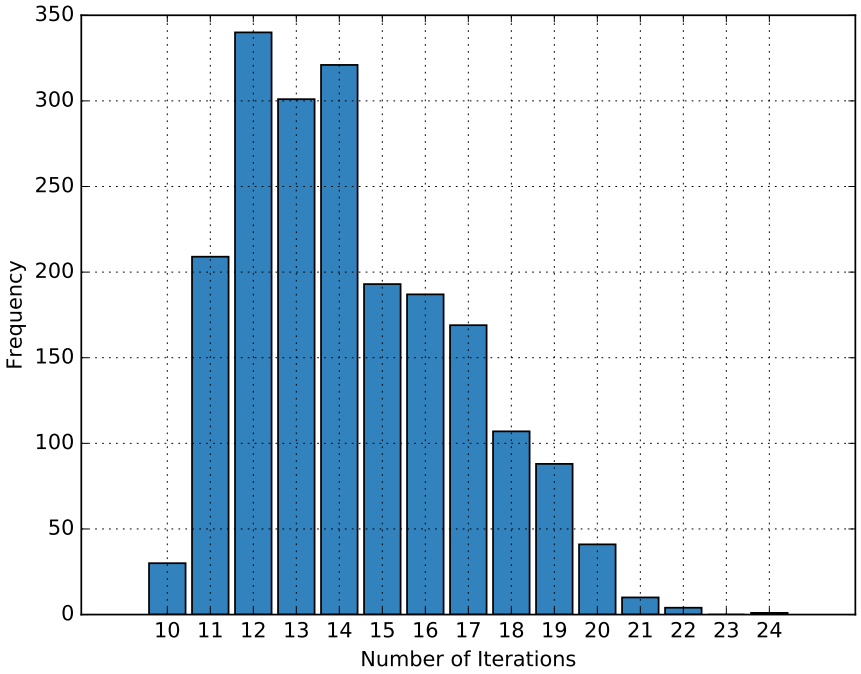


(b) Fixed point iterations in semi-implicit algorithm

Figure 6.9: Histogram of fixed point iterations while solving "Van der Pol" system.



(a) Fixed point iterations in WR algorithm.



(b) Fixed point iterations in semi-implicit algorithm

Figure 6.10: Histogram of fixed point iterations while solving “Lorenz attactor”.

7 Conclusion

Focus of the presented thesis is co-simulation in distributed environment. To some the problem looks pretty simple, because many practitioners think about co-simulation in terms of time synchronization and data exchange only. Section 3.1 addresses the misconception and presents major challenges in dealing with co-simulation of real world phenomena. Some scientists have already developed solutions targeted to limited domains (see chapter 2), yet they have only focused on domain specific solutions without theoretical elaboration of the underlying simulation algorithms. Mathematicians have been working to develop algorithms for solver coupling, but how they can be implemented in distributed topology is first explained in the presented work.

Developed distributed algorithms heavily make use of the High Level Architecture (HLA), yet their specification is generic and does not bound the future developer to restrict himself to any standards at all. Components complying to Functional Mock up Interface (FMI) can be viewed as an embodiment of abstract subunits of the complete mathematical model. A future developer may not like to adhere to FMI, and is free to use his own abstractions to simulation subunits. In this regard the presented distributed algorithms are a formal specification of the numerical algorithms for co-simulation. Nevertheless, using HLA and FMI for the implementation greatly reduces the effort. Timing, data sharing and other services provided by the HLA allow to work in a more flexible distributed computing model. Similarly using FMI truly makes the solution practical and beneficial for the practitioners, as there are quite a few simulation packages adhering to, or trying to adhere to the FMI specifications.

As soon as FMI 2.0 specification is available in open source community, even more scientists will try to come up with even better algorithms for distributed co-simulation. Lack of FMI 2.0 compliant simulation packages restricted the author to delve into more advanced solver coupling algorithms already presented in mathematics community. Semi-implicit algorithm, without having access to Jacobian information, is less effective. FMI 2.0 can be very useful in this regard. Once there is an open source simulation package which could generate FM1 2.0 specific FMUs, a great deal of development will be possible in semi-implicit algorithm.

For continuous research and development a framework for distributed simulation is presented in section 5.6. The framework is named as Standardized Architecture for Interoperability of Hybrid Simulations (SAHISim). Already presented six algorithms have been implemented using the SAHISim framework. In future, more advanced algorithms can be developed and tested using SAHISim framework. The SAHISim framework is built upon the foundations of HLA and FMI. In author's opinions it is very beneficial because even if a scientist wants to develop some specific algorithms for distributed co-simulation, he can use SAHISim framework to test and verify his idea with ease. So a rapid development using SAHISim framework of a distributed co-simulation algorithm can serve as a proof of concept.

Test examples for the developed algorithms are not picked from any specific domain, due to mainly two reasons. One, picking a complex example from a specific domain makes it difficult for the reader to focus on the real problem, that is, efficiency of the distributed co-simulation algorithm. Complex examples from specific domains may serve as a news of conquering a difficult problem, and may offer better understanding to the experts of that domain, but they serve less for general audience of modeling and simulation community. So such test examples have been chosen which have been used in various texts of modeling and simulation community. All those problems are well renowned for their certain types of challenges in modeling and simulation community. Results produced by a developed algorithm are compared with the results produced by OpenModelica, using results of OpenModelica as a reference solution. Difference from the reference result are presented in the study. This leads us to the second main reason of not using complex examples. A complex example takes a lot more effort to be verified with a reference implementation. An example with hundred different state variables with non-stiff nature may take many weeks to implement in OpenModelica, but will not virtually offer any challenge to the distributed co-simulation algorithm. On the other hand, a stiff system such as Lorenz attractor (equation 5.9) is pretty easy to implement in OpenModelica, but poses major challenges to a distributed co-simulation algorithm. Its famous stiff behavior also saves the time and effort to learn about the nature of a new complex model.

Similarly a hybrid system like a bouncing ball on stairs (section 5.5.3) can be easily implemented in OpenModelica, but simulating it without knowing the generation model of discrete events, makes it much harder. As OpenModelica knows the system of equations, so it also knows the generation model of discrete events. Tackling a discrete event with a known model of generation is not too difficult. Though, in real world scenarios it is less likely to have a complete knowledge of how and when a discrete event is going to occur. The hybrid simulation algorithm presented in section 5.5 tackles discrete events without any knowledge of their generation model. It reacts to a discrete event spontaneously and keeps solving the system accurately.

Just like the traditional monolithic algorithms, distributed algorithms can be used according to the problem at hand. Explicit methods discussed in section 4 are used where the coupling between subsystems is loose. Waveform relaxation algorithm and semi-implicit algorithm are more stable because they iteratively converge on the right solution. However, the stability of the solution to some extent also depends on the individual solver used at the subsystem level. SUNDIALS is very popular for its efficiency and stability, and looks to be a promising choice, but there are few issues with it. SUNDIALS solver was not written to become part of co-simulation solution, so it has its traditional way of solving systems. If there are few changes made in the SUNDIALS solver then its can become very useful asset in co-simulation solutions. The modifications in SUNDIALS that may make it useful for co-simulation are listed in section 5.6.1.

In section 6.2 a detailed argument is given why a distributed co-simulation algorithm should not be compared with a monolithic algorithm, in time and efficiency. Both serve their own domains of problems. Solver coupling algorithms in mathematical terms are a relatively new concept. They are not as advanced as traditional numerical algorithms. With continuous research, it is possible that one day for some specific types of problems, coupled algorithms may outperform traditional algorithms in all respects. With the current state of the research, it is not possible to say so.

Appendices

A Results of Standalone Algorithms

A.1 Explicit Jacobi Method

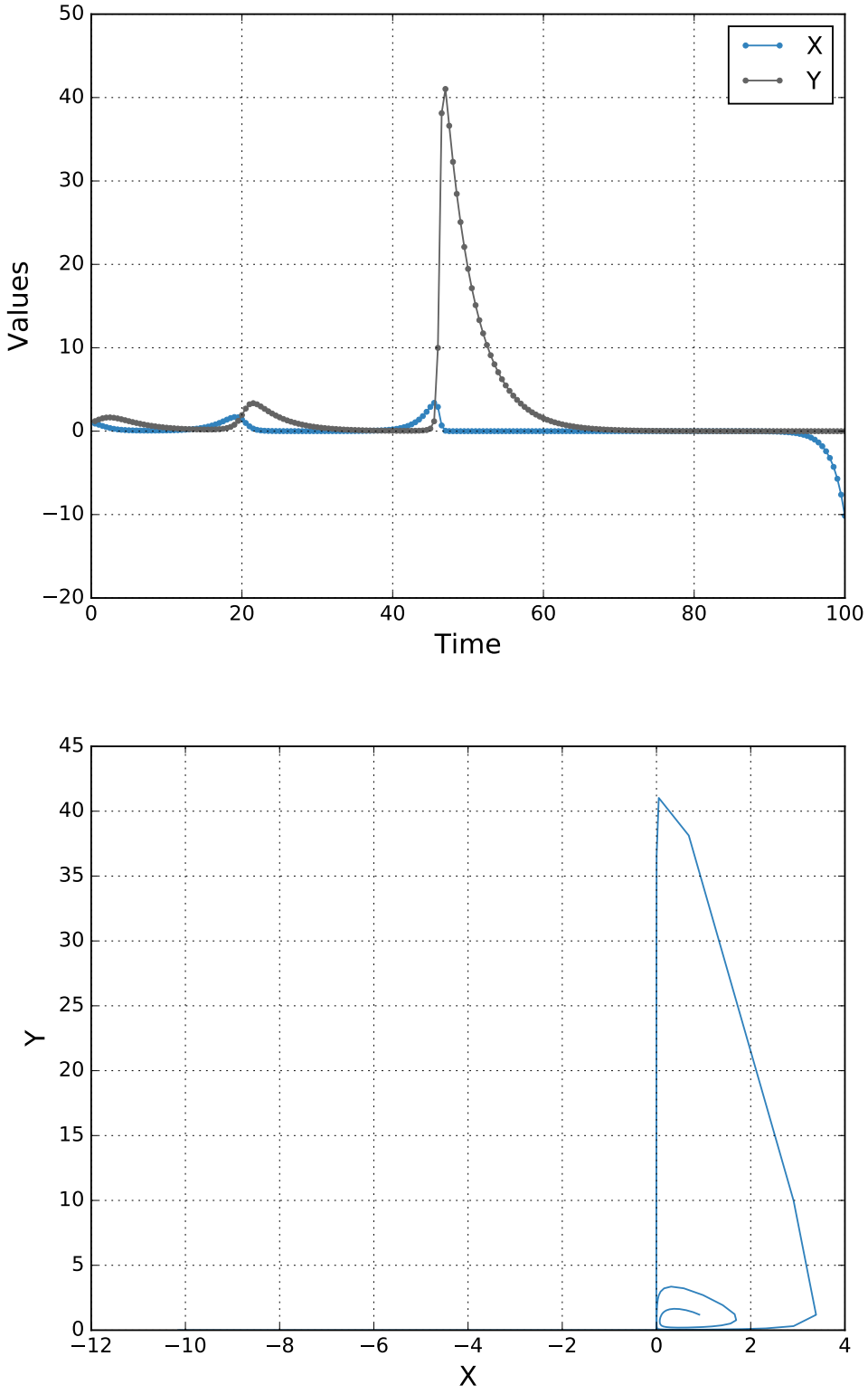


Figure A.1: "Lotka-Volterra " system simulated using explicit Jacobi method. The "step size" is 0.5 here.

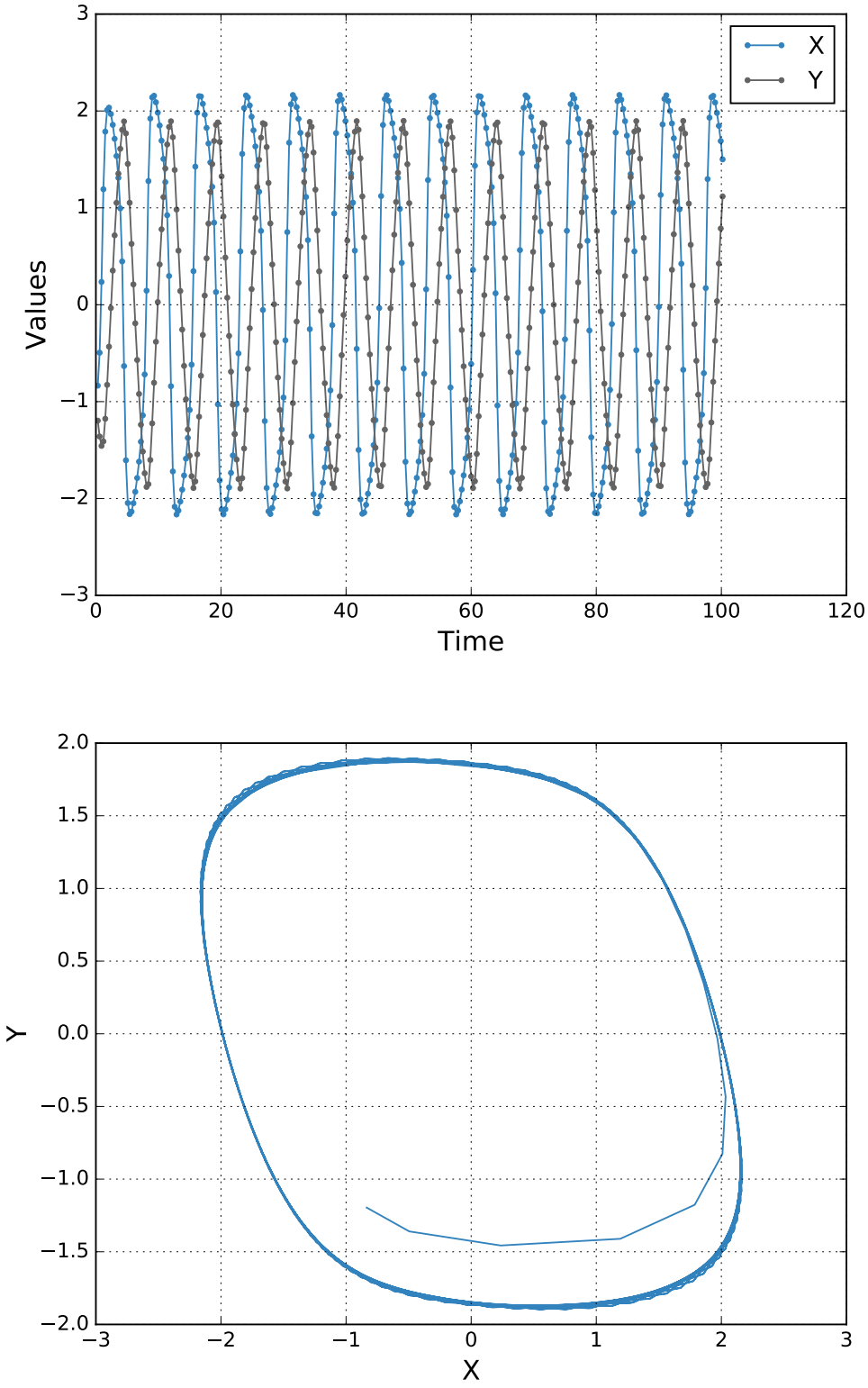


Figure A.2: "Van der Pol" oscillator simulated using explicit Jacobi method. The "step size" is 0.3 here.

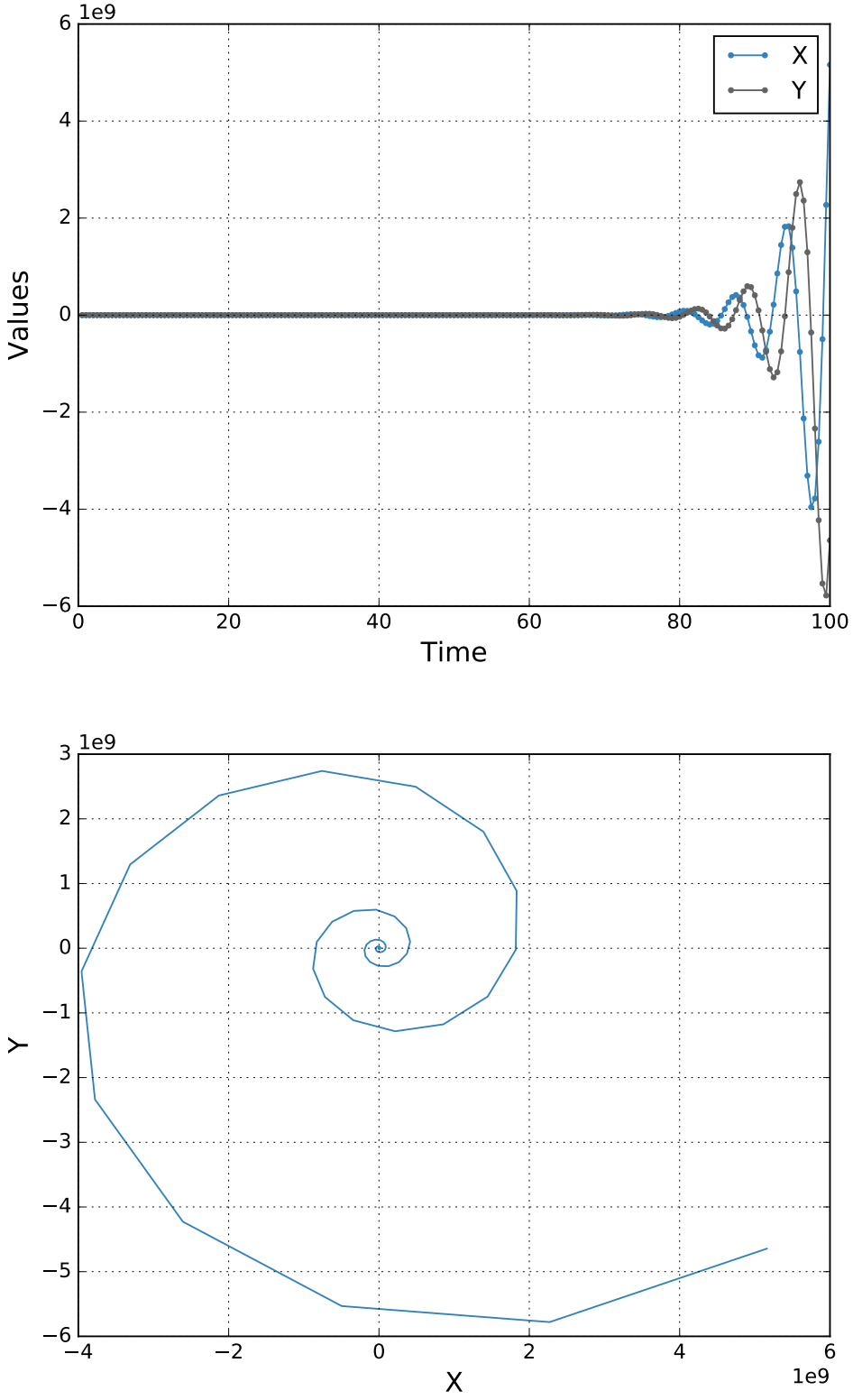
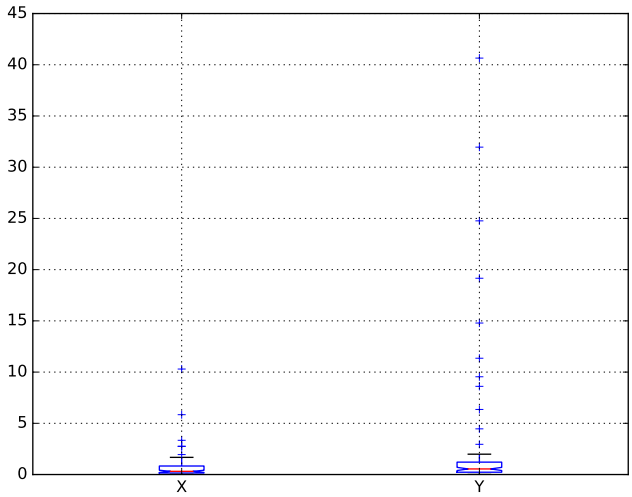
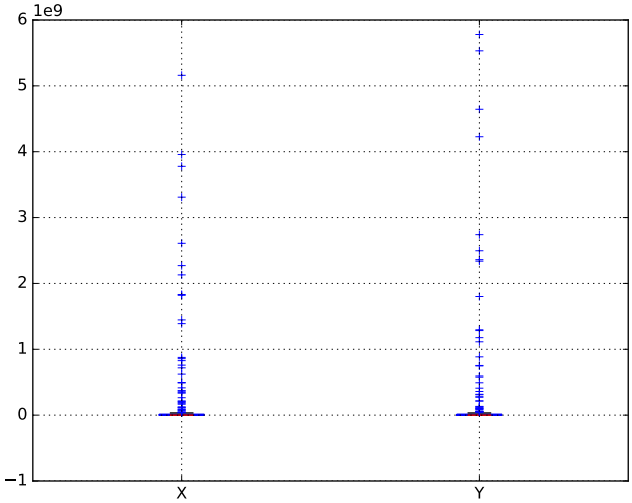


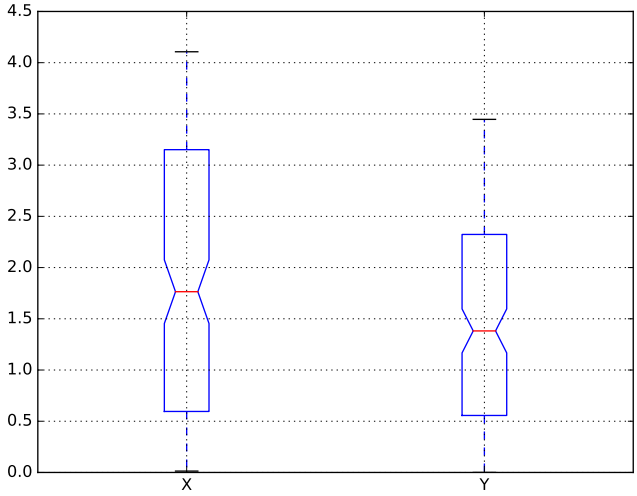
Figure A.3: "Simple oscillator" simulated using explicit Jacobi method. The "step size" is 0.5 here.



(a) Cumulative difference of results between OpenModelica and Jacobi method for “Lotka Volterra” system.



(b) Cumulative difference of results between OpenModelica and Jacobi method for “simple oscillator”.



(c) Cumulative difference of results between OpenModelica and Jacobi method for “Van der Pol” oscillator.

Figure A.4: Difference of results between Jacobi method and OpenModelica. The whisker bars show how the values produced at each time step using Jacobi method differ from the values of OpenModelica.

A.2 Explicit Gauss-Seidel Method

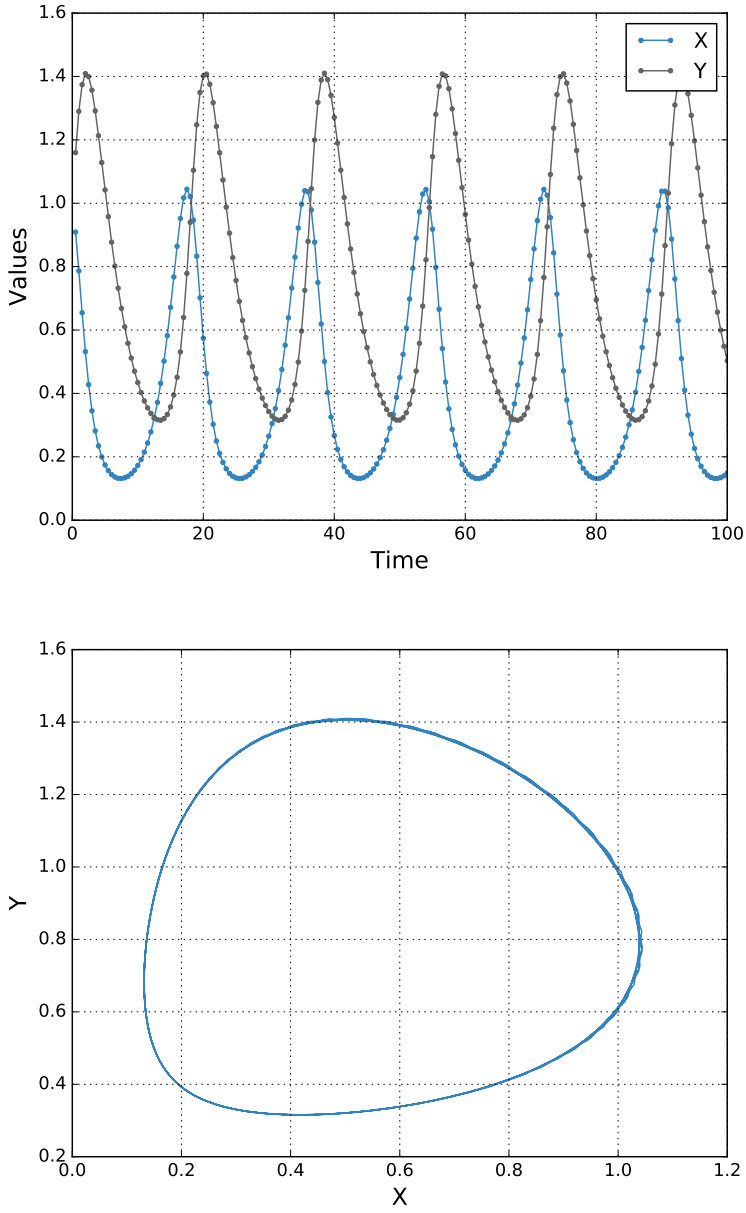


Figure A.5: "Lotka-Volterra " system simulated using explicit Gauss-Seidel method. The "step size" is 0.5 here.

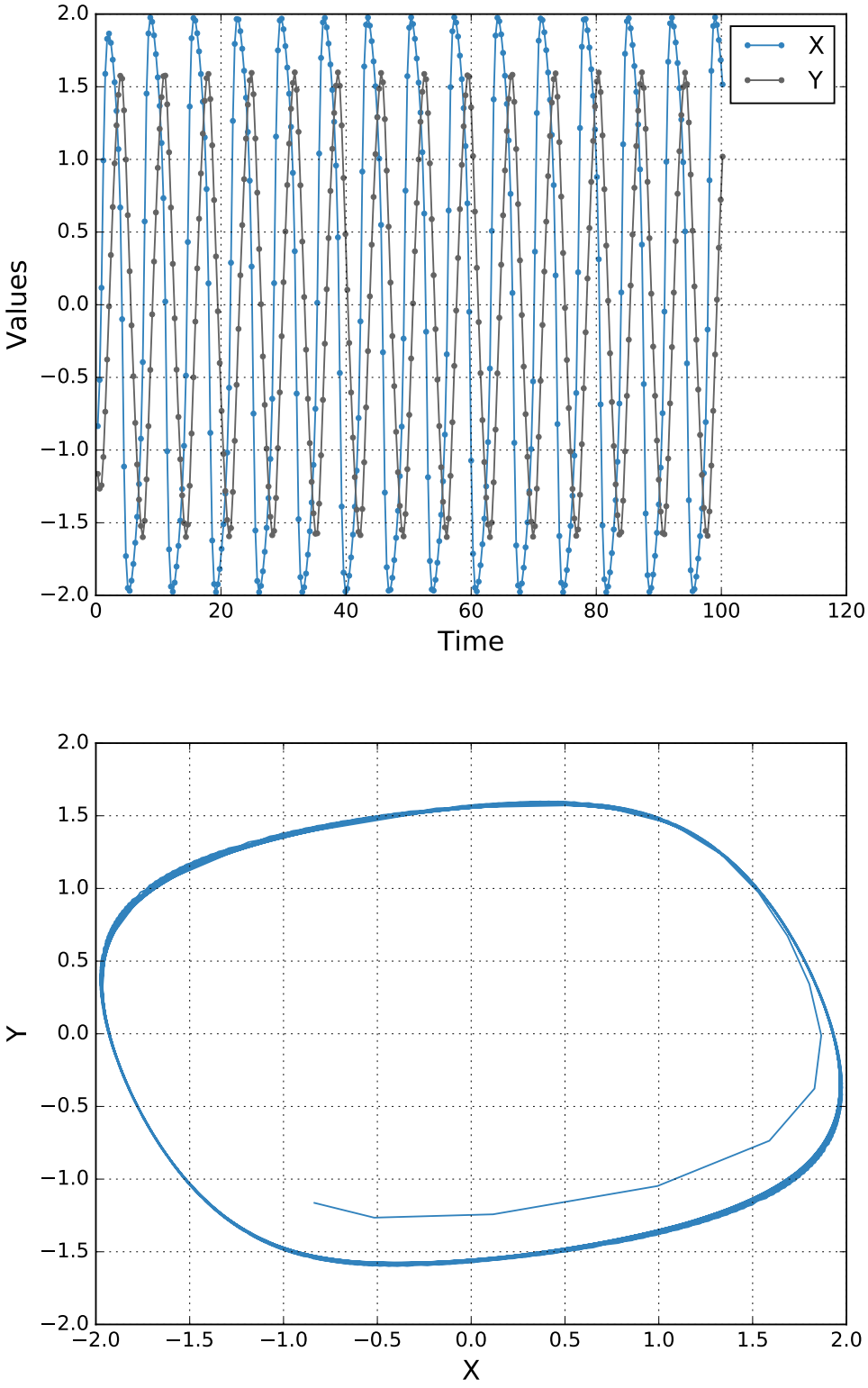


Figure A.6: "Van der Pol" oscillator simulated using explicit Gauss-Seidel method. The "step size" is 0.3 here.

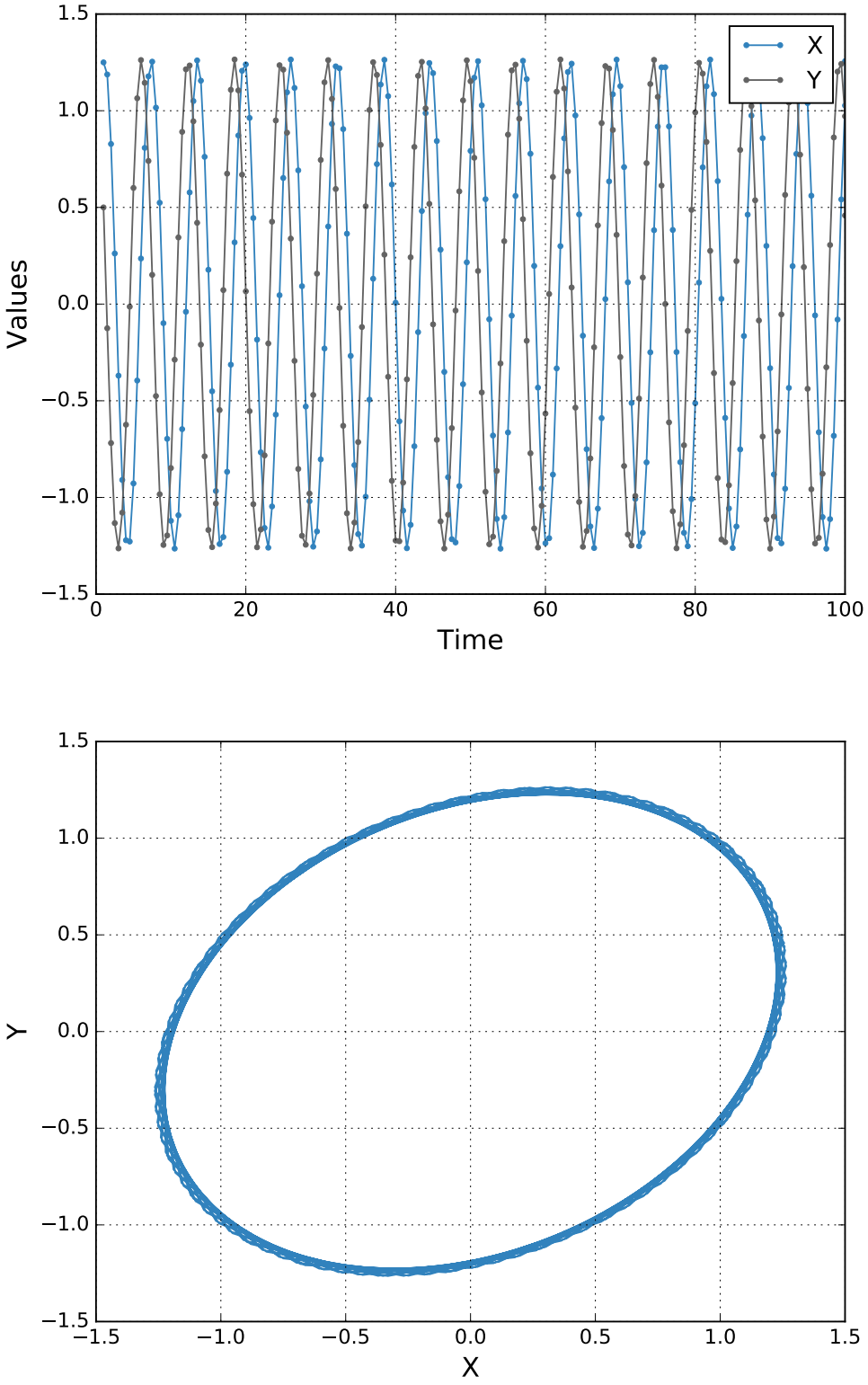
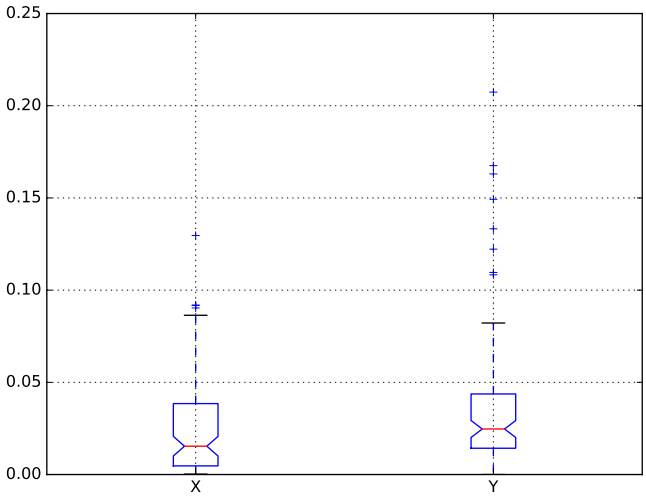
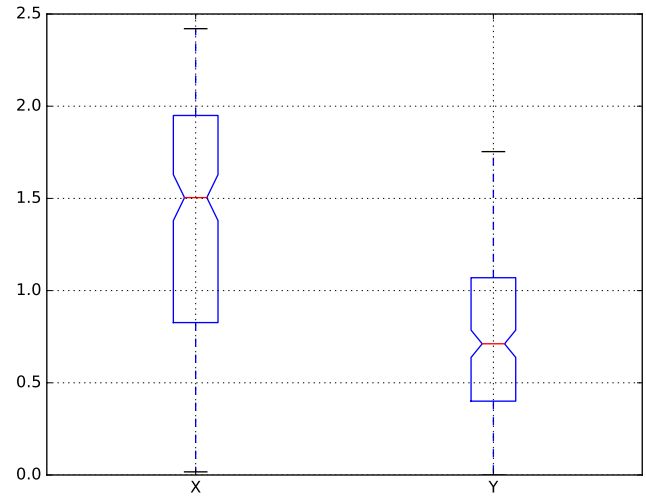


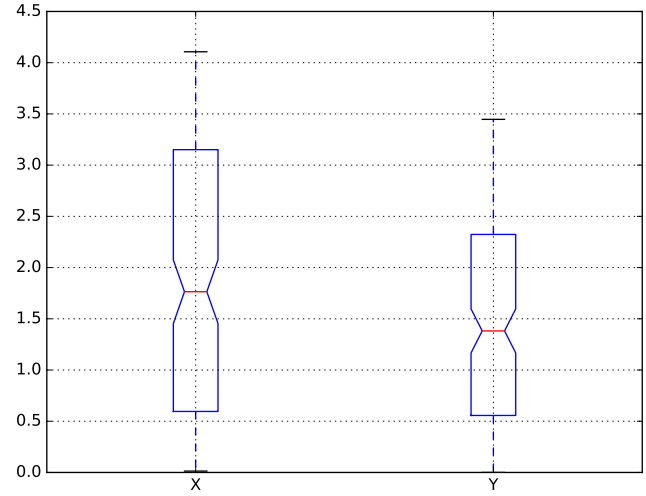
Figure A.7: "Simple oscillator" simulated using explicit Gauss-Seidel method. The "step size" is 0.5 here.



(a) Cumulative difference of results between OpenModelica and Gauss-Seidel method for “Lotka Volterra” system.



(b) Cumulative difference of results between OpenModelica and Gauss-Seidel method for “simple oscialltor”.



(c) Cumulative difference of results between OpenModelica and Gauss-Seidel method for “Van der Pol” oscillator.

Figure A.8: Difference of results between Gauss-Seidel method and OpenModelica. The whisker bars show how the values produced at each time step using Gauss-Seidel method differ from the values of OpenModelica.

B Results of Master-Slave Algorithms

B.1 Waveform Relaxation Algorithm

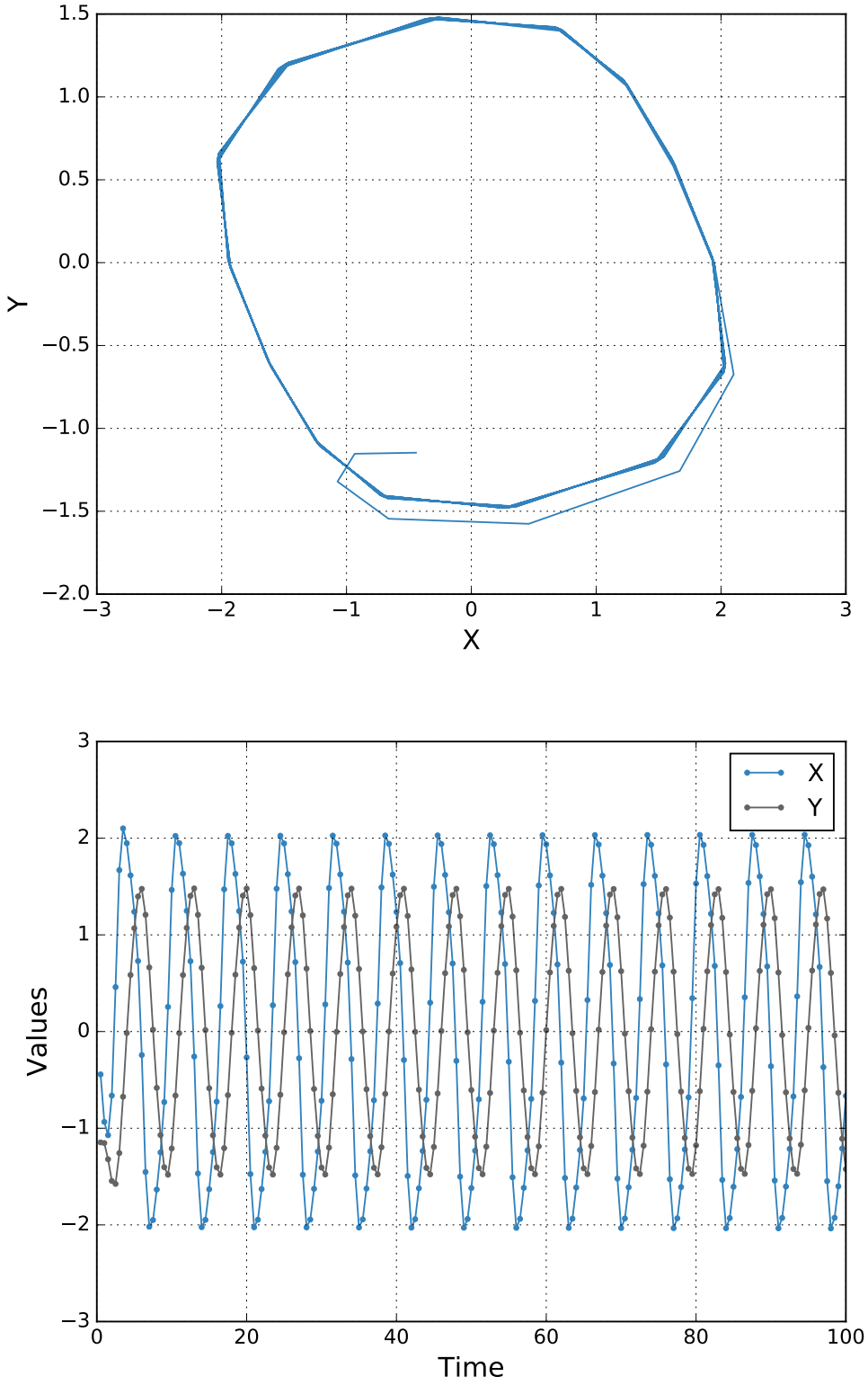


Figure B.1: "Van der Pol" oscillator simulated using the waveform relaxation algorithm, the "step size" is 0.5 here.

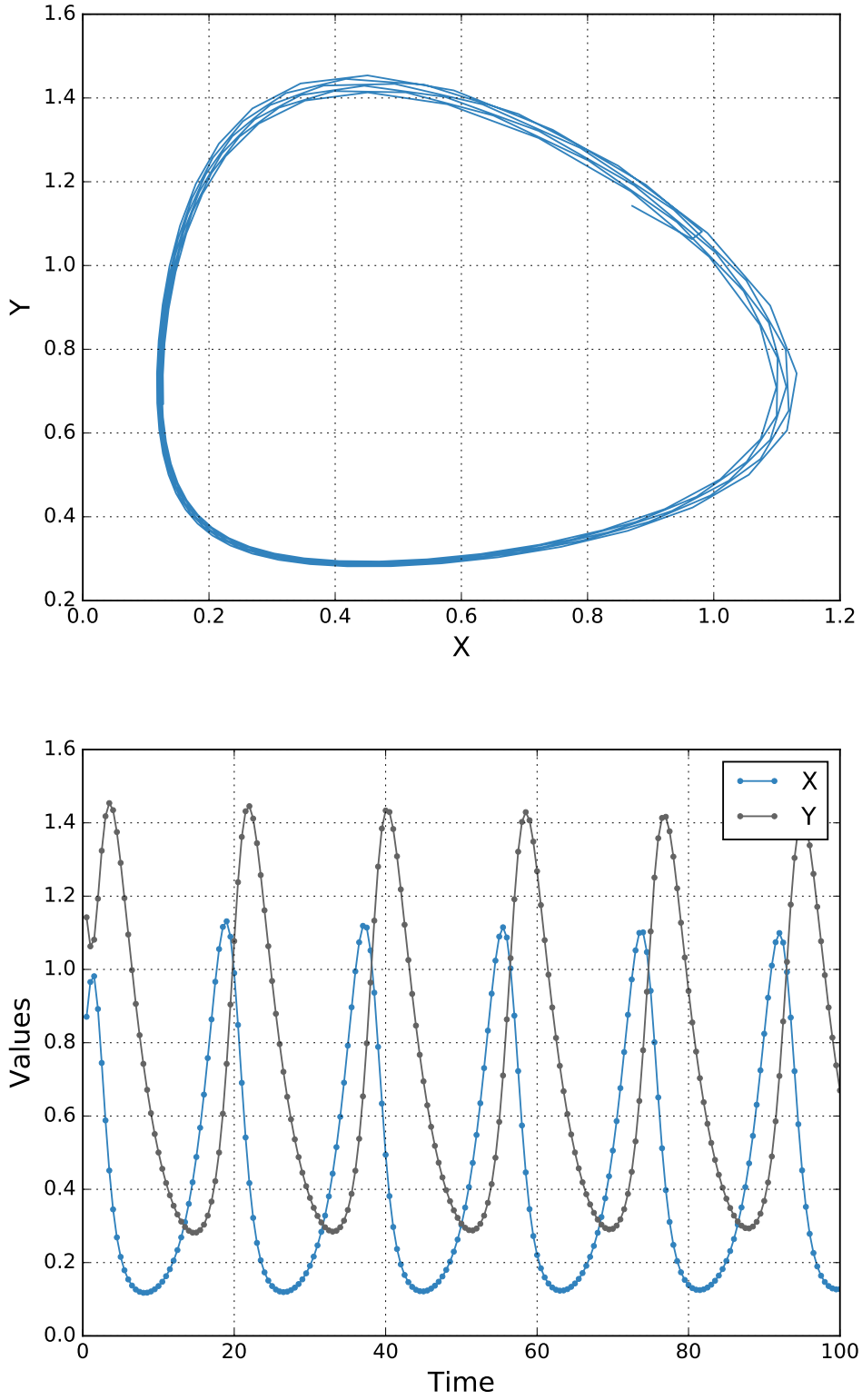


Figure B.2: "Lotka Volterra" system simulated using the waveform relaxation algorithm,the "step size" is 0.5 here.

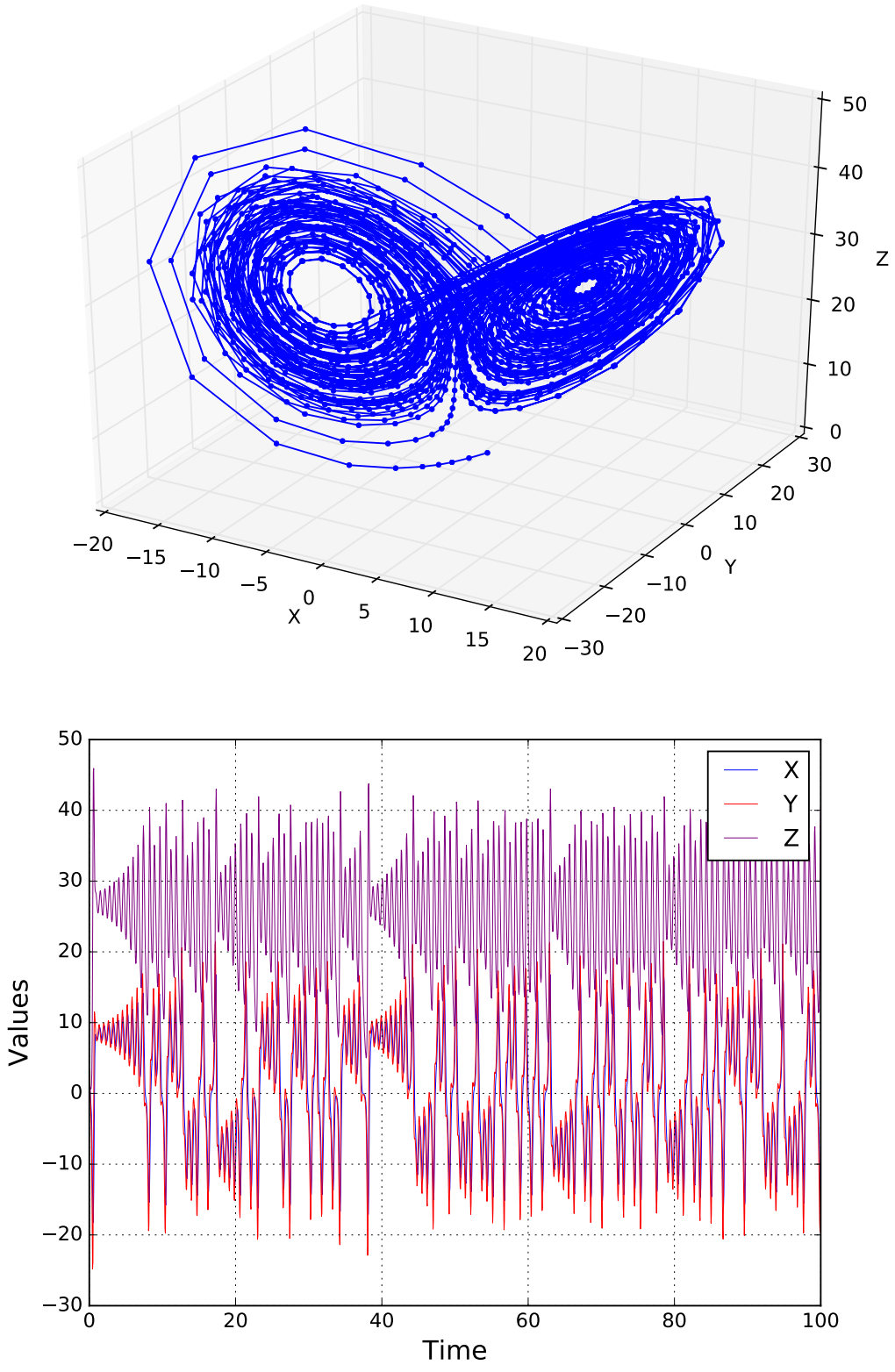
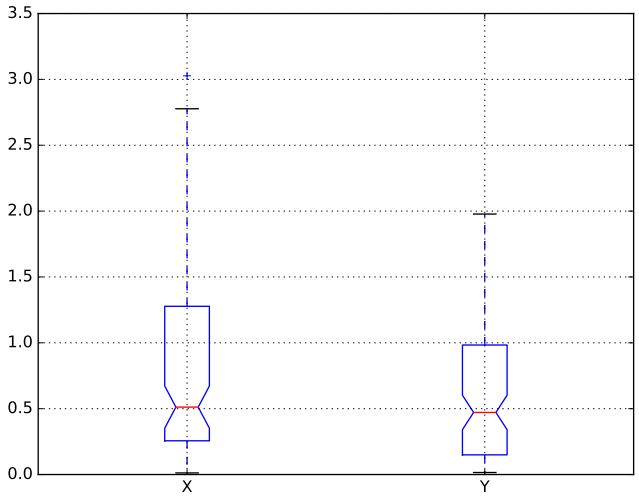
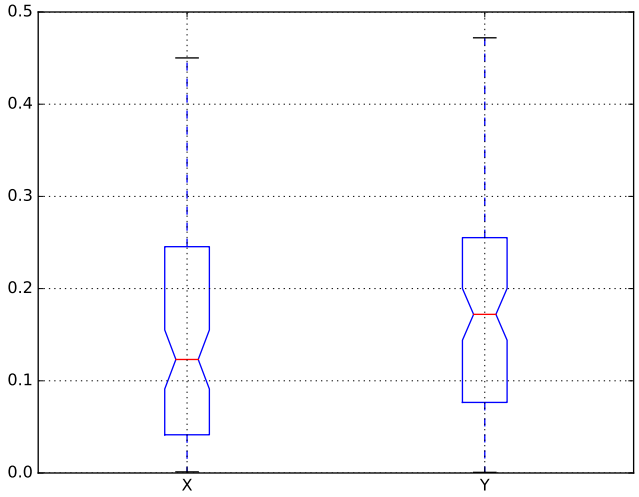


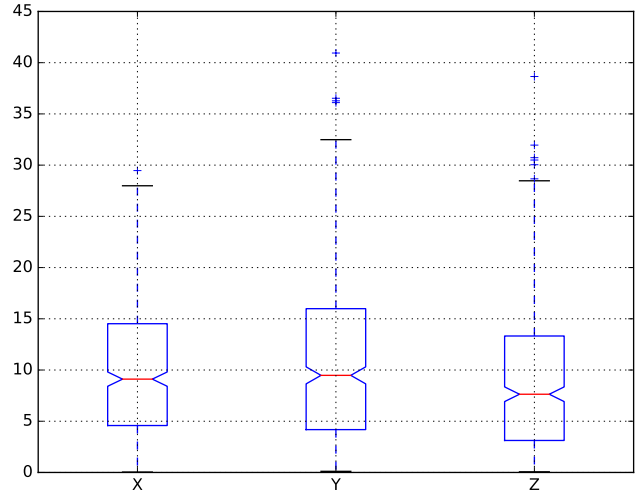
Figure B.3: "Lorenz attractor" simulated using the waveform relaxation algorithm, the "step size" is 0.05 here.



(a) Cumulative difference of results between OpenModelica and WR algorithm for "Van der Pol" oscillator.



(b) Cumulative difference of results between OpenModelica and WR algorithm for "Lotka Volterra" system.



(c) Cumulative difference of results between OpenModelica and WR algorithm for "Lorenz attractor".

Figure B.4: Difference of results between WR algorithm and OpenModelica. The whisker bars show how the values produced at each time step using WR algorithm differ from the values of OpenModelica.

B.2 Semi-Implicit Algorithm

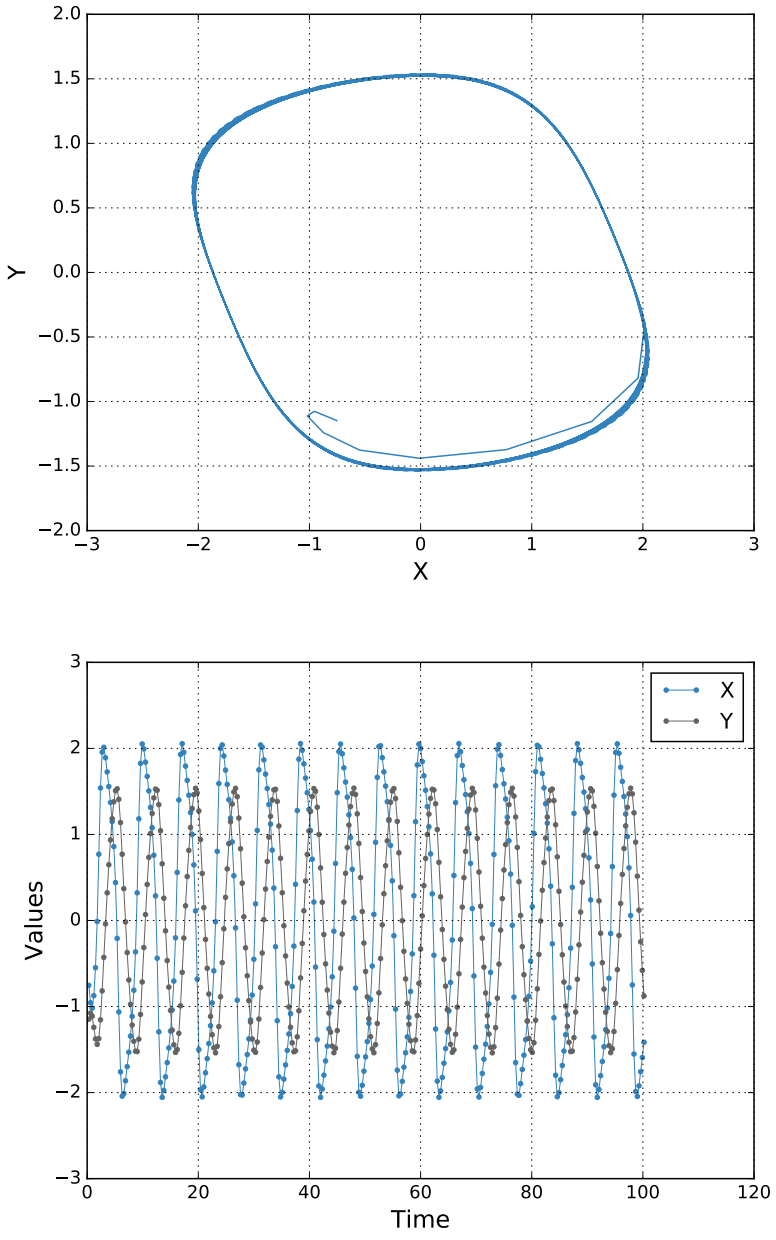


Figure B.5: "Van der Pol" oscillator simulated using the semi-implicit algorithm, the "step size" is 0.5 here.

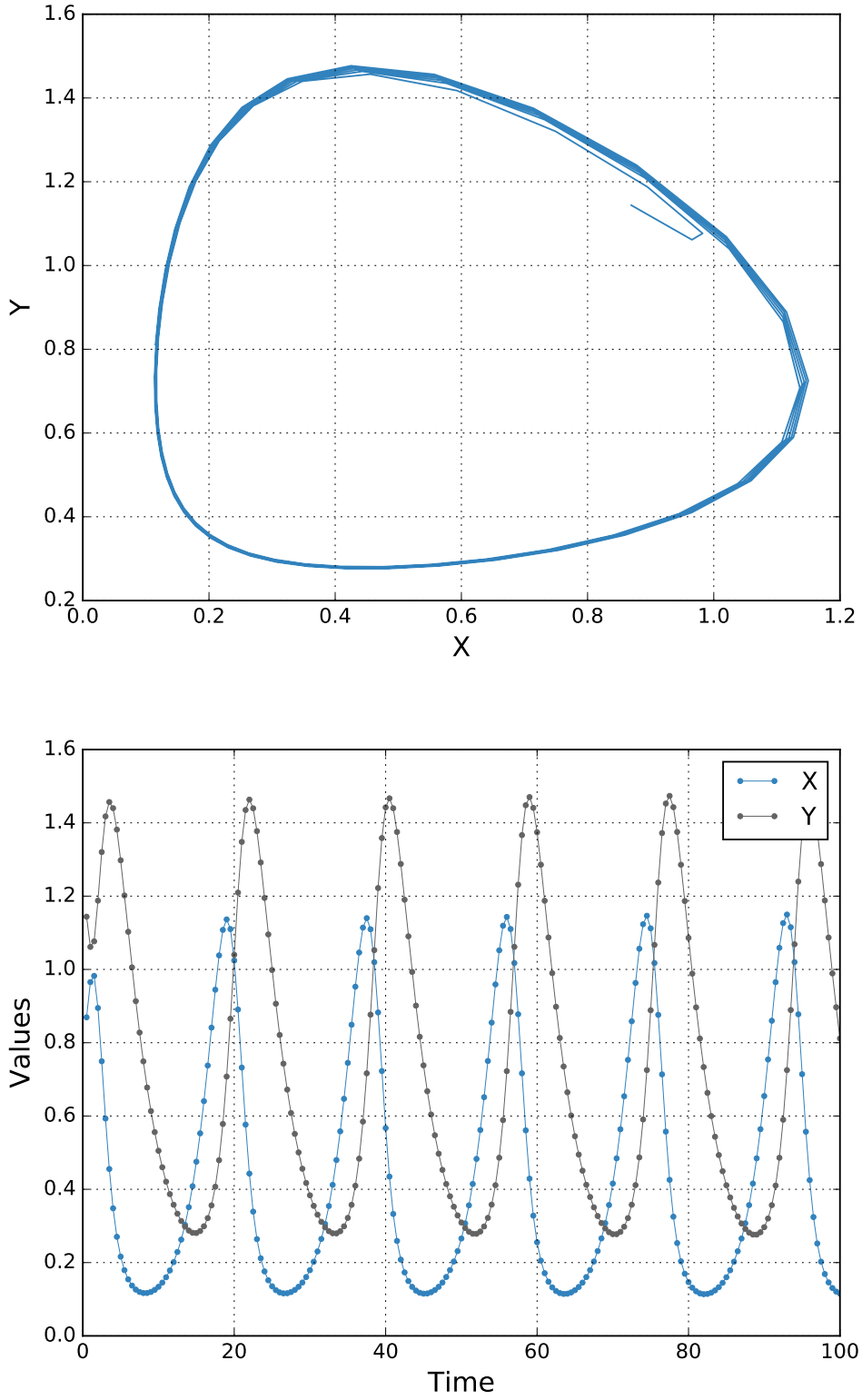


Figure B.6: "Lotka Volterra" system simulated using the semi-implicit algorithm,the "step size" is 0.5 here.

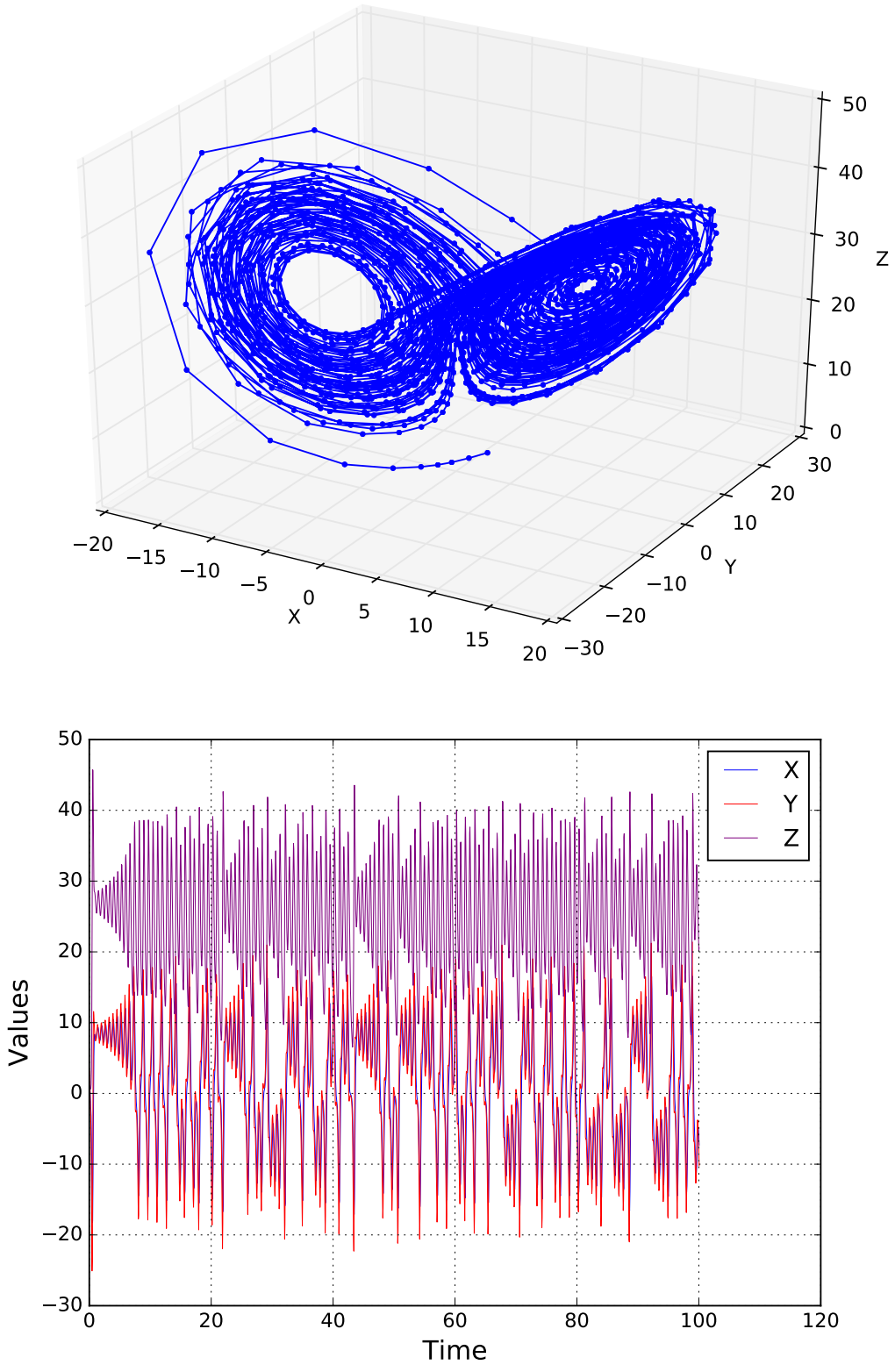
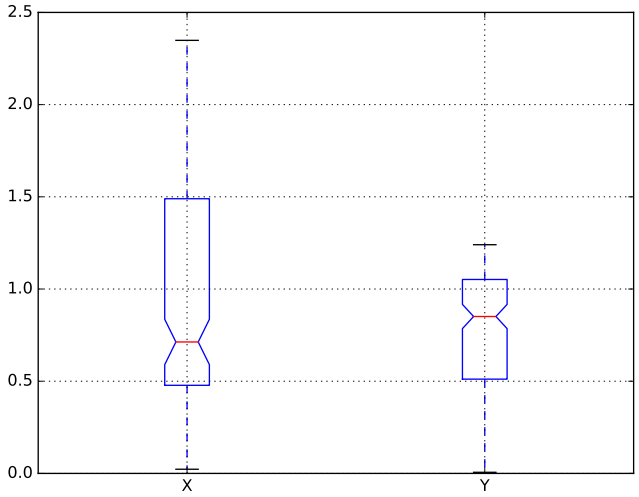
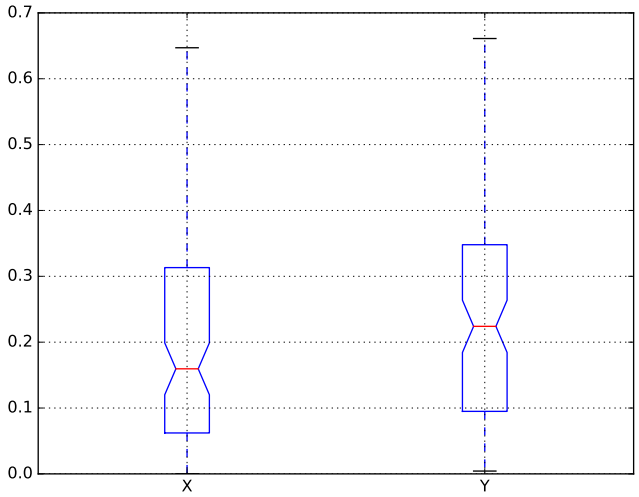


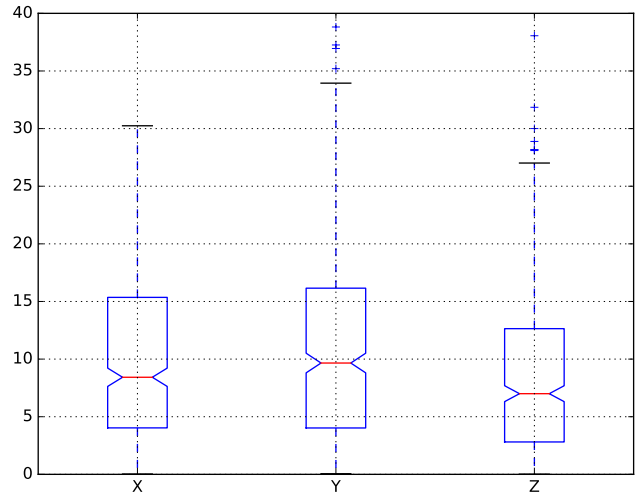
Figure B.7: "Lorenz attractor" simulated using the semi-implicit algorithm, the "step size" is 0.05 here.



(a) Cumulative difference of results between OpenModelica and semi-implicit algorithm for “Van der Pol” oscillator.



(b) Cumulative difference of results between OpenModelica and semi-implicit algorithm for “Lotka Volterra” system.



(c) Cumulative difference of results between OpenModelica and semi-implicit algorithm for “Lorenz attractor”.

Figure B.8: Difference of results between semi-implicit algorithm and OpenModelica. The whisker bars show how the values produced at each time step using semi-implicit algorithm differ from the values of OpenModelica.

B.3 Strong Coupling using SUNDIALS

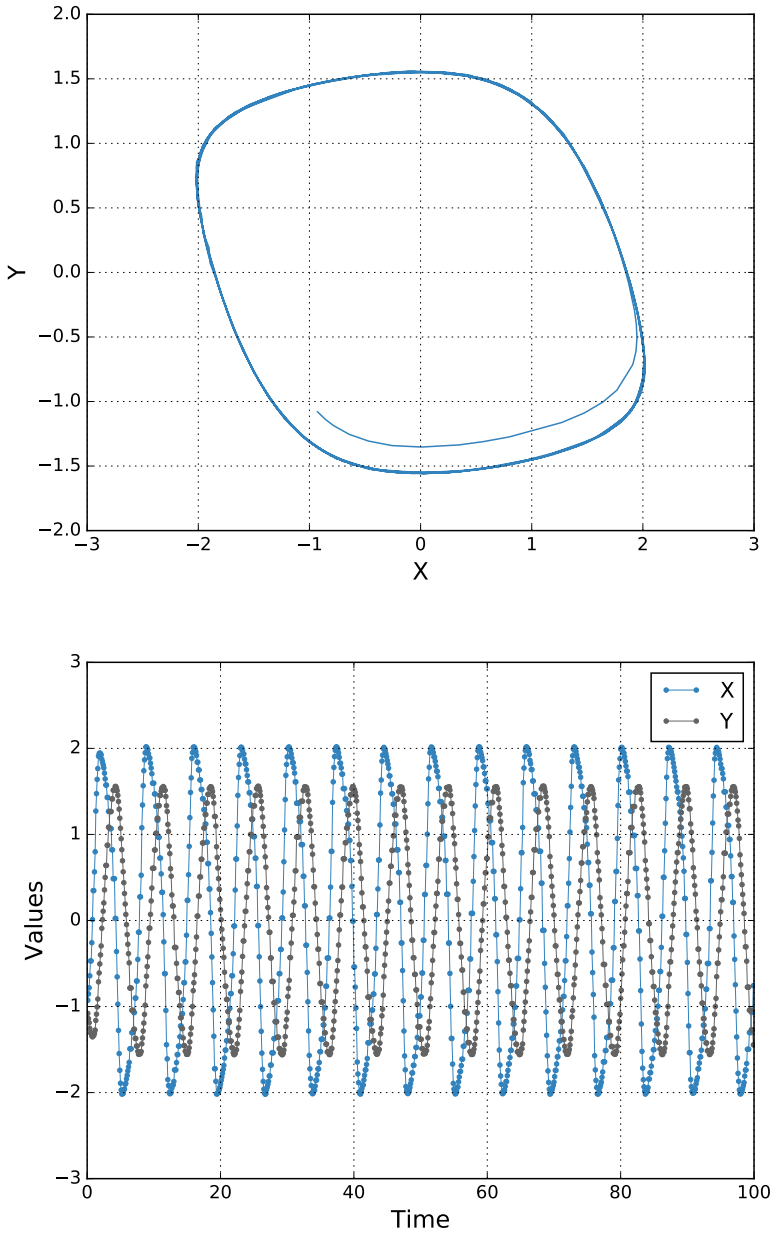


Figure B.9: "Van der Pol" oscillator simulated using the strong coupling algorithm, the "step size" is 0.5 here.

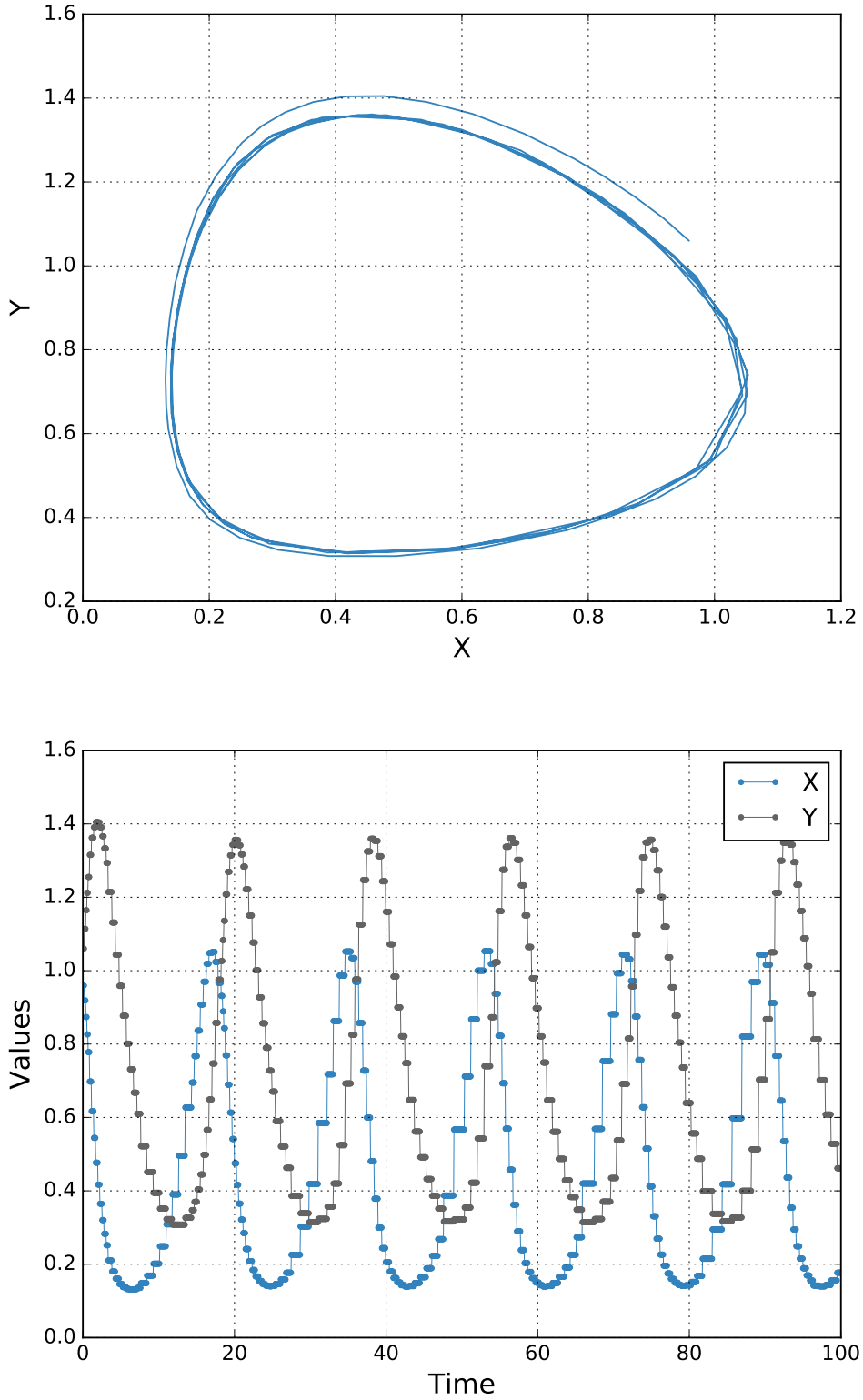


Figure B.10: "Lotka Volterra" system simulated using the strong coupling algorithm,the "step size" is 0.5 here.

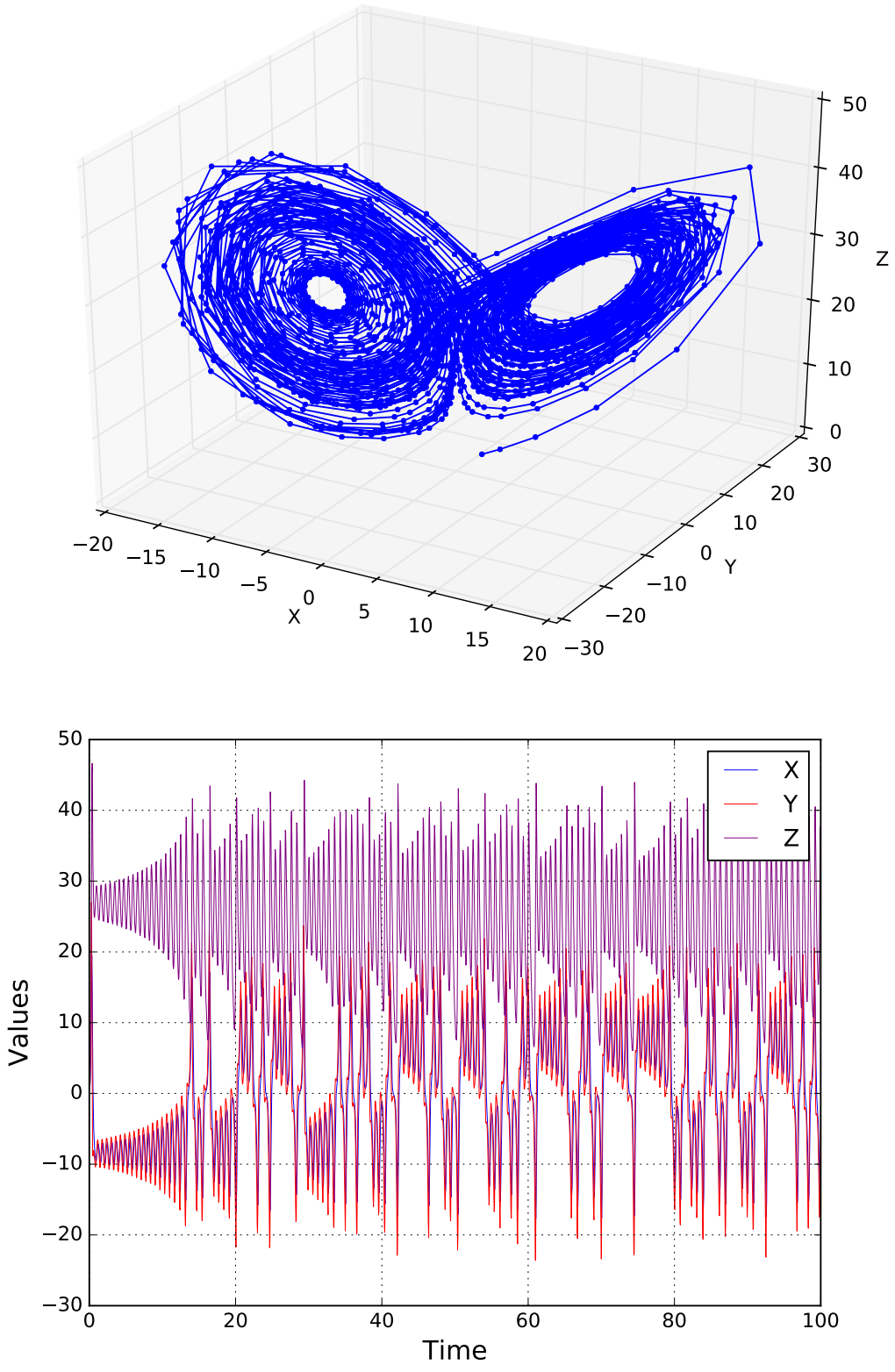
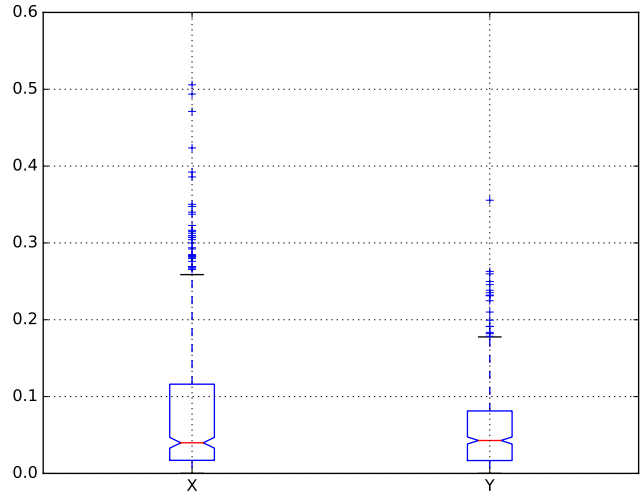
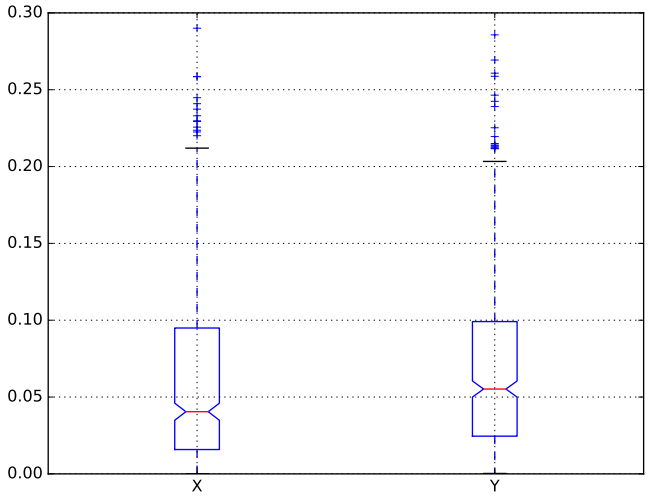


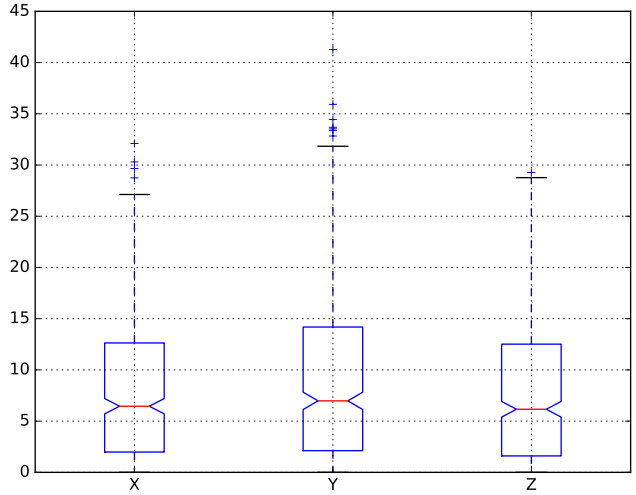
Figure B.11: "Lorenz attractor" simulated using the strong coupling algorithm, the "step size" is 0.05 here.



(a) Cumulative difference of results between OpenModelica and strong coupling algorithm for "Van der Pol" oscillator.



(b) Cumulative difference of results between OpenModelica and strong coupling algorithm for "Lotka Volterra" system.



(c) Cumulative difference of results between OpenModelica and strong coupling algorithm for "Lorenz attractor".

Figure B.12: Difference of results between strong coupling algorithm and OpenModelica. The whisker bars show how the values produced at each time step using strong coupling algorithm differ from the values of OpenModelica.

LITERATURE

[AGDCP15] AWAIS, Muhammad U. ; GAWLIK, Wolfgang ; DE-CILLIA, Gregor ; PALENSKY, Peter: Hybrid simulation using SAHISim framework. In: *Proceedings of the 8th International Conference on Simulation Tools and Techniques* ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2015, S. 273–278

[AME+13] AWAIS, Muhammad U. ; MUELLER, Wolfgang ; ELSHEIKH, Atiyah ; PALENSKY, Peter ; WIDL, Edmund: Using the HLA for Distributed Continuous Simulations. In: *proceeding of: The 8th EUROSIM Congress on Modelling and Simulation*, 2013

[APE+13] AWAIS, M. ; PALENSKY, P. ; ELSHEIKH, A. ; WIDL, E. ; STIFTER, M.: The High Level Architecture RTI as a master to the Functional Mock-up Interface components. In: *International Workshop on Cyber-Physical System (CPS) and its Computing and Networking Design (ICNC 2013)*, 2013, S. 315–320

[APM+13] AWAIS, M.U. ; PALENSKY, P. ; MUELLER, W. ; WIDL, E. ; ELSHEIKH, A.: Distributed hybrid simulation using the HLA and the Functional Mock-up Interface. In: *Industrial Electronics Society, IECON 2013 - 39th Annual Conference of the IEEE*, 2013. – ISSN 1553–572X, S. 7564–7569

[Arn10] ARNOLD, Martin: Stability of sequential modular time integration methods for coupled multibody system models. In: *Journal of computational and nonlinear dynamics* 5 (2010), Nr. 3, S. 031003

[AW04] ATTIYA, Hagit ; WELCH, Jennifer: *Distributed computing: fundamentals, simulations, and advanced topics*. Bd. 19. John Wiley & Sons, 2004

[BBC+10] BEEKER, Emmet ; BELL, David ; CESARE, Sergio d. ; EM, Don C. ; ELDER, Mark [u. a.]: Standard for Commercial-off-the-shelf Simulation Package Interoperability Reference Models-SISO-STD-006-2010 / SISO COTS Simulation Package Interoperability Product Development Group. 2010. – Forschungsbericht

[BOA+11] BLOCHWITZ, T. ; OTTER, M. ; ARNOLD, M. ; BAUSCH, C. ; CLAUSS, C. [u. a.]: The Functional Mockup Interface for Tool independent Exchange of Simulation Models. In: *Modelica'2011 Conference*, 2011, S. 20–22

[BOAk+12] BLOCHWITZ, Torsten ; OTTER, Martin ; Å KESSON, Johan ; ARNOLD, Martin ; CLAUSS, Christoph ; ELMQVIST, Hilding ; FRIEDRICH, Markus ; JUNGHANNS, Andreas ; MAUSS, Jakob ; NEUMERKEL, Dietmar ; OLSSON, Hans ; VIEL, Antoine: Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models, 2012

[BPM+04] BRUTZMAN, Don ; PULLEN, J. M. ; MORSE, Katherine L. ; TOLK, Andreas ; ZYDA, Michael. *Extensible Modeling and Simulation Framework (XMSF)*. `https://www.movesinstitute.org/xmsf/xmsf.html`. 2004

[Bro65] Broyden, Charles G.: A class of methods for solving nonlinear simultaneous equations. In: *Mathematics of computation* (1965), S. 577–593

[BS86] Breitenecker, Felix ; Solar, Dietmar: Models, Methods, Experiments-Modern aspects of simulation languages. In: *Proc. 2nd European Simulation Conference, Antwerpen*, 1986, S. 195–199

[BS10a] Busch, M ; Schweizer, B: Numerical stability and accuracy of different co-simulation techniques: analytical investigations based on a 2-DOF test model. In: *Proceedings of The 1st Joint International Conference on Multibody System Dynamics, IMSD*, 2010, S. 25–27

[BS10b] Busch, Martin ; Schweizer, Bernhard: Explicit and Implicit Solver Coupling: Stability Analysis Based on an Eight-Parameter Test Model. In: *PAMM* 10 (2010), Nr. 1, S. 61–62

[BS12] Busch, Martin ; Schweizer, Bernhard: Coupled simulation of multibody and finite element systems: an efficient and robust semi-implicit coupling approach. In: *Archive of Applied Mechanics* 82 (2012), Nr. 6, S. 723–741

[C+98] Committee, DIS S. [u. a.]: IEEE Standard for Distributed Interactive Simulation-Application Protocols. In: *IEEE Standard* 1278 (1998)

[C+00] Committee, Simulation Interoperability S. [u. a.]. *IEEE standard for modeling and simulation (M&S) high level architecture (HLA)-IEEE std 1516-2000, 1516.1-2000, 1516.2-2000.* 2000

[CK06] Cellier, François E ; Kofman, Ernesto: *Continuous system simulation*. Springer US, 2006

[DFW97] Dahmann, J.S. ; Fujimoto, R.M. ; Weatherly, R.M.: The department of defense high level architecture. In: *Proceedings of the 29th conference on Winter simulation* IEEE Computer Society, 1997, S. 142–149

[DoD02] DoD, US. *TENA-The test and training enabling architecture reference document*. 2002

[DW03] D'Abreu, M.C. ; Wainer, G.A.: Models for continuous and hybrid system simulation. In: *Simulation Conference, 2003. Proceedings of the 2003 Winter* Bd. 1, 2003, S. 641–649 Vol.1

[EJL+03] Eker, J. ; Janneck, J.W. ; Lee, E.A. ; Liu, J. ; Liu, X. ; Ludvig, J. ; Neuendorffer, S. ; Sachs, S. ; Xiong, Y.: Taming heterogeneity-the Ptolemy approach. In: *Proceedings of the IEEE* 91 (2003), Nr. 1, S. 127–144

[EWP12] Elsheikh, A. ; Widl, E. ; Palensky, P.: Simulating complex energy systems with Modelica: A primary evaluation. In: *2012 6th IEEE International Conference on Digital Ecosystems Technologies (DEST)* IEEE, 2012, S. 1–6

[FCK10] Floros, Xenofon ; Cellier, François E ; Kofman, Ernesto: Discretizing Time or States? A Comparative Study between DASSL and QSS-Work in Progress Paper. In: *3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools* Citeseer, 2010, S. 107

[Fuj00] Fujimoto, RM: *Parallel and Distributed Simulation Systems*. New York, USA: John Wiley& Sons, 2000

[GGW07] Gregersen, JB ; Gijsbers, PJA ; Westen, SJP: OpenMI: Open modelling interface. In: *Journal of Hydroinformatics* 9 (2007), Nr. 3, S. 175–191

[Gus88] Gustafson, John L.: Reevaluating Amdahl's law. In: *Communications of the ACM* 31 (1988), Nr. 5, S. 532–533

[GWR86] Gorman, M ; Widmann, PJ ; Robbins, KA: Nonlinear dynamics of a convection loop: a quantitative comparison of experiment with theory. In: *Physica D: Nonlinear Phenomena* 19 (1986), Nr. 2, S. 255–267

[HBG+05] Hindmarsh, Alan C. ; Brown, Peter N. ; Grant, Keith E. ; Lee, Steven L. ; Serban, Radu ; Shumaker, Dan E. ; Woodward, Carol S.: SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. In: *ACM Transactions on Mathematical Software (TOMS)*

31 (2005), Nr. 3, S. 363–396

[HDS+04] Hill, C. ; DeLuca, C. ; Suarez, M. ; Silva, A. D. [u. a.]: The architecture of the earth system modeling framework. In: *Computing in Science & Engineering* 6 (2004), Nr. 1, S. 18–28

[KJ01] Kofman, Ernesto ; Junco, Sergio: Quantized-state systems: a DEVS Approach for continuous system simulation. In: *Transactions of the Society for Modeling and Simulation International* 18 (2001), Nr. 3, S. 123–132

[KK04] Knoll, Dana A. ; Keyes, David E.: Jacobian-free Newton–Krylov methods: a survey of approaches and applications. In: *Journal of Computational Physics* 193 (2004), Nr. 2, S. 357–397

[KKK03] Kim, Y.J. ; Kim, J.H. ; Kim, T.G.: Heterogeneous simulation framework using DEVS BUS. In: *Simulation* 79 (2003), Nr. 1, S. 3–18

[KS00] Kübler, R ; Schiehlen, W: Two methods of simulator coupling. In: *Mathematical and Computer Modelling of Dynamical Systems* 6 (2000), Nr. 2, S. 93–113

[KW14] Ketcheson, David ; bin Waheed, Umair: A comparison of high-order explicit Runge–Kutta, extrapolation, and deferred correction methods in serial and parallel. In: *Communications in Applied Mathematics and Computational Science* 9 (2014), Nr. 2, S. 175–200

[KWD99] Kuhl, F. ; Weatherly, R. ; Dahmann, J.: *Creating computer simulation systems: an introduction to the high level architecture*. Prentice Hall PTR, 1999

[LFB08] Lundvall, Håkan ; Fritzson, Peter ; Bachmann, Bernhard: Event handling in the openmodelica compiler and runtime system. (2008)

[LJO05] Larson, J. ; Jacob, R. ; Ong, E.: The Model Coupling Toolkit: A new Fortran90 toolkit for building multiphysics parallel coupled models. In: *International Journal of High Performance Computing Applications* 19 (2005), Nr. 3, S. 277–292

[LK01] Lim, Seong Y. ; Kim, Tag G.: Hybrid Modeling and Simulation Methodology based on DEVS formalism. In: *SUMMER COMPUTER SIMULATION CONFERENCE* Society for Computer Simulation International; 1998, 2001, S. 188–193

[LRSV82] Lelarasmee, Ekachai ; Ruehli, Albert E. ; Sangiovanni-Vincentelli, Alberto L.: The waveform relaxation method for time-domain analysis of large scale integrated circuits. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 1 (1982), Nr. 3, S. 131–145

[MOD14] Mets, Kevin ; Ojea, J ; Develder, Chris: Combining Power and Communication Network Simulation for Cost-Effective Smart Grid Analysis. In: *Communications Surveys & Tutorials,* Issue: 99 (2014), S. xx

[OR00] Ortega, James M. ; Rheinboldt, Werner C.: *Iterative solution of nonlinear equations in several variables*. Bd. 30. Siam, 2000

[PK13] Palensky, Peter ; Kupzog, Friederich: Smart Grids. In: *Annual Review of Environment and Resources* 38 (2013), Nr. 1, S. 201–226

[R+94] Rowson, James [u. a.]: Hardware/software co-simulation. In: *Design Automation, 1994. 31st Conference on* IEEE, 1994, S. 439–440

[SAC12] Schierz, Tom ; Arnold, Martin ; Clauss, Christoph: Co-simulation with communication step size control in an FMI compatible master algorithm. In: *9th International Modelica Conference. Munich*, 2012

[SBE+14] Sicklinger, S ; Belsky, V ; Engelmann, B ; Elmqvist, H ; Olsson, H ; Wüchner, R ; Bletzinger, K-U: Interface Jacobian-based Co-Simulation. In: *International Journal for Numerical Methods in Engineering* 98 (2014), Nr. 6, S. 418–444

[SH05] Serban, Radu ; Hindmarsh, Alan C.: CVODES: the sensitivity-enabled ODE solver in SUNDIALS. In: *ASME 2005 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference* American Society of Mechanical

Engineers, 2005, S. 257–269

[SK11] Sung, Changho ; Kim, Tag G.: Framework for simulation of hybrid systems: Interoperation of discrete event and continuous simulators using HLA/RTI. In: *Proceedings of the 2011 IEEE Workshop on Principles of Advanced and Distributed Simulation* IEEE Computer Society, 2011, S. 1–8

[SL14a] Schweizer, Bernhard ; Lu, Daixing: Semi-implicit co-simulation approach for solver coupling. In: *Archive of Applied Mechanics* 84 (2014), Nr. 12, S. 1739–1769

[SL14b] Schweizer, Bernhard ; Lu, Daixing: Stabilized index-2 co-simulation approach for solver coupling with algebraic constraints. In: *Multibody System Dynamics* 34 (2014), Nr. 2, S. 129–161

[Val09] Valasek, Michael: Modeling simulation and control of mechatronical systems. In: *Simulation Techniques for Applied Dynamics* 507 (2009), S. 75

[Van00] Vangheluwe, H.L.M.: DEVS as a common denominator for multi-formalism hybrid systems modelling. In: *Computer-Aided Control System Design, 2000. CACSD 2000. IEEE International Symposium on* IEEE, 2000, S. 129–134

[VG01] Van Glabbeek, Rob J.: The linear time-branching time spectrum I - The semantics of concrete, sequential processes. In: *Handbook of Process Algebra, chapter 1* Elsevier, 2001

[WH91] Wanner, G ; Hairer, E: *Solving ordinary differential equations II: Stiff and differential-algebraic problems*. Bd. 1. Springer-Verlag, Berlin, 1991

[Win99] Winsberg, E.: Sanctioning models: The epistemology of simulation. In: *Science in Context* 12 (1999), Nr. 02, S. 275–292

[WPE12] Widl, E. ; Palensky, P. ; Elsheikh, A.: Evaluation of two approaches for simulating cyber-physical energy systems. In: *Proceedings of the 38th IEEE Conference on Industrial Electronics IECON 2012*. Montreal, Canada, 2012

[WW94] Wilson, A.L. ; Weatherly, R.M.: The aggregate level simulation protocol: an evolving system. In: *Proceedings of the 26th conference on Winter simulation* Society for Computer Simulation International, 1994, S. 781–787

INTERNET REFERENCES

[58] Modelica. www.modelica.org.

[59] OpenModelica. https://openmodelica.org/.

[60] Modelisar. www.modelisar.com.

[61] Simulation Interoperability and Standardized Organization (SISO). www.sisostds.org.

[62] Extensible Modeling and Simulation Framework (XMSF). www.movesinstitute.org/xmsf/xmsf.html.

[63] Modeling and Simulation Coordinating Office (M&S CO) . www.msco.mil.

[64] FMU SDK free software development kit. http://www.qtronic.de/en/fmusdk.html.