Improved time representation in discrete-event simulation

Damián Alberto Vicino

UNIVERSITÉ NICE - SOPHIA ANTIPOLIS

ÉCOLE DOCTORALE STIC

SCIENCES ET TECHNOLOGIES DE L'INFORMATION
ET DE LA COMMUNICATION

T H È S E

pour l'obtention du grade de

Docteur en Sciences

de l'Université Nice - Sophia Antipolis

Mention : Informatique

presentée et soutenue par

Damián Alberto Vicino

Amélioration de la représentation du temps dans les simulations à événements discrets

Improved Time Representation in Discrete-Event Simulation

Thèse dirigée par: Prof. Gabriel Wainer et Prof. Françoise Baude

et encadrée par: Dr. Olivier Dalle

Improved Time Representation in Discrete-Event Simulation

by

Damián Alberto Vicino

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Electrical and Computer Engineering

Carleton University
Ottawa, Ontario

Abstract

Improved Time Representation in Discrete-Event Simulation

Discrete-Event Simulation (DES) is a technique in which the simulation engine plays a history following a chronology of events. The technique is called "discrete-event" because the processing of each event of the chronology takes place at discrete points of a continuous timeline. In computer implementations, an event could be represented by a message, and a time occurrence. The message datatype is usually defined as part of the model and the simulator algorithms do not operate with them. Opposite is the case of time variables; simulator has to interact actively with them for reproducing the chronology of events over $\mathbb{R}^+$, which is usually represented by approximated datatypes as floating-point (FP). The approximation of time values in the simulation can affect the timeline preventing the generation of correct results. In addition, it is common to collect data from real systems to predict future phenomena, for example for weather forecasting. These data are measured using measuring instruments and procedures. Measurement results obtained never have perfect accuracy. For them, uncertainty quantifications are included, usually as uncertainty intervals. Sometimes, answering questions require evaluating all values in the uncertainty interval. This thesis proposes datatypes for handling representation of time properly in DES, including irrational and periodic time values. Moreover, we propose a method for obtaining every possible simulation result of DES models when feeding them events with uncertainty quantification on their time component.

Résumé

Amélioration de la représentation du temps dans les simulations à événements discrets

La simulation à événements discrets (SED) est une technique dans laquelle le simulateur joue une histoire suivant une chronologie d'événements, chaque événement se produisant en des points discrets de la ligne continue du temps. Lors de l'implémentation, un événement peut être représenté par un message et une heure d'occurrence. Le type du message n'est lié qu'au modèle et donc sans conséquences pour le simulateur. En revanche, les variables de temps ont un rôle critique dans le simulateur, pour construire la chronologie des événements, dans $\mathbb{R}+$. Or ces variables sont souvent représentées par des types de données produisant des approximations, tels que les nombres flottants. Cette approximation des valeurs du temps dans la simulation peut altérer la ligne de temps et conduire à des résultats incorrects. Par ailleurs, il est courant de collecter des données à partir de systèmes réels afin de prédire des phénomènes futurs, comme les prévisions météorologiques. Les résultats de cette collecte, à l'aide d'instruments et procédures de mesures, incluent une quantification d'incertitude, habituellement présentée sous forme d'intervalles. Or répondre à une question requiert parfois l'évaluation des résultats pour toutes les valeurs comprises dans l'intervalle d'incertitude. Cette thèse propose des types de données pour une gestion sans erreur du temps en SED, y compris pour des valeurs irrationnelles et périodiques. De plus, nous proposons une méthode pour obtenir tous les résultats possibles d'une simulation soumise à des événements dont l'heure d'occurrence comporte une quantification d'incertitude.

Contents

List of Tables

List of Figures

List of Algorithms

List of Listings

List of Formulas

Acronyms

BIPM Bureau International de Poids et Mesures. 20, 21, 54, 82

CPU Central Processing Unit. 48, 69

DEDS Discrete-Event Dynamic Systems. 2

DES Discrete-Event Simulation. v, xii, 2–7, 9, 10, 19–21, 23, 24, 26, 27, 29–33, 40, 41, 54, 69, 72, 73, 82, 83, 106, 117, 118

DEVS Discrete-Event System Specification. xi, xiv, 2, 3, 5, 6, 9–14, 18–20, 24–28, 31, 35, 53, 54, 56, 62, 72, 73, 75, 81, 83, 86, 87, 98–101, 103, 105, 106, 109, 113, 117–119

DTS Discrete-Time Simulation. 83

FD-DEVS Finite & Deterministic DEVS. 18, 103, 105

FEL Future Events List. 28, 107, 110–112

FF-DEVS Finite-Forkable DEVS. 5, 6, 99–103, 105, 118, 119

FP floating-point. v, xii, xvii, 3, 23–35, 37, 40–49, 53–56, 65, 66, 68, 72, 73, 117

GCD Greatest Common Divisor. 36, 61, 65, 67

HI High-level of Input coupling. xii, 69, 113–115

LCM Lowest Common Multiple. 61, 65, 67

LI Low-level of Interconnections. xii, xiii, 68, 69, 113–116

MBFP Multi-Base floating-point. xvii, 39, 55, 66–69, 71, 74, 117, 118

NaN Not-a-Number. 33

non-SP non-Schedule-Preserving. xii, xiv, 84, 85, 103, 104

PDEVS Parallel-DEVS. xii, xiv, 5, 6, 14–18, 20, 24, 26, 31, 53, 106, 108–110, 113, 115, 118

QSS Quantized-State Systems. 19

RSFP Rational-Scaled floating-point. xvi, xvii, 39, 55–57, 59–61, 63–69, 74, 117, 118

RT-DEVS Real-Time DEVS. 18

RTA-DEVS Rational Time Advance DEVS. 18, 38, 72

SF Safe-Float. xi, xv, 39–54, 56, 69, 71, 117

SP-DEVS Schedule Preserving DEVS. 18, 103, 105

Sym-DEVS Symbolic DEVS. 18, 38, 73

Chapter 1

Introduction

Since we are born, we try to figure out how the world around us works, how to interact with it and how to predict the responses of the world to our stimulus. The first approach we develop is the experimental one. In this approach, we repeat and refine experiments until obtaining the desired result. For example, this is the process we follow to learn to walk and talk.

At some point in our lives, our curiosity leads us to start thinking of problems that are theoretically or practically impossible to solve using the experimental approach. For example, when sending a robot to Mars, it would be extremely expensive to start sending robots up until one arrives to Mars, or, if we want to know the age of the Sun, we cannot measure its initial value. These cases need to be studied in abstract. In order to do that, different Modeling techniques were invented.

A model is the abstract representation of an entity we want to study. As an abstraction, the model cannot capture all the properties of the studied entity. The same entity may be modeled differently to study different problems. For example, when studying how strong can be the hit of a hammer, we may not model its color, and when we are studying its kinematics, we may not model the weight of its pieces. In both cases, we are modeling a hammer, but the model in each is intended to answer different questions. A good model should focus on the properties considered important for the study in which it will be used.

Once the experiments are modeled properly, the results obtained experimenting in the abstract world can be interpreted to predict the results of similar experiments in reality. If the experimentation is reproduced in reality and the results do not match the interpretation we are in presence of an invalid model.

The analytical method for modeling, defined centuries ago, uses equations developed for reasoning over experimental results, observation and axioms previously defined. An example of this approach is the mechanics equations defined by Newton. In this method, it is usually hard to formulate equations for new systems, and when the equations are defined, it may be too complex to solve them.

In general, for cases where the models are properly defined and the methods to solve the equations

are known, its too complex scaling composing models. For example, the equations defined for Newtonian mechanics can be easily applied to study the interaction between two pieces in a car. Nevertheless, it might be impossible to compose the equations to study the transfer of movement from the explosion inside the engine all way through the wheels of the car describing the position of each piece participating.

Models defined and studied using analytical methods provide good results for many continuous systems. These models were usually appropriate for studying phenomena in nature. Nevertheless, they are not good for human defined systems as transit, integrated circuits, or computers, which are intrinsically discrete. New methods needed to be developed for this kind of analysis.

In the last century, the introduction of computers allowed to work with symbolic manipulations that are more complex. These symbolic manipulations allowed the study of models through the generation of traces of their dynamic behavior helping to understand the evolution of the real system. We call Simulation the generation of these traces, and the device to run these algorithms a simulator. The simulator can be a person, a mechanical machine, a computer, a grid of computers, or anything achieving the goal of executing the models. The results obtained by simulating help to understand the dynamic behavior of the original system, and to produce conclusions.

To improve collaboration, sharing, and reuse about the models and simulations developed, several modeling formalisms had been proposed. Some of them are more convenient than others for each project. These formalisms can be classified according to the representation used for their time and state variables; each of these variables is classified as discrete or continuous.

In [Wai09], the modeling formalisms are classified in four groups:

- The group of formalisms representing time and state with continuous variables, known as Continuous Variable Dynamic Systems; an example of this could be those models defined by differential equations.

- The group of formalisms representing time with discrete variables and state with continuous ones, known as Discrete Time Dynamic Systems; an example of use is the study of sampled electronic devices signals.

- The group of formalisms representing state and time with discrete variables, known as Discrete Dynamic Systems; an example of use is the modeling of digital computers with clocks providing synchronism.

- Finally, the group of formalisms representing state with discrete variables and time with continuous ones, known as Discrete-Event Dynamic Systems (DEDS); an example of use is the modeling of a traffic light with pedestrians' call button, which can be pushed at any time in a continuous real timeline.

In the context of this thesis, we focus our research in Discrete-Event Simulation (DES), the simulation of DEDS models. We have special interest in the family of Discrete-Event System Specification (DEVS) formalisms [ZPK+76, ZPK00].

DEVS is a formalism based on Systems Theory for the class of DEDS defined by Bernard Zeigler in

the 70s. In DEVS, basic models are defined by their internal behavior and how they react to external inputs. These basic models can be hierarchically composed to produce models that are more complex. The algorithms for the simulation are independent of the models, and several computer simulators implementing them were developed in several architectures including distributed, parallel, sequential, object oriented, and functional. We cover the details of the DEVS modeling specification and simulation algorithms on Chapter 2.

It was proved for a large quantity of Modeling and Simulation formalisms that they model a strict subset of what DEVS can model. For most of those formalisms, translation methods are known. Using these translations, models originally developed in different formalisms can be hierarchically combined, to produce models that are more complex. Here, DEVS has the role of being the glue between formalisms for multi-formalism simulation [MZRM09].

The datatype for state variables is usually defined as part of the model and the simulator algorithms do not operate with these variables. Opposite is the case of time variables; the simulator has to interact actively with them for reproducing the chronology of events over $\mathbb{R}^+$, which is usually represented by approximated datatypes as floating-point (FP). The approximation of time values in the simulation can affect the timeline preventing the generation of correct results. In addition, the existence of incomputable numbers prevents us from finding a datatype able to represent complete segments in $\mathbb{R}$.

For the most popular formalisms, several implementations have been developed in various languages and platforms (we present a survey of the datatypes used by eleven popular open source DES simulators in Chapter 3). These simulators usually represent time with one of the following datatypes: FP, fixed-point, Integer, or Integer tuples. These datatypes are conveniently implemented in processors, or they are easily implementable combining native ones. Nevertheless, these datatypes have limitations in their usage that need to be noticed.

Working unaware of these limitations can be translated into issues affecting the correct generation of simulation timeline. These issues can be classified as time shifting errors, event reordering errors, and Zeno problems.

Furthermore, for DES, these errors can break the causality chain of the simulation [Lac10]. In DES, the state of the simulation can be thought of as a function of its history, producing a causal relation between them. When an event in the history is approximated, it may diverge the resulting trajectory of events. When this happens, we say that the causality chain was broken. Thus, the results obtained from the simulation are incorrect.

Current simulator implementations are silent about these errors, usually because it is impossible to detect them properly using their time datatypes. For example, when using the FP arithmetic implemented in the processor hardware, if no mechanism is available to notify about rounding, then the hardware outputs a value as accurate as possible.

Literature proposes some alternative approaches to be used, as rational interval datatypes, or symbolic algebra. This approaches have limitations too. We discuss them in Chapter 3.

A problem that none of the mentioned datatypes properly solve is the definition, and operation, of

computable irrational numbers. Only the rational interval and the symbolic algebra approaches attempt to represent them. However, neither of them can compare close values, and the second only supports non transcendental numbers. This restricts the set of models and simulations to those never using simultaneous events.

Another problem of interest when studying timelines is related to the uncertainty of the represented time lapses. In general, models could be simulated to predict future phenomena based on data collected in reality. When collecting data to define the models, measuring instruments and procedures are used. These instruments usually have a known associated precision. For example, a micrometer is an instrument designed to measure distance with 1 micron precision; when it is properly calibrated every measure taken with could have, i.e., an associated uncertainty of 1 micron; in this case, when the instrument indicates 99 microns, we could assert that the real value is between 98 and 100 microns. Sometimes, in engineering and experimental research, answering a question requires evaluating all the possible values of the measured magnitude.

Likewise, when working with continuous systems [Hoo99], there are methodologies to propagate errors widely studied in calculus and numerical methods. Mathematical tools exist to estimate the error of the output based on the error associated to the input. In discrete time systems, obtaining all the possible results starting from an imprecise input is as easy as evaluating every value in the error interval. This is possible because these systems have only a finite set of time points in each error interval.

Nevertheless, none of these methods exists in DES mathematical tools, making it impossible nowadays to track uncertainties in simulation of DES models. The tools defined for continuous systems cannot be used for DES having discrete states. Nor can the approach taken in discrete time systems because it is possible for uncertainty intervals in DES timelines to include infinite values in $\mathbb{R}$.

1.1 Thesis Objectives

Considering the problems discussed above, the goals of this thesis are to devise a set of data structures and algorithms for handling representation of time properly in DES. We also want to define a method for simulating DES models and feeding them with input events having uncertainty in their time component.

For the first goal, the new datatypes should provide proper representation of time for simulating without producing timeline errors, should include representation for numbers with periodicity, and should include subsets of necessary computable irrational numbers.

The algorithms provided should be able to handle input events with uncertainty in their time component for predicting future phenomena based on measures obtained in the real world. In addition, it should provide the tools to carry these errors through the model simulation. The results of these simulations should generate multiple outputs depending on the uncertainty intervals of the input. After simulating, tracing the origin of each output should be possible. We need this mechanism for understanding how an adjustment of the precision of inputs may alter the obtained results.

1.2 Thesis Contributions

The main contributions of this thesis include:

- The definition of new datatypes for representing time in DES properly for producing correct trajectories, supporting customizable subsets of computable irrational numbers.
- The definition of simulation algorithms for simulating every possible trajectory of a DEVS model when uncertainty quantified input events are introduced, and a subclass of DEVS models where these algorithms can be computed with finite forks in each simulation step.
- The study of a new architecture for sequential DEVS/Parallel-DEVS (PDEVS) simulators, and the implementation of such architecture in a new efficient DEVS simulator for studying its performance.

As a result of this research, the following articles have been published or submitted for publication:

- Damián Vicino, Olivier Dalle, and Gabriel Wainer. Using DEVS models to define fluid based μTP model. Poster session presented at SIGSIM-PADS 2013; Montreal, QC, Canada. This poster proposes a DEVS model for large-scale peer-to-peer file sharing networks using μTP protocol.
- Damián Vicino, Olivier Dalle, and Gabriel Wainer. A datatype for discretized time representation in DEVS. In Proceedings of the 7th International ICST Conference on Simulation Tools and Techniques, pages 11–20. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2014; Lisbon, Portugal. This paper presents the first datatype proposed for precise time and an empirical performance comparison showed.
- Damián Vicino, Chung-Horng Lung, Gabriel Wainer, and Olivier Dalle. Evaluating the Impact of Software-defined Networks' Reactive Routing on BitTorrent Performance. In proceedings of SDN-NGAS 2014; Niagara Falls, ON, Canada. This paper studies the behavior of BitTorrent file propagation on emulated Software-Defined Networks.
- Damián Vicino, Chung-Horng Lung, Gabriel Wainer, and Olivier Dalle. Investigation on software-defined networks' reactive routing against BitTorrent. Journal paper in IET Networks 2015/4. This paper is an extended version of the previous, comparing behavior of BitTorrent in SDN against the behavior of HTTP protocol when used in the same context.
- Mandeep Kaur Guraya, Rupinder Singh Bajwa, Damián Vicino, and Chung-Horng Lung. The Assessment of BitTorrent's Performance Using SDN in a Mesh topology. Poster session presented in the sixth international conference on network of the future (NOF15); Montreal, QC, Canada. This paper explores if the results obtained in previous papers is reproducible in the mesh topology.
- Damián Vicino, Daniella Niyonkuru, Gabriel Wainer, and Olivier Dalle. Sequential PDEVS architecture. In proceedings of SpringSim15-TMS/DEVS, 2015; Alexandria, VA. This paper presents the current architecture, implementation and experimentation of the CDboost simulator.
- Damián Vicino, Olivier Dalle, and Gabriel Wainer. Extending Discrete-Event System Specification simulator to support metrical systems. Poster session presented at SIGSIM-PADS 2015; London, England. This poster presents the problem of introducing events with uncertainty in the time component into simulations and defines the Finite-Forkable DEVS (FF-DEVS) models' class.

- Damián Vicino, Olivier Dalle, and Gabriel Wainer. Using Finite-Forkable DEVS for Decision-Making Based on Time Measured with Uncertainty, In Proceedings of the 8th EAI International Conference on Simulation Tools and Techniques 2015; Athens, Greece. This paper compares FF-DEVS against other subclasses of DEVS, and introduces the simulation algorithms.

1.3 Structure of the Thesis

The rest of this thesis is organized as follows. In Chapter 2, we review the state of the art of the literature and the main concepts used. This review includes time representation, computable numbers, Discrete-Event Simulation, and Measurement Uncertainty topics.

In Chapter 3, we review current datatypes used in computer simulators. We describe these datatypes limitations, and classify how these limitations can affect the timeline of the simulation.

In Chapter 4, we propose four new datatypes for detecting and preventing errors in timelines. We describe the usage scenarios for each of them. And, we end the chapter with an empirical performance evaluation.

In Chapter 5, we proposed a method for supporting irrational numbers in DES timelines.

In Chapter 6, we propose a method for simulating DES models considering input with uncertainty. In addition, we define a subclass of models that could be simulated using this method for any input.

In Chapter 7, we present a sequential architecture for implementing PDEVS simulators. We describe our implementation in C++11 of this architecture. And, we compare it against other simulators using the DEVStone benchmark.

In Chapter 8, we present our conclusions and propose future lines of work.

Chapter 2

Background

2.1 Time representation

Foundational work about Time representations in computers was developed in the 70s and 80s. These works were focused on studying problems related to the synchronization of distributed systems and real time applications [GHJ97]. Lamport formalized time [Lam78] for distributed systems, defining it as a sequence of partially ordered events using the relation "happened before". Later, formal approaches were proposed based in temporal logic [MMP92, MP92]. Temporal logic allows specifying constraints between events and continuous intervals of time. Using temporal logic, it is possible to reason about time constraints and provides formal proofs to properties, datatypes, and algorithms as those explained in Chapter 1.

In [All81, All83], an interval-based representation for time is proposed. This approach defines five relations between the time intervals to be considered: before, equal, meets, overlaps, and during; complementary relations are defined to reach a complete set of thirteen relation operations. In this work, a definition of "now" is provided together with decision algorithms for interpreting information represented using time intervals.

In 2009, Clock Constraint Specification Language (CCSL) [And09] was proposed as a standard to extend the Unified Modeling Language (UML). This new addition is used to describe relations between time instants in dense and discrete time representations. The specification language allows a definition of clocks and relations between them. Using these clocks and relations, formal proofs can be obtained on the behavior of a specified system. The goal of the project is the use of automated model checking tools in systems having time semantics, i.e., simulators.

In modeling and simulation, it is common to allow the definition of instantaneous or close to instantaneous actions. Allowing this kind of actions may lead to Zeno conditions [Lee14], which means that it is possible to have infinite actions in a finite period.

Zeno behavior is not limited to Discrete-Event Simulation (DES), which we discuss in Section 2.4;

it is also possible in continuous systems. In [Lee14], the continuous system described by Formula 2.1 is proposed. This (continuous) function produces infinite oscillations when evaluated with t between 0 and 1.

$$
\begin{aligned}
&x : \mathbb{R} \to \mathbb{R} \\
&x(t) = sin(\frac{2\pi t}{1-t}) \quad \texttt{if } 0 \leq t < 1 \\
&x(t) = 0 \quad \texttt{otherwise}
\end{aligned}
$$

Formula 2.1: Example of a continuous system with Zeno problems

2.2 Computable Numbers

Alan Turing introduced computable real numbers in his foundational paper 'On Computable Numbers, with an Application to the Entscheidungsproblem' [Tur36]. He defines: 'The computable numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means'. Different authors provided several equivalent definitions, for example, an alternative equivalent definition is 'a number is computable if there exist an algorithm to produce each digit of its decimal expansion and for any digit requested it finishes the execution successfully' [Abe01]. There are subsets of real numbers that had been proved non-computable. Examples of such a subset include the family of Chaitin's constants [Cha75].

A new area of applied mathematics was developed based on the theory of computable real numbers, called Computable Calculus [Abe01]. The main goal of this area is spotting the adjustments that need to be done to theorems and properties on real calculus that does not apply to computable real numbers.

For example, since an algorithm can describe each computable number, the set of computable numbers is countable, while real numbers are not [Tur36].

Some interesting results in Computable Calculus are the following non-decidable problems in 'Computable Calculus' by Oliver Aberth [Abe01]:

- In the general case, it is impossible to decide if a computable real number is irrational or not. This is because to be certain about numbers periodicity, it is necessary to check every digit of the numeric expansion, which may be infinite.
- In the general case, it is impossible to decide if a computable real number is greater than zero or equal to zero. It is possible to say that a number is not zero as soon as the first digit different from zero appears in its digit expansion, but there is no guarantee this will ever happen. When the number is effectively zero, infinite digits might need to be checked, which is impossible by finite means. In the case of comparison for greater than zero, the same argument can be used: there is no way to find out if the number is slightly greater than zero or if it is effectively zero in finite steps.
- Other problems can be derived from the previous two examples, which are also non-solvable; for

instance the equality of two numbers and the order of two numbers. This is because if they were solvable, we could subtract the two numbers and it will allow us to solve the previous problems.

Detailed proofs of these non-decidable problems can be found in [Abe01].

To operate with computable numbers, we can use interval arithmetic based on the k-digits decimal expansion of the considered number N, noted N_0, and the number N_1 obtained by adding '1' to its last digit, these numbers can be used as borders of an interval representing the considered number N [Abe01].

In interval arithmetic, the addition of two intervals is defined as the interval where lower end is defined by the addition of the two lower ends and higher end is defined by the addition of the two higher ends [Moo66]. Similar definitions exist for other operations as subtraction, multiplication and division [Moo66].

The definition of comparison operations of intervals is more complicate because the two intervals being compared may intersect; this is why multiple comparison operations need to be defined for them [Abe01, Moo66, All83].

In the case of DES, most simulation is advanced adding time values, and deciding which event to process next. In particular, if we analyze the Discrete-Event System Specification (DEVS) family simulation algorithms proposed by Zeigler [ZPK+76], when using intervals to approximate computable numbers it is enough to define the equality and lower than compare operators for intervals that do not intersect [Abe01]. In the case the intervals approximating two numbers overlap, it is impossible to decide comparison for the numbers being approximated based in the current approximation.

When algorithmic definitions of computable numbers are available, they can be used to get a more accurate approximation, and to compare again using smaller intervals to reduce the possibility of intersection.

If there is a chance that the numbers are equal we have no way to detect it in the general case. Thus we could keep reducing the interval indefinitely without obtaining any definitive result [Abe01].

2.3 Discrete-Event Simulation

Discrete-Event Simulation (DES) is a technique in which the simulation engine plays a history following a chronology of events [ZPK+76, ZPK00, Wai09]. The technique is called "discrete-event" because the processing of each event of the chronology takes place at discrete points of a timeline. The virtual time of the simulation does not require any synchronization with real time. This allows us to predict future phenomena or study complex process happening in a short time. Unless the system being modeled already follows a discretized schedule (i.e., in the case of basic traffic lights), the process of modeling a phenomenon using a discrete-event simulator can be considered as a discretization process.

In the earlier days of DES, multiple modeling and simulation languages were developed, such as the popular SIMULA [DN66]. Most of these languages lacked of formal soundness [Nan81]. The ear-

liest approaches for formalizing DES added time semantics to well-known static modeling approaches as Timed-Automata [AD94], and Generalized Semi-Markovian Processes (GSMP) [Gly89]. Others formalisms focused on concurrency problems emerged at the time, i.e. Petri-nets [Pet81], Calculus of Communicating Systems (CCS) [Mil80], and Communicating Sequential Processes (CSP) [Hoa78].

Among the many DES techniques, we are interested in the DEVS formalism [ZPK+76, ZPK00], which provides a theoretical framework to think about Modeling using a hierarchical, modular approach. This formalism is proven universal for DES modeling, meaning that any model described by other DES formalism has an equivalent model in DEVS. Other characteristic of DEVS is the clear separation between model and simulation: models are described using a formal notation, and simulation algorithms are provided for running any model.

Nowadays, the DEVS term is not only used to designate a specific formalism, but more generally, to designate a family of formalisms derived from the original DEVS. For the sake of clarity, we will call the first developed DEVS formalism Classical-DEVS [ZPK00] from now on.

2.3.1 Classical-DEVS

Classical-DEVS, originally DEVS, is the formalism proposed by Zeigler in the 70s. This formalism has been implemented in multiple languages and platforms. Some implementation examples include CD++[Wai09], DEVS++ [Hwa07], DEVSJava [ZS03], James II [HU07b], pyDEVS [BV02], and SmallDEVS [JK06].

In Classical-DEVS, the modeling hierarchy has two kinds of components: atomic models and coupled models. The atomic models are defined as a tuple: $A = \langle S, X, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$ where:
S is the set of states,
X is the set of input ports and values,
Y is the set of output ports and values,
$\delta_{int} : S \to S$ is the internal transition function,
$Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$ is the total state set (where e is the time elapsed since last transition),
$\delta_{ext} : Q \times X \to S$ is the external transition function,
$\lambda : S \to Y$ is the output function, and
$ta : S \to \mathbb{R}^+$ is the time-advance function.

An atomic model, also known as a basic model, is always in a specific state waiting to complete the lifespan delay returned by the ta function, unless an input of a new external event occurs. If no external event is received during the lifespan delay, the output function λ is called first, and then the state is changed according to the value returned by the δ_{int} function. If an external event is received, then the state is changed according to the value returned by the δ_{ext} function, but no output is generated.

For instance, a pedestrian traffic light with a call-button for crossing could be modeled as follows:
$S = \{red, white\} \times \mathbb{R}^+$
$X = \{\textit{button-pressed}\}$
$Y = \{\textit{red-light}, \textit{white-light}\}$

$\delta_{int}(\langle white, . \rangle) = \langle red, 10 \rangle$
$\delta_{int}(\langle red, . \rangle) = \langle white, 5 \rangle$
$\delta_{ext}(\langle white, t \rangle, e, \textit{button-pressed}) = \langle white, t - e \rangle$
$\delta_{ext}(\langle red, . \rangle, e, \textit{button-pressed}) = \langle red, 1 \rangle$
$\lambda(\langle white, t \rangle) = \textit{red-light}$
$\lambda(\langle red, t \rangle) = \textit{white-light}$
$ta(\langle ., t \rangle) = t$

Here, the state has two components, the first for keeping record of which light is on, and the second for keeping track of how long it will take until the next change of light happens. I.e. the state $\langle red, 1 \rangle$ means in a second from now, the light will switch to white.

The only input is the result of pressing the pedestrian call-button; this button is in charge of rescheduling the next white light. There is two possible outputs defined, *red-light* meaning that light is being switched to red, and *white-light* meaning it is switched to white.

The internal transition function (δ_{int}) switches the state when the waiting time expires, when the light is switched, a constant is assigned to the time component of the state. When interpreting the internal transition the time component of the state is not considered for deciding the new state. When result is independent from a variable, we mark it with a point. The external transition function (δ_{ext}) schedules the switch to white to happen in next second if the light is currently red. In case the button is pressed while the light is white, nothing is rescheduled. Depending the color of current state, the time component would be important. In the case current state is white, we preserve the scheduling of the next switch by subtracting the elapsed time to the time registered in the state. In the case current state being red, we maintain the color and change the time component to one. The output function (λ), executed right before the internal transition function signals the light being registered as new state by the internal transition function. I.e. if current state is red, output function result is white-light, which matches the new state after the internal transition function is executed. The time-advance function (ta), used for signaling how long to wait for next transition is always returning the time value of the current state.

Coupled models define a network structure in which nodes are any Classical-DEVS models (coupled or basic) and oriented links represent the routing of events between outputs and inputs or to/from upper level.

Formally, a Coupled Model is represented by the tuple $C = \langle X, Y, D, M, C_{xx}, C_{yx}, C_{yy}, SELECT \rangle$ where:
X is the set of input events,
Y is the set of output events,
D is an index for the Classical-DEVS component models of the coupled model,
$M = \{M_d | d \in D\}$ is a tuple of Classical-DEVS models as previously defined,
$C_{xx} \subseteq X \times \cup_{i \in D}(X_i)$ is the set of external input couplings;
$C_{yx} \subseteq \cup_{i \in D}(Y_i) \times \cup i \in D(X_i)$ is the set of internal couplings;
$C_{yy} : \cup i \in D(Yi) \times Y$ is the external output coupling function;

$SELECT : 2^D \setminus \emptyset \to D$ is the tie-breaker function that sets priority in case of simultaneous events.

Alternatively, in the variant of DEVS with ports (were each model has multiple input and output ports), the coupling relations are defined using EIC, EOC, and IC notation. EIC, EOC, and IC are respectively the External Input Coupling, External Output Couplings and Internal Couplings that explicit the connections and port associations respectively from external inputs to internal inputs, from internal outputs to external outputs, and from internal outputs to internal inputs,

In some cases, models producing zero time-advances in their transitions could reach an infinite loop and stale the simulation. For this reason, a legitimacy rule was added to the formalism; the legitimacy rule for DEVS [ZPK00] states models must never produce infinite zero-time-advances in finite segments of its timeline to be considered legitimate.

The formalism also provides abstract algorithms of an abstract simulator for these models, which define the semantics of the simulation. In Algorithms 2.1 and 2.2, we show the algorithms for simulator and coordinator as defined in [ZPK+76]. The simulator is in charge of simulating atomic models, while the coordinator simulates the coupled models.

Data: *parent*, A
`// simulator variables:`
parent `// parent coordinator`
t_{last} `// time of last event`
t_{next} `// time of next event`
A `// the simulated atomic model`
s `// the current state of` A

When receive *init-message(Time t)* **do**
  $t_{last} := t$
  $t_{next} := t_{last} + A.ta(s)$
done

When receive **-message(Time t)* **do**
  if $t \neq t_{next}$ **then**
    error: bad synchronization
  end
  $y := A.\lambda(s)$
  send y-message(y, t) to parent
  $s := A.\delta_{int}(s)$
  $t_{last} := t$
  $t_{next} := t_{last} + A.ta(s)$
done

When receive *x-message(X x, Time t)* **do**
  if $\neg(t_{last} \leq t \leq t_{next})$ **then**
    error: bad synchronization
  end
  $s := A.\delta_{ext}(s, t, t - t_{last}, x)$
  $t_{last} := t$
  $t_{next} := t_{last} + A.ta(s)$
done

Algorithm 2.1: Classical-DEVS simulator

Notice that time is never exchanged directly between models, but only between the models and the

Data: *parent*, N
// coordinator Variables:
parent// parent coordinator
t_{last}// time of last event
t_{next}// time of next event
N // the simulated Coupled model

When receive *init-message(Time t)* **do**
  foreach $i \in D$ **do**
    send init-message(t) to child i
    $t_{last} := max\{t_{last}[i] : i \in D\}$
    $t_{next} := min\{t_{next}[i] : i \in D\}$
  end
done

When receive **-message(Time t)* **do**
  if $t \neq t_{next}$ **then**
    error: bad synchronization
  end
  $i' = N.Select(\{i \in D : t_{next}[i] = t_{next}\})$
  send *-message(t) to i'
  $t_{last} := max\{t_{last}[i] : i \in D\}$
  $t_{next} := min\{t_{next}[i] : i \in D\}$
done

When receive *x-message(X x, Time t)* **do**
  if $\neg(t_{last} \leq t \leq t_{next})$ **then**
    error: bad synchronization
  end
  foreach $(x, x_i) \in C_{xx}$ **do**
    send x-message(x_i, t) to child i
    $t_{last} := max\{t_{last}[i] : i \in D\}$
    $t_{next} := min\{t_{next}[i] : i \in D\}$
  end
done

When receive *y-message(Y y_i, Time t)* **do**
  foreach $(y_i, x_i) \in C_{yx}$ **do**
    send x-message(x_i, t) to child i
    if $C_{yy} \neq \emptyset$ **then**
      send y-message($C_{yy}(y_i)$, t) to *parent*
      $t_{last} := max\{t_{last}[i] : i \in D\}$
      $t_{next} := min\{t_{next}[i] : i \in D\}$
    end
  end
done

Algorithm 2.2: Classical-DEVS coordinator

simulation engine. Simultaneous events correspond to the cases in which the discretization that results from the discrete-event modeling produces identical time values. Simultaneous events occur in DEVS in two ways: either a model receives input events from multiple models at the same time, or it receives events from other models at the same time as it is reaching the end of the time-advance delay. In this case, the tie-breaker function named $SELECT$ is used to decide what model is simulated first.

The simulation starts from a main loop (root-coordinator), which drives the whole simulation by repeatedly sending **-messages* to the topmost coordinator.

2.3.2 Parallel DEVS (PDEVS)

The Parallel-DEVS (PDEVS) [CZ94, CZK94] goal is to remove problems of serial computation caused by the $SELECT$ function under the occurrence of simultaneous events in Classical-DEVS. To achieve this, atomic models receive bags of messages; all events in the bag are considered simultaneous. The external transition function needs to process all events in the bag at once rather than one at a time as in Classical-DEVS. In addition, a new function is included in atomic models for handling cases where internal transitions and external transitions are simultaneous. This new function is called Confluence function and has to deal with bags of external events.

An atomic model in PDEVS is specified as a tuple $M = \langle X, Y, S, \delta_{ext}, \delta_{int}, \delta_{conf}, \lambda, ta \rangle$ where:
S is the set of sequential states,
X is the set of input events,
Y is the set of output events,
δ_{ext} is the external transition function,
δ_{int} is the internal transition function,
δ_{conf} is the confluent function,
λ is the output function, and
ta is the time-advance function.

In this specification, the confluent transition function ($\delta_{conf}(s, e, x)$) computes the next state using the current state s, the elapsed time e and the input events x when the internal and external events occur simultaneously. The rest of the PDEVS specification follows the Classical-DEVS specification.

A coupled model in PDEVS is specified [CZ94] as $DN = \{X, Y, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}$ where:
X is the set of input events,
Y is the set of output events,
D is the set of the component names,
M_i is the DEVS system of component name $i \in D$,
I_i is the influencers of M_i for each $i \in D$ and
$Z_{i,j}$ defines the M_i-to-M_j output translation for each i, j in I_i, and from/to upper levels.

The whole DN specification follows the Classical-DEVS Coupled specification except for the $SELECT$ function, which is removed.

In Algorithms 2.3, 2.4, and 2.5, we show the abstract algorithms for the PDEVS simulators [CZK94]. This simulator mainly uses three types of messages: internal state transition message (*, x_{count}, t), output/input message (@, *content*, t) and ending message (done, t_{next}). (*, x_{count}, t) and (done, t_{next}) messages are for state transition synchronization. A (@, *content*, t) message conveys the content of an output event to its parent coordinator, which then routes the messages to appropriate influences.

The simulation starts from a main loop (root coordinator in Algorithm 2.3), which drives the whole simulation by repeatedly sending (*, 0, t) to the topmost coordinator and waiting for a done message to advance the global simulation clock to t_{next}.

In Algorithm 2.4, we show the algorithms for PDEVS Coordinators. Here, when an internal message (*, x_{count}, t) is received, it is forwarded as (*, i_{count}, t) to all components of the coupled model members of the imminent set (IMM) or receivers (INF).

Note that x_{count} originally contains the number of influencers of the coupled model, and i_{count} will save the total number of influencers of a component defined inside a coordinator. $semaphore_{count}$ is increased every time a *-message is sent to an inferior PDEVS component. The ending message will only be sent when $semaphore_{count}$ is 0. Here, any output/input message is forwarded according to the coupling relations $Z_{i,j}$ to other simulators and coordinators. In addition, when an ending message is received, t_{next} (next internal event time) is saved in an event list and the internal variable $semaphore_{count}$ is decreased.

```
// Root-coordinator Variables:
t // current time
t := t_next of the topmost coordinator
repeat
    send(@, t) to the coordinator of the topmost coupled model
    wait until (done, t_next) is received
    send(*, t) to the coordinator of the topmost coupled model
    wait until (done, t_next) is received
    t := t_next of the coordinator of the topmost coupled model
until t = ∞;
exit // simulation complete
```

Algorithm 2.3: PDEVS root-coordinator

In Algorithm 2.5, we show the algorithms for the PDEVS abstract simulator. Here, when a (*, x_{count}, t) message is received, it indicates an event, internal or external, has to be processed.

If the simulation time reaches t_{next}(time of the next internal event), compute the output function and send the output events to the parent coordinator. If $semaphore_{count}$ is 0, only the internal transition takes place, otherwise, both internal and external functions take place at the same time leading to the confluent function to be executed once $semaphore_{count}$ is 0. After this, the t_{next} is calculated and an ending message is sent to the parent coordinator. Here, any content of an input message (@, *content*, t) is saved in a vector x_b (event bag) and the value of $semaphore_{count}$ decreased by one. $semaphore_{count}$ will be equal to 0 when all input events at the simulation time t have been saved in the event bag x_b.

These algorithms are known as Chow's simulation algorithms. An alternative set of algorithms known

```
// Coordinator Variables:
parent// parent coordinator
t_last// time of last event
t_next// time of next event
C// the coordinated coupled model
child_set// set of child simulators/coordinators
synchronize_set bag// messages to be processed
When receive @-message(Time t) do
    if t = t_next then
        t_last := t
        forall the imminent child processors i with minimum t_next do
            send(@, t) to child i
            cache i in the synchronize_set
        end
        wait until (done, t) is received form all imminent processors
        send (done, t) to the parent
    end
    raise an error
done
When receive y-message(Y y, Time t) from child i do
    forall the all influences j of child i do
        q := C.z_{i,j}(y)
        send(q, t) to child j
        cache j in the synchronize_set
    end
    wait until all (done, t)'s are received from j
    if self ∈ I_i then //y needs to be transmitted upwards
        y := C.z_{i,self}(y)
        send(y,t) to parent
    end
done
When receive q-message(Event q, Time t) from parent do
    lock the bag
    add event q to the bag
    unlock the bag
done
When receive *-message(Time t do
    if t_last ≤ t ≤ t_next then
        forall the receivers, j ∈ I_self ∧ all q ∈ bag do
            q := C.z_{self,j}(q)
            send(q, t) to j
            cache j in the sinchronize_set
        end
        wait until all (done, t_next)'s are received
        t_last := t
        t_next := min{t_next[i] : i ∈ D}
        clear the syncronize_set
        send(done, t) to parent
    end
    raise an error
done
```

Algorithm 2.4: PDEVS coordinator

```
// Simulator Variables:
parent// parent coordinator
t_last// time of last event
t_next// time of next event
A // the simulated atomic model
s // current state of A
bag // messages to be processed

When receive @-message(Time t) do
    if t = t_next then
        y := A.λ(s)
        send(y, t) to the parent
        send(done, t) to the parent
    end
    raise error
done

When receive q-message(Event q, Time t) do
    lock the bag
    add event q to the bag
    unlock the bag
    send(done, t) to the parent
done

When receive *-message(Time t) do
    if t_last ≤ t < t_next ∧ bag ≠ ∅ then
        e := t − t_last
        s := A.δ_ext(s, e, bag)
        empty bag
        t_last := t
        t_next := A.ta(s)
    end
    if t = t_next ∧ bag = ∅ then
        s := A.δ_int(s)
        t_last := t
        t_next = t_last + A.ta(s)
    end
    if t = t_next ∧ bag ≠ ∅ then
        s := A.δ_conf(s, bag)
        empty bag
        t_last := t
        t_next := t_last + A.ta(s)
    end
    if t > t_next ∨ t < t_last then
        raise error
    end
    send(done, t_next) to parent
done
```

Algorithm 2.5: PDEVS simulator

as Zeigler's simulation algorithms are defined for PDEVS [HU06]. In the context of our work, we always refer to Chow's algorithms presented in Algorithms 2.4, 2.3, 2.5.

2.3.3 Simulating by closure

Several simulators have implemented PDEVS. In particular, in aDEVS [Nut03], Nutaro takes a different approach [MN05] using the closure under coupling property of DEVS models, which states that for every coupled model it exists an equivalent atomic model [ZPK00].

Each coupled model (also referred to as network model in Nutaro proposed architecture) are reduced to an equivalent atomic model called the resultant. The resultant of a coupled model is an atomic model in which the set of states, transition functions and output functions are defined by its interconnected components. Exploiting this closure property, the resultant transformation produces a single atomic model, and hence eliminates the necessity of coordinators.

2.3.4 Other DEVS extensions

Several extensions have been proposed to DEVS. These extensions include, Fuzzy DEVS [ZPK00] providing a fuzzy logic to the state definition, Symbolic DEVS (Sym-DEVS) [ZC92] using symbolic algebra to represent time, Rational Time Advance DEVS (RTA-DEVS) [SW10, Saa12] using interval arithmetic to operate timelines, and others. In this section, we describe those most relevant for our work.

Sym-DEVS [ZC92, ZPK00] was developed in early 90s; it extends the DEVS formalism to define time as linear polynomials in place of real numbers. This formalism can be used to study the fault conditions and other properties when doing model verification. Its abstract simulator examines all possible strict choices of imminent forking execution when needed.

RTA-DEVS [SW10, Saa12] is another proposed extension to DEVS formalism. In RTA-DEVS, time is defined as intervals with rational borders. The goal of this formalism is to allow only the specification of models that can be automatically verified using model checking methods. To achieve this goal, the set of specifiable models is reduced to those that never have irrational time-advances.

Schedule Preserving DEVS (SP-DEVS) [HC04] and Finite & Deterministic DEVS (FD-DEVS) [HZ06] only allow the modeling of a strict subclass of DEVS models. The restriction is applied for researching state reachability.

The FD-DEVS subclass is restricted to those models having a finite set of states and scheduling transitions expressed by rational time-advances. In addition to the restrictions imposed by FD-DEVS, the SP-DEVS subclass is restricted to the models never changing scheduled transition times when receiving exogenous events.

The Real-Time DEVS (RT-DEVS) [HSKP97] was proposed for specifying simulation under real-time constraints, for example for simulation with human in the loop for training. In this formalism, actions are introduced, which have to be completed into time-windows. If an action is not reproduced before

the time-window is expired, it is discarded and the simulation goes on, like if it had never happened.

Finally, in Quantized-State Systems (QSS) [KJ01], a method is proposed for simulating approximations of continuous systems using DEVS.

2.3.5 The DEVStone benchmark

DEVStone [GW05, WGGA11] is a synthetic benchmark devoted to automate the evaluation of DEVS-based simulators, and it can be adapted to other DES engines. It generates a suite of models of different sizes, complexities and behaviors similar to diverse applications that exist in the real world.

DEVStone was created to study and compare the efficiency of DEVS simulators, compare different versions of a specific simulation engine, and aid the measurement and improvement of different DEVS-based software. The method proposes a theoretical time computed from the topology generated and the time required to run every transition, each executing Dhrystones [Wei84]. The execution time of the DEVStone is compared to the theoretical expected time to evaluate the overhead introduced by the simulator.

The DEVStone model generator permits one focusing on essential aspects that impact performance namely the size of the model and the workload done in the transition functions. The following parameters are used to generate a model: type (structure and interconnections between the model components), depth (number of levels in the modeling hierarchy), width (number of components in each immediate coupled model), internal transition time (execution time spent by the internal transition functions) and external transition time (execution time taken by external transition functions).

Four types of models (LI, HI, HO and HOMod) with different internal and external structure can be used:

- LI: Models with a low level of interconnections for each coupled model. Each coupled component has only one input and one output port. The input port is connected to each component but only one component produced an output through the output port.
- HI: Models with a high level of input couplings. HI Models have the same number of atomic components with more interconnections: each atomic component (a) connects its output port to the input port of the (a + 1)th component.
- HO and HOmod [WGGA11]: Models with high level of coupling and numerous outputs. HO models have two input and two output ports at each level while HOmod have a second set of (width - 1) models where each one of the atomic components triggers the entire first set of (width - 1) atomic models.

Because the model structure and the time spent in transition functions, which consume CPU clocks by running Dhrystones [Wei84] is known, the model execution time can be easily computed. Detailed computation formulas can be found in [GW05, WGGA11].

Several simulators have been compared using DEVStone. In [GAW08], aDEVS outperformed CD++

by a significant margin for large models. Several other PDEVS simulators (JAMES II, VLE, PyDEVS, and DEVS-Ruby) have been compared to aDEVS using DEVStone as well. The most recent work is by Franceschini et al. [FBT+14] where aDEVS remains the reference with regard to performance. This has been closely followed by a recent reimplementation of PyPDEVS [VTV14].

The most common use of DEVStone is for comparing different simulators; nevertheless, it can be also used to evaluate proposed improvements to a simulator. An example was the use of DEVStone for evaluating an improvement proposed by Kim et al. in [KKSS00]. To remove performance inefficiency, flattening the model has been proposed by Kim et al.. Using this approach, the model structure is modified so that the new model has no intermediary coupled models. This approach is different from the one used in aDEVS to produce a closure resultant, it rewrites the coupling to obtain a single coupled model. The impact of flattening has also been measured using DEVStone. In [GW05], and results show a clear improvement when using the flattening approach compared to the hierarchical approach.

2.3.6 Simulators with branching

Simulators not implementing DEVS have used execution branch or fork to solve some problems in the past. Most notorious examples are in the group of Logical-Process based simulators. In [PM07] the Moose simulator is presented, this simulator does not have tie-breaking functions as DEVS. When simultaneous events are received the execution is branched to simulate the simultaneous events in all possible permutations. Each branch of simulation may lead to different results.

In [HF97, HF01] branching (or cloning) is proposed as a mechanism to advance interactive simulations at points where user input is "undecided". In [HF02], optimizations to exploit repetition of information between cloned simulations and lazy-cloning ideas are presented in the context of Parallel Logical Processes simulation.

2.4 Measurement Uncertainty

One of many usages of DES is being part of a decision-making process for industrial and research works. Here, data is collected from the real system using measuring instruments and processes. From them, a set of measurement results is obtained and used as input to feed the simulation.

Metrology is the science of measurements and its applications. In this thesis, we adhere to the metrological vocabulary and practices proposed by the Bureau International de Poids et Mesures (BIPM), an international organization whose goal is to produce globally adopted metrological standards. BIPM is responsible for the standardization of the meter, gram, second and other international standardized units, for the provision of procedures and practices for properly measure in industrial and scientific works, and for the publication of other metrology related articles to advance the state of the art in the discipline.

For the metrology concepts and terminology, we follow Guide to expression of Uncertainty in Mea-

surement [BIP08] and International Vocabulary of Metrology [BIII08] documents provided by BIPM.

Mostly, we only refer to the following terms:

- Measurand [BIII08]: the quantity to be measured.
- True value [BIII08]: a theoretical value of the measurement assuming perfect accuracy, in the practice this value is unknowable.
- Measurement Result [BIII08]: set of quantity values being attributed to a measurand together with any other relevant information, generally expressed as a measurement value and a measurement uncertainty.
- Measurement Uncertainty [BIII08, BIP08]: non-negative parameter characterizing the dispersion of the quantity values being attributed to a measurand, based on the information used.

Metrology [BIII08] states it is not possible to determine true values from measuring magnitudes. For this reason, measurement results are provided with uncertainty quantifications, or error margins. The measurement result with uncertainty quantification is usually represented as an uncertainty interval for an acceptable level of confidence.

2.4.1 Uncertainty on DES

In the area of Parallel DES, uncertainty was used for speeding up simulations. On models not containing uncertainty quantifications, simulator assigns them and uses them for speeding up the simulation. Some authors have chosen to introduce uncertainty intervals for every time point in the model [LF00, LF04a, LF04b]. Any point in the intervals is considered equal for simulation purposes and selecting which one to use for running the events is based on parallelizing the execution of largest possible quantity of events.

For this purpose, authors refer to the concepts of Approximate Time and Approximate Time Event Ordering [Fuj99, LF00]. In Approximate Time Event Ordering, the common ordering used is Approximate Time Causal, which is defined for Events being concurrent if any point in their uncertainty intervals overlaps. For concurrent events, a second ordering is used, the causality relation "before than" [Lam78].

These approaches state that there is a relation between the simulation results accuracy and the uncertainty introduced, but they do not provide tools to bound the errors, quantify the results uncertainty, or express qualitative information about the validity of the obtained result.

The general approach is based on the Approximated Time presented in [Fuj99]. A case study is presented in [LF04a] in the context of interactive simulation; Architectural details for process-oriented distributed simulation are also provided in [LF04b].

In [LF00] an alternative approach is described, where after assigning uncertainty to the values, a precise value is chosen from a pre-sampled set of random numbers for each event. This allows using pre-existing federated simulation tools as HLA without requiring modifications while speeding up simulations. The paper also define how to constraint the uncertainty-look-ahead relation to obtain better speed-ups.

Other authors proposed to introduce uncertainty on spatial properties of the model for obtaining speed-ups [GCQ08, QB04]. Here, models are expected to have positional behavior, for example cell-phones and cell-towers for unwired communication. This approach can be combined to obtain further speed-ups with the Approximated Time approach previously mentioned.

Finally, in [BNO03], an architecture extending the Time Warp algorithm with Temporal Uncertainty is proposed. This architecture exploits uncertainty for avoiding rollbacks when the events to be rolled-back could be moved into their uncertainty interval to make them still useful.

Chapter 3

Precise time representation

As discussed in earlier chapters, Discrete-Event Simulation (DES) is a technique in which the simulation engine plays a history following a chronology of events. The technique is called "discrete-event" because the processing of each Event of the chronology takes place at discrete points of the timeline. The Events in the chronology are not required to align to any clock; they could be placed freely over a continuous timeline, usually represented by $\mathbb{R}^+$.

When implementing computer simulators, time datatypes and their operations are needed. In the most general case, time variables may require representing arbitrary numbers in $\mathbb{R}^+$. The common approach is either by choosing a standard fixed length approximated datatypes such as floating-point (FP), or by choosing a datatype native to the programming language being used, as fixed-point or rational.

The fixed length approximation datatypes and techniques are widely adopted for studying Continuous Systems. On them, approximated results are obtained when using approximated input values and approximated operations. Furthermore, the errors introduced by approximating can be bounded and propagated following well-established concepts and practices borrowed from Calculus and Numerical Methods.

Using min-max in a restricted domain, an interval of possible results can be obtained by bounding the error of the simulation. To obtain the min-max, derivatives and other Calculus concepts can be used.

In the case of DES, approximating an input can adversely affect the behavior of the model, making its trajectory diverge from that point forward. This is because there is a chance that, for example, an approximation causes the order of two events in the timeline to permute. We stated earlier that DES states are products of their histories; a change in their histories potentially leads to different events.

In this chapter, we provide a classification for the issues found when using datatypes approximating values for representing time in DES. We survey what datatypes are popular in DES simulators for representing time and we discuss the strengths and weaknesses of each of them.

3.1 Approximated representation problems

In the most generic DES formalisms, the models' domain of the time variables is $\mathbb{R}^+$. This results in a quantization problem at the time of implementing the model in computers. Depending on the datatype chosen for implementation, different approximations or operation restrictions may be observed.

We classify the approximation effects as belonging to one of three categories: time shifting, event reordering, and Zeno problems.

3.1.1 Time shifting

The most common errors in DES timelines are the time shifting errors [VDW14]. These errors are direct consequence of approximating the time values, reproducing events in the simulator slightly earlier or later than formally defined, but always maintaining the partial order of the events in the chronology. Time shifting errors usually have a minor impact or no impact at all on the simulation execution.

For instance, in the case of normalized half-precision FPs (16 bits) described by the IEEE-754 standard, we have a mantissa of 10 bits (plus an implicit bit always in 1), an exponent of 5 bits, and a bit for deciding the sign of the number. A number is read as: $sign \cdot 2^{exp-15} \cdot mantissa$. The diagram of Figure 3.1 we show examples of possible values represented as half-precision FPs.

$$\underbrace{0}_{sign}\ \underbrace{10000}_{exponent}\ \underbrace{0000000000}_{mantissa} = 1 \times 2^{16-15} \times 1.0000000000_{bin} = 2$$
$$\underbrace{0}_{sign}\ \underbrace{10000}_{exponent}\ \underbrace{0000000010}_{mantissa} = 1 \times 2^{16-15} \times 1.0000000010_{bin} = 2.0039062$$
$$\underbrace{0}_{sign}\ \underbrace{10000}_{exponent}\ \underbrace{1111111111}_{mantissa} = 1 \times 2^{16-15} \times 1.1111111111_{bin} = 3.9980469$$

Figure 3.1: Half-precision floating-point representation examples: 2, 2.0039062, and 3.9980469

Adding the last two examples of Figure 3.1 (2.0039062 + 3.9980469) should result 6.0019531. This is not the case, because accurately representing 6.0019531 requires a mantissa with more than 10 bits. The obtained result is 6, represented as shown by the diagram in Figure 3.2.

$$\underbrace{0}_{sign}\ \underbrace{10001}_{exponent}\ \underbrace{1000000000}_{mantissa} = 1 \times 2^{17-15} \times 1.1000000000_{bin} = 6$$

Figure 3.2: Half-precision floating-point addition result example

In several formalisms where only the logical order of events is considered to decide the next state (i.e. Petri Nets), this is not a problem. Nevertheless, this is a problem for more generic formalisms, i.e. DEVS or PDEVS, where it is legal to use the time elapsed since last event to define the new state, and time shifting errors may be enough to make the resulting trajectories diverge.

3.1.2 Event reordering

In the case of event reordering [VDW14], the approximation leads to a different order of events in the timeline than the one expected using exact arithmetic on real numbers. Here, the list of events on the timeline is permuted and the partial order of events is erroneous. In some cases the causality chain breaks, and the resulting trajectory diverges arbitrarily from the expected one, since the following events in the chain are not necessarily related to those actually expected. Sometimes, an event reordering error is the result of accumulating multiple time shifting errors. An example of this error is formally explained in Section 3.1.4.

3.1.3 Zeno problem

Finally, the distance between two events in the timeline can be defined as any real number, particularly those very close to zero. If the values represented are small enough, when added to the current time, they will not produce any change. This behavior can be reproduced indefinitely making the system stale. Systems producing this behavior were studied in concurrent systems; this is called the Zeno problem [Lee14, MP92].

In some simulation formalisms, such as DEVS, legitimate models never reproduce Zeno behavior [ZPK00]. Once models legitimacy is formally proven, the placement of Events in the timeline is assumed exact.

A model theoretically safe in regard to Zeno conditions may still be unsafe in practice; a simulator using fixed-size approximated datatypes recreates the conditions for it. For example, with half-precision FP, adding 1 to 4096 results in 4096, but the correct arithmetic result is 4097. Here, the result is approximated because 4097 is not representable in this datatype. If this addition was to represent the time-advance of the simulation, we could reach a Zeno condition when simulating.

The diagrams in Figure 3.3 show the representation of 4096 and 1 for reference.

$$\underbrace{0}_{sign}\ \underbrace{11011}_{exponent}\ \underbrace{0000000000}_{mantissa} = 1 \times 2^{27-15} \times 1.000000000_{bin} = 2^{12} = 4096$$
$$\underbrace{0}_{sign}\ \underbrace{01111}_{exponent}\ \underbrace{0000000000}_{mantissa} = 1 \times 2^{15-15} \times 1.000000000_{bin} = 1$$

Figure 3.3: Half-precision floating-point representation examples: 1 and 4096

For example, Zeno problem can easily be reproduced in simulators using a FP variable for representing a global clock, a common design choice in many implementations; in this case, if we simulate a metronome (a device repeatedly ticking after fixed time intervals), after enough simulation time, the accumulation in the variable renders the subsequent additions irrelevant (being too small for affecting the mantissa).

3.1.4 Problems with timing errors

The errors introduced in Sections 3.1.1, 3.1.2, and 3.1.3 are difficult to explicit in formal proofs. In formal proofs, the computation model is not included, and approximation errors, as time-shift, are related to the computation model and datatypes selected for implementing simulators.

Furthermore, reordering errors and Zeno problem errors are practically impossible to predict. Moreover, they may occur in irregular frequency patterns, which, in the worst case, will pass undetected through the validation tests. This was the case of the popular simulators NS-3, and OMNeT++. [Lac10] presents a case reporting getting different trajectories when using different processors to run the same simulation, the reason being the differences in the FP arithmetic implementation of each processor, including following different standards or differ in the rounding policies implemented. Having different implementations produce different approximations of the values when operating. Similar case is presented in [VH08] about OMNeT++. Here, authors state, "Well-known precision problems with floating-point calculations however, have caused problems in simulations from time to time." Neither NS-3 authors, nor OMNeT++ authors investigates further the problem. They both propose a solution based in integer representation. In the case of NS-3, a 128bits integer is used, while for OMNeT++ an integer of 64bits plus a scale factor is proposed.

The major risk when these problems and errors appear in the simulation execution is that they cause a break of the causality chain. In DES, the state of the simulation can be thought as a function of its history, producing a causal relation between them. When an event in the history is approximated, the evolution of the simulation may diverge from the expected one, and produce an incorrect sequence of states. When this happens, we say that the causality chain was broken.

Current simulator implementations are generally silent about these errors, usually because it is impossible to detect them properly using their Time datatypes. The FP datatypes native in most programming languages do not include any reporting of errors. For example, Java does not have any reporting mechanism for notifying rounding, overflow, or anything else.

3.2 Time representation in existing DES simulators

To start understanding the problems of time representations in current simulators, we reviewed the code of the various open source simulators implementing Discrete-Event System Specification (DEVS) or Parallel-DEVS (PDEVS) formalisms.

- aDEVS [Nut03]: "A Discrete EVent System simulator" is developed at ORNL in C++ since 2001;
- CD++ [Wai09, LW04]: "Cell-DEVS++" is developed at Carleton University, in C++, since 1998;
- DEVSJava [ZS03]: "DEVSJava" is developed at ACIMS in Java since 1997; it is based on its predecessor DEVS++ (implemented in C++). Currently it is part of the DEVS-Suite project [HZ08];
- Galatea [SQK11]: A bigger project with an internal component (Glider) developed at Universidad de Los Andes implementing DEVS in Java since 2000;

- James II [HU07b] "JAva-based Multipurpose Environment for Simulation" is developed at University of Rostock, in Java, since 2003. It is the successor of the simulation system JAMES (Java-based Agent Modeling Environment for Simulation). The project includes a set of Simulators, we reviewed the DEVS one;

- ODEVSPP [Hwa09]: "Open DEVS in C++" is developed in C++, since 2007;

- pyDEVS [BV02]: "DEVS for Python" is developed at Mc Gill University in Python since 2002;

- SmallDEVS [JK06]: "SmallDEVS" is developed at Brno University of Technology in SELF/SmallTalk since 2003.

- JDEVS [FB02, FB04]: "JDEVS" was developed by J. B. Filippi at Universita di Corsica.

A common feature in all of them is that they embed the passive state by using some kind of representation for the infinity value. All simulators, but aDEVS and CD++, represent Time using the double-precision FP datatype provided by the programming language used by the simulator. In the case of pyDEVS, where the variables have dynamic types, the use of FP is forced by systematically using Python casting syntax in each operation.

All simulators but one use the internal representation of infinity provided by the datatype. The exception is ODEVSPP that uses MAX_DOUBLE constant in place of infinity.

In the case of aDEVS, the time is represented using a C++ class template. The class works as a type wrapper adding representation of infinity to the provided type. The operators are defined in a way that, if one of the parameters is an infinity the defined algorithms are used, else the wrapped type operators are used.

In the case of CD++, the time is represented using a class with five attributes: four integers for the hours, minutes, seconds and milliseconds, and a double-precision FP for sub-milliseconds timings. There is not an explicit representation of infinity, but the passive state is evaluated comparing the Hour field to 32767.

In addition, we surveyed two DES simulators not implementing simulators defined by formalisms, but following DES ideas, mostly from Logical Processes methodologies.

The NS-3 implementation was initially developed as a joint effort between University of Washington and INRIA. It is released as a C++ library since 2007; as part of the effort, the NS-3 consortium was founded for providing project guidance and support.

The OMNeT++ implementation was initially designed for network simulation; current implementation extends it for simulating more generic DES models. András Varga started the project in 2007; nowadays, OpenSim ltd is currently maintaining it.

These simulators reproduced approximation errors in the past; their case is mentioned in Section 3.1.4; the solutions proposed, and implemented, for these simulators was using integer values for representing time. We will go over the limitations of this approach in Section 3.3.2.

The NS-3 simulator represents the time with 128bits integers; the integers represent the magnitude while the unit is assumed nanosecond for every time value.

The OMNeT++ simulator represents the time with a pair of integers, the first (of 64 bits) represents the quantity, and the second represents the resolution unit, being the options from seconds to atto-seconds.

All the simulators above mentioned use a hierarchical approach. Regardless of the simulator, a global variable is needed to keep track of the global Simulation Time. In some simulators, the Future Events List (FEL) is shared among every model in the simulation; in others, the FEL is local to each model. Someone may think that the second approach has no global time, but the hierarchical approach used in DEVS results in the root model to have an event at every effective step of the simulation.

The following example, modeled in DEVS, shows that even a simple model may be subject to re-ordering errors when using FP time variables. In this example, we model a counter with two inputs, one to increase the count and the other to reset and output the current counter result.

The top coupled model is defined as the tuple $C = \langle X, Y, D, M, I, EIC, IC, EOC, SELECT \rangle$ where:
$X = \emptyset$,
$Y \subseteq \mathbb{N}$,
$D = \{1, 2, 3\}$ with: M_1 and M_2 two generators sending tick respectively every 0.1 second and every 1 second, and M_3 a counter with 2 input ports: inc and reset,
$EIC = \emptyset$,
$IC = \{\langle\langle M1, out\rangle, \langle M3, inc\rangle\rangle, \langle\langle M2, out\rangle, \langle M3, reset\rangle\rangle\}$,
$EOC = \{\langle\langle M3, out\rangle, \langle self, out\rangle\rangle\}$,
$SELECT = \langle M1, M2, M3\rangle$

In Figure 3.4 we show a diagram of the coupled model.

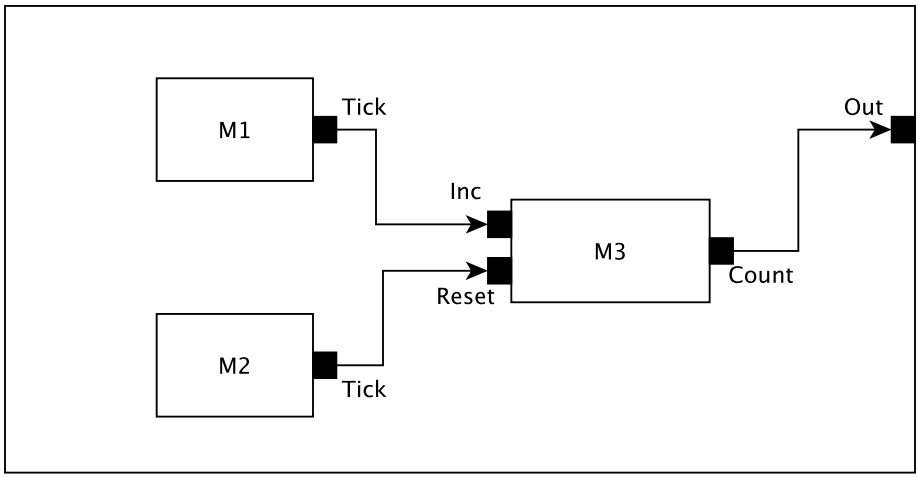

Figure 3.4: An example model subject to reordering errors when representing time using floating-point

The output of this simulation is, in theory, the value 10 on the out port of C model every second. This value corresponds to how many ticks were received from M1 model between ticks of the M2 model. In Figure 3.5, we show a plot of expected output trajectories of each model.

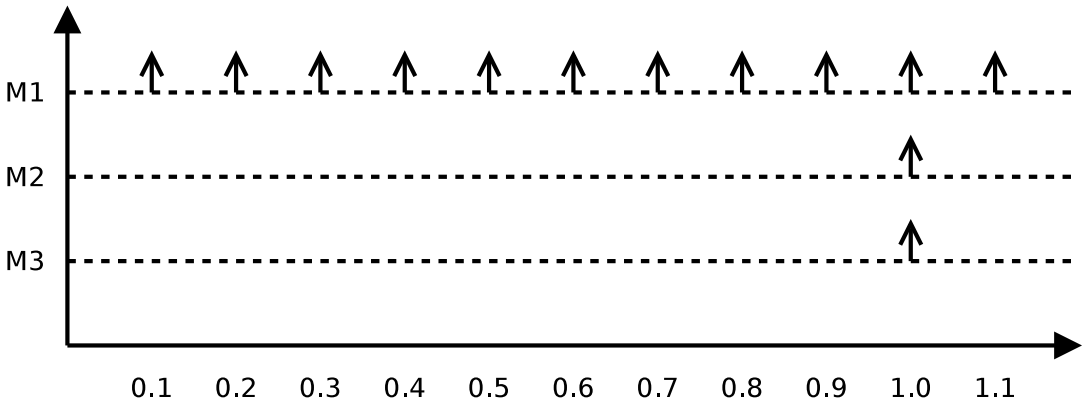

Figure 3.5: Expected output trajectories of the example model

We implemented this in DEVSJAVA 3.1 and pyDEVS 1.1, and kept it running. In DEVSJava, we got a different result than expected: the output consisted of a majority of 10 values, but we also obtained some outputs with values 9 and 11.

In pyDEVS, we first obtained the results expected for this experiment. Nevertheless, after changing the parameter of M2 from 1.0 to 100.0 seconds, we observed similar discrepancies, with a few occurrences of 999 and 1001 values on the out port of C instead of only 1000 as theoretically expected. Interestingly, the errors of this model occur following a regular pattern (a geometric law of factor 4 in both simulators). This result can be explained knowing that both simulators use single- or double-precision FP numbers for time representation. Indeed, the value 0.1 is a well-known bad quantization point in the FP standard. Therefore, every time an algorithm accumulates many times the value 0.1 (as the model, M1 does) the rounding error accumulates, which leads to the errors we observed.

3.3 Analysis of classic time datatypes

In DES, the commonly used datatypes for representing time include: floating-point (FP), integers, fixed-point, rational, and intervals between rational numbers.

These datatypes are conveniently implemented in processors, or they are easily implementable combining native ones. Nevertheless, these datatypes have limitations in their usage that need to be noticed. Not taking in account these limitations may produce incorrect simulation results.

The following is a more detailed explanation of the most important limitations of the above mentioned datatypes.

3.3.1 Floating-point datatypes

Adopted by a majority of simulators, this datatype leads to several variants due to the selection of different precision levels, and to how the passive state of a model is represented. For example, infinity can be mapped onto a reserved value, or an additional variable (e.g., a Boolean) can be used to handle the special case of infinity, in a structure or wrapper datatype.

The floating-point (FP) datatype was engineered to represent an approximated real number and to support a wide range of values. The basic structure is the use of a fixed length mantissa and a fixed length exponent.

The main strengths of using FP are:

- A compact representation, usually between 32 and 128 bits;
- Implemented in almost every processor;
- A large spectrum between max and min representable numbers;
- An internal representation for infinities
- Existence of widely used standards make the simulator code more portable;
- The mechanics of its arithmetic approximations has been studied in detail.

On the other hand, FP has well known limitations [Gol91]:

- Rounding errors: operations (including assignment) may round the values;
- Cancellation issues;
- Portability: the same operation may be implemented differently in different CPUs or languages, providing different results, rounding modes, and flags/exceptions;
- Associativity: FP arithmetic is not associative, but usually used as if it was;
- Unexpected results for special cases: e.g. in IEEE-754, an addition overflow will result in an infinite value.

These limitations are often considered an acceptable trade-off in some areas, due to their intuitive use and properties guaranteeing the errors of the result obtained could be bounded enough to produce the required answers.

We classified the problems affecting DES in five groups: quantization error, non-fixed steps, cancellation issues, compiler optimization problems and standard complexity.

Quantization error

FP arithmetic rounds the results continuously, not only with divisions (as it happen with integers), but also with additions and subtractions. Rounding errors result in two kinds of errors: shifted values, and

artificially coincidental values.

We do not elaborate more on the case of shifted values, as this case was already discussed in Section 3.1.1 when describing time shifting errors. We shown how this kind of error could break the causality chain in formalisms as DEVS where it is possible to link states to elapsed time since last event. Likewise, on Section 3.2 we presented an example of a model where the event reordering errors happen when using FP variables to represent time, in some cases by accumulation of small approximations.

The rounding of values due to quantization may also produce artificially coincidental values.

As mentioned in Chapter 2, the case of simultaneous events must be dealt with care, which explains the need for a SELECT function in Classical-DEVS and the use of Bags of events and confluence function in PDEVS. Unfortunately, when the coincidence is the result of a hidden process, these functions may not be able to make a proper decision. In the case of Classical-DEVS, without the rounding error, two distinct time values have an implicit natural order, and this natural order may not be the one produced by the SELECT function call after they became simultaneous.

Non-fixed step

The FP representation uses a variable step size between consecutive represented values. The reasons for having variable step come from the nature of the incremental sequence of powers. As value of the exponent increases, higher is the number multiplying the mantissa. Thus, the distance between two numbers with consecutive mantissa values is larger.

This means that having two numbers accurately represented does not guarantee they will produce an accurate result when added. It depends on the granularity of representable values in the number line defined by FP at the point the result is expected, which is controlled by the exponent field.

The following is an example of how this affects a DES simulation. We define a model whose behavior is to send a message every second. A half-precision FP is used to represent time. Initially, the current time is set to zero, after incrementing it by one 2048 times, it reaches the value 2048, which is larger than any odd number representable in this datatype. In Figure 3.6 we show the diagram of the next step addition (2048 + 1).

$$A = \underbrace{0}_{sign}\ \underbrace{11010}_{exponent}\ \underbrace{0000000000}_{mantissa} = 1 \times 2^{26-15} \times 1.000000000_{bin} = 2^{11} = 2048$$
$$B = \underbrace{0}_{sign}\ \underbrace{01111}_{exponent}\ \underbrace{0000000000}_{mantissa} = 1 \times 2^{15-15} \times 1.000000000_{bin} = 1$$
$$C = A + B$$
$$C = \underbrace{0}_{sign}\ \underbrace{11010}_{exponent}\ \underbrace{000000000?}_{mantissa} = 1 \times 2^{26-15} \times 1.00000000?_{bin} = 2048\ or\ 2050$$

Figure 3.6: Example of half-precision floating-point addition affecting DES resulting trajectories

Here, the result of adding 1 to 2048 could result in 2050, or 2048, depending on what is the value of the last bit of the result's mantissa. The value of the last bit in the mantissa depends on the rounding policy implemented. In both cases, result being 2048 or 2050, a time shifting error is introduced. In

addition, in the case of result being 2048, the rounding policy is making impossible to ever increment the current-time variable again (with additions of 1); this is introducing also a Zeno problem.

A common work around for these problems is incrementing the size of the FP datatype used. When simulating the same model using single-precision FP to represent time, we will obtain the expected behavior during a longer period. After enough iteration, the difference between the exponent of the simulated time and the increment will exceed the size of the mantissa. Here, the increment will fall outside the resultant's mantissa and will need to be rounded. The diagram in Figure 3.7 shows how the same scenario can be reached using single-precision FP.

$$A = \underbrace{0}_{sign}\,\underbrace{10010110}_{exponent}\,\underbrace{00000000000000000000000}_{mantissa} = 2^{150-127} = 2^{23}$$
$$B = \underbrace{0}_{sign}\,\underbrace{01111111}_{exponent}\,\underbrace{00000000000000000000000}_{mantissa} = 2^{127-127} = 1$$
$$C = A + B$$
$$C = \underbrace{0}_{sign}\,\underbrace{10010110}_{exponent}\,\underbrace{00000000000000000000000}_{mantissa} = 2^{150-127} = 2^{23}$$

Figure 3.7: Example of single-precision floating-point increment affecting DES resulting trajectories

In single-precision, FP number uses 23 bits to represent the mantissa. If we add 1 to 1×2^{23}, the result is again 1×2^{23} as shown in the diagram in Figure 3.7. Depending on implementation details, such an increment can be rounded up or truncated. Same as when operating with half-precision FP, both rounding policies produce time shifting errors, and in case it is truncated, it will reproduce the Zeno problem. This shows that incrementing the size of the FP mantissa may delay the manifestation of these problems in some contexts, but it does not solve them in a definitive way.

Some FP implementations - for instance those following the IEEE754 standard [C+08] - allow sub-normal numbers. In this case, numbers that are too close to zero are treated differently: the exponent is set to a fixed value and the significant digits in their mantissa are reduced. When operating with these numbers, analogous problems to those described before still appear, but their details are more complex to follow.

Cancellation issues

This kind of problem happens when subtracting two close numbers. The effect is that the mantissa loses significant bits. For instance, in Figure 3.8, we show the diagram of the subtraction of two close numbers. In the example, the exponent of both numbers is the same, and the first three bits of the mantissa match (including the implicit one). Once the subtraction result is obtained the three first bits of the mantissa are canceled, and the result has to be normalized shifting the mantissa three bits to match the first 1 in the result with the implicit 1 of the normalized representation. The result of the subtraction, after normalization has only 7 significant bits after the point, and the last 3 bits (marked as ?) are filled accordingly to the rounding policy.

The quantity of significant bits lost during the operation (marked as ? in the example) are not reported to the user. On the contrary, these bits are silently filled with 0s or 1s depending on rounding

$$A = \underbrace{0}_{sign}\ \underbrace{10000}_{exponent}\ \underbrace{1111000001}_{mantissa} = 1 \times 2^{16-15} \times 1.111100001_{bin} = 3.876953125$$
$$A = \underbrace{0}_{sign}\ \underbrace{10000}_{exponent}\ \underbrace{1101000001}_{mantissa} = 1 \times 2^{16-15} \times 1.110100001_{bin} = 3.626953125$$
$$C = A - B$$
$$C = \underbrace{0}_{sign}\ \underbrace{01101}_{exponent}\ \underbrace{0000000???}_{mantissa} = 1 \times 2^{13-15} \times 1.000000???_{bin} = 0.25$$

Figure 3.8: Example of half-precision floating-point increment affecting DES resulting trajectories

policy implemented. The bits introduced as filling are representing values smaller than any bit in the subtraction parameters. This increment of precision in the results has the effect of not setting the flag of inexact result in following operations that should be considered inexact otherwise.

In DES, most simulators only use addition and comparison operations; the addition is used for advancing the current time, and the comparisons are used for deciding which is the next event to process. When using FP to track the current time, the granularity of time is changed every time the exponent is incremented. To mitigate the problem of having variable granularity, some simulators implement a framing mechanism. The framing mechanism uses a variable (sometimes implicit) storing the lowest time value in the system, and every other value is relative to this one. This variable works as a threshold, when the minimal value in the system is incremented, the threshold variable is incremented, and all other values are adjusted (subtracting the time incremented to the threshold variable). Here, the goal is having all variables operating as close as possible to the origin, or other exponent having the best-fit granularity for the model being simulated. A downside of using this approach is that it increases the number of subtractions, which in turn lead to an increase in cancellation errors.

If a non-native implementation of FP is used, error detection can be implemented by keeping track in every operation of how the mantissa significance was affected. This does not solve the problem, but may still give a confidence indicator about the results.

Compiler optimization dangers

In [Gol91], the author shows a set of optimizations that, when used with FP, can affect the precision of the computation. Changing compilers (for updating or portability) can change the results of these optimizations, which might be hard or impossible to detect.

Likewise, at the time of distributing the simulation in a heterogeneous cloud or grid, different architectures, operating systems and compilers may be used, making the problems scale together with the infrastructure.

Standard complexity

If the floating implementation follows the IEEE754 [C+08], which is common, there are more issues to consider. IEEE754 gives special meaning to some quantities (Not-a-Number (NaN), infinities, positive

and negative zeros); it defines exceptions, flags, trap handlers and the option to choose a rounding model. All these mechanisms need to be taken into account when implementing the Simulator, or they could result in unexpected behaviors.

3.3.2 Integer datatypes

Using integers is the first idea that comes up when trying to avoid the issues related to rounding and precision with FP numbers. For example, in the field of networking, well-known simulators such as NS-3 and OMNeT++ have gone through major rewriting in order to change their representation from FP to integer: NS-3 has chosen a 128 bits integer representation [Lac10], while OMNeT++ v4.x has chosen a 64 bits integer representation [VH08].

The integer datatype is used to represent a subset of consecutive numbers in $\mathbb{Z}$.

Some interesting characteristics of the integer datatype are:

- Its compact representation, usually between 8 and 128 bits;
- Its generalized support by all processors, making it a fast datatype;
- It provides exact arithmetic results for every operation except division, in the case a division is rounded, the modulo operation can be used to detect and measure the inaccuracy.
- It has a fixed step between any two consecutive values.

Using integers in place of FP can be seen as a trade-off in which the large range and the ability to approximate the (dense) real numbers is traded against the accuracy of an exact representation. In many situations, when the range is not that much a concern, integers offer a better trade-off.

In the case of integers, our concerns are not related to approximations breaking the causality chain, but to the application limitations imposed by their range, the quantity of unused values, and the composition with different time scales.

Range

For a given number of bits, the absolute range of values supported by an integer datatype using that amount of bits is much narrower than the absolute range of a FP using the same amount of bits.

Starting with single-precision FP (32-bits), the largest absolute value is around 3×10^{38} while it is only around 2×10^{9} for a 32-bit integer. At the other end, the smallest single-precision FP value above zero is around 10^{-38} while for any positive integer the smallest (absolute) value above zero is one.

Trying to cover the same range as a FP with integers, using the smallest step offered by the FP, is feasible but very space-inefficient for single-precision (76 bits vs. 32) and unreasonable for higher precision.

However, since there is no approximation with the additive arithmetic on integers, a frame shifting mechanism as described in Section 3.3.1 can be exploited. It is safe to implement it because it will not incur problems such as the cancellation mentioned in FP.

Quantity of unused representation

Since the smallest representable positive integer value is one, the smallest representable time value, using integers, is the unit associated to the variable. That is, if we have an integer variable representing seconds, the smallest representable time would be 1 second; if we have an integer variable representing nanoseconds, the smallest representable time would be 1 nanosecond.

Some simulators use a fixed unit value, e.g., nanoseconds. This fixed unit implies that all timing is a multiple of one nanosecond. Therefore, a model of a generator that would output ticks every second would waste 99.9% of an already limited representation range.

To avoid this waste, an analysis of the model can help to adjust the unit. In this case, the unit does not need to be a standard one: a rational, e.g., $\frac{1}{7}s$, can even be chosen if it allows capturing all the possible timing needed for the simulation of the model timeline. However, this causes problems with the model definition in terms of maintainability and software design.

Multiple scales and model composition

One of the strengths of the models based in systems theory, as in DEVS, is its ability to compose models. This allows reusing previously developed models to create more complex ones, which allows dividing the studied system into smaller parts.

If the time unit is hard-coded within the Simulator implementation, we need a unit to be found that covers all possible models. This approach can be used for simulators dedicated to a specific purpose, e.g., micro-controllers electronics or planet dynamics. However, if the simulator is intended to be multi-purpose, it becomes impossible to find a compact and efficient representation that covers the full spectrum of possible requirements. An alternative is to use time units defined locally within each model. For instance, this approach allows picking months as the time unit for building a gravitational model of planets dynamics, and picoseconds for micro-controllers as suggested by each model analysis.

When we have different time scales in different (Sub-)models, we need to have extra computation to compose them as DEVS Coupled Models. In case the time-step in one representation is divisible by the one in the other, the smallest step can be used as the common step. This requires the time-values in the larger-step model have to be systematically adjusted, at the cost of an extra multiplication operation.

When using time scales that do not divide each other, e.g., $\frac{1}{7}s$ and $\frac{1}{5}s$, finding a common time-step requires finding a common denominator. In this case, systematic multiplications are required on both numbers participating.

Adding multiplications is not just a performance issue: it also requires increasing the size of the

integers used to ensure that values that were valid in the original models are still representable in the composed ones.

Fixed-point representations, which can be seen as integers with a scale factor, present similar issues to the integer datatypes.

3.3.3 Rational datatypes

Even when rational numbers are not a native datatype implemented in processors, they are common in several programming languages, or in libraries. The common implementation of rational is to define two integer variables as numerator and denominator and provide the basic arithmetic and comparison operators.

Two classic variants exist for the implementation: the first one consists in doing a simplification after every operation, and the second is never to simplify. In the first, the same number always has the same internal representation, but we need to compute the Greatest Common Divisor (GCD). In the second case, the performance is not affected by computing the GCD, but the representation space may be exhausted faster. The first option is slow because simplification is a complex process. The second option increases the representation space used in almost every operation failing to exploit the representation space properly. There are optimized versions that delay simplifications until needed in order to keep the values inside the representable ranges of both numerator and denominator as long as possible. In any case, both representations must deal with redundant values that have not useful meaning in the representation (i.e., $\frac{1}{2}$ and $\frac{2}{4}$).

In summary, rational is not limited by periodicity, which is a well supported feature. Rational main limitations are the space required, and the complexity. Complexity mostly comes from the need for simplifications, and dealing with ambiguity.

We still consider this datatype an approximation because of irrational numbers. They still need being approximated to the closest rational representable number.

3.3.4 Structures and objects

Complex datatypes that come with each programming language can also be used in place of the aforementioned ones, in order to avoid some problems we have surveyed. These datatypes include classes, objects, structures, tuples, arrays, vectors and other containers. However, using such datatypes often results in trade-offs between user-friendliness, memory consumption, performance, arithmetic complexity and they do not necessary solve the problems we stated before.

In some implementations of time, a structure with an integer field to handle different units of a time value is used. This can be done using an array, vector, class or structures, with the main objective of expanding the range provided by a single native variable. For example, this kind of structure is used in the Time class in CD++ in which the fields are aligned on standard time sub-units (days, hours,

minutes, nanoseconds).

The use of human-friendly units, even when convenient for the model definition, makes the arithmetic more complicated, and is space-inefficient. For example, the use of a three byte-long integers to represent hours, minutes, and seconds on 8 bits each wastes 76% of the values for minutes, and seconds. Here, the 8 bits used for representing the magnitude associated to each unit can store 256 different values, while only 60 are required for minutes, and seconds, after 60 the value is carried to the hours in the case of minutes, and to the minutes in the case of seconds. If we add one more integer to represent days, 90% of the values in the byte assigned to hours is wasted, because the carry from hours to days happens at 24. Using fewer bits to reduce the space overhead is not convenient in modern processors, because of memory alignment. Even when reducing where possible, human-readable values are not powers of two; then, we will always require some bit patterns being unused. The best possible adjustment would be assigning 6 bits for minutes and seconds and using 60 of the 64 possible values, and 5 bits for hours using 24 of 32 possible values. In addition, the use of byte-long variables is not efficient in some CPUs where registers are longer than 8 bits, for example most C++ compilers for i386 architectures, assign 32 bits to small integer variables for the sake of keeping them aligned.

Integer and FP numbers can both use 16, 32, 64 or 128 bits depending on the processor architecture. A structure having multiple integers as previously defined, when implemented as a Class, Object or Structure, requires a pointer to the structure and offsets to each component. This is worse if we also use a human friendly representation. For efficiency purposes, it is advisable to use integers that match the processor register size (i.e., 32 or 64 bits on current architectures). Therefore, a structure for [HH:MM:SS] would use 224 bits and represent almost same range than a single 64 bits integer coding the same range using the second unit. In some languages, the unnecessary use of space can be reduced choosing an 8 bit integer datatype, but this is not a possibility in every language.

In the case a vector-type collection is used, the memory footprint is increased to define pointers needed for iteration or other general-purpose use of the datatype as memory allocators.

In some architecture with few available registers, accessing multiple integers generates a higher number of cache miss and results in significant performance degradation.

In addition, the use of smaller than bus size or misaligned datatypes will require extra processing to read and write. For example, if the bus width is 32-bits, two consecutive integer values of 16 bits can be read using a single memory access, but extra operations are needed for the actual values to be unpacked prior operation.

Similarly, having a 16 bits integer followed by a 32-bits one ends up either with the second one being misaligned and requiring two memory access each time it is read/write, or with the waste of 16 bits to realign the representation.

If we want the datatype to be reusable, the operations giving the semantic to the datatype and its representation need to be encapsulated. The common datatypes providing the mechanisms to embed operations are classes and objects, and in most languages, one depends on the other. In case the datatype is implemented as a class or object, adding the operators to it, does not increase the memory access to its internal fields.

3.3.5 Rational Intervals

Using rational intervals, as in Rational Time Advance DEVS (RTA-DEVS) [Saa12], requires operating with two rational numbers for each operation [Abe01]. This requires at least double the space than with rational for representing each "boxed" number and double the time to process every operation. The point of using intervals is providing a basic support for irrational numbers representing each irrational number as a small interval containing it; i.e., we can represent Euler's constant using the interval [2.7, 2.8]. The main idea is thinking about the interval as an error around the number being represented. A limitation of the approach is that it is impossible to compare numbers for equality and, when intervals intersect, it is not even possible to order them. Operating with these numbers usually increment the error associated at every step, incrementing also the chance of making them non-comparable.

3.3.6 Symbolic Algebra

Symbolic Algebra had being proposed as an alternative approach representing real numbers including irrationals. In symbolic algebra, the numbers are represented by expressions defining them, the operations are defined as composition rules of these expressions, and solvers are implemented to evaluate comparisons when needed.

This approach looks promising in the sense of accuracy, but as far as we know, Symbolic DEVS (Sym-DEVS) [ZC92, ZPK00] was the only formalism using it. Here, the expressions were limited to represent roots of polynomials. The methods for solving polynomials have high complexity, and there is no representation for transcendental numbers using them.

Symbolic Algebra is also implemented in several Mathematical applications and libraries, but restrictions apply to them. In these tools, when symbolical manipulation is unknown for an expression, they operate using a fixed length digits expansion of the number.

It was proven in Computable Calculus that comparisons of arbitrary expressions are non-decidable [Abe01], and using fixed length approximations is not a valid solution. For example, in a one-digit expansion, π and 3 are considered equal, while we know they are different.

Anyway, this approach seems to be the most promising one for including irrationals in the timeline; using expression manipulation techniques allows, in some cases, to detect relations between two expressions without the need for expanding any digit. The equality problem can then be reduced for some well know expressions providing a way to avoid falling into non-decidable operations.

An example to show how this method is effective is the implementation of the comparisons of square roots of rational numbers. In fixed length or intervals it is not possible to compare for equality the square root of 2 against the square root of 2.00000000001 because they may be considered equal after a fixed length rounding or they may produce overlapping intervals when using interval arithmetic. On contrary, it is easy to compare them using Symbolic Algebra manipulations, for example by comparing the two radicands.

3.4 Summary

In this chapter, we described the problems related to use approximated representation representing Time variables when implementing Discrete-Event Simulators, and classified it in three categories: time shifting, event reordering, and Zeno problems.

We discussed the datatypes used in eleven implementations, and their limitations. In addition, we discussed other datatypes commonly available in programming languages, and alternative approaches proposed in the literature.

In Chapter 4, we propose three datatypes for dealing with the described problems and concerns, Safe-Float (SF), Rational-Scaled floating-point (RSFP), and Multi-Base floating-point (MBFP). In addition, in Chapter 5, we propose a method using concepts and properties from Computable Calculus [Abe01] for extending Time datatypes for supporting the representation of subsets of computable irrational numbers.

Chapter 4

Datatypes for precise time in DES

In this chapter, we present three approaches for dealing with Time representation problems in Discrete-Event Simulation (DES). First, we present Safe-Float (SF), a datatype for detecting and notifying floating-point (FP) errors. We can use this datatype for evaluating the relevance of the topic. Second, we present two datatypes mixing ideas from FP and rational. We propose using these datatypes as replacement in current simulators implementations. In Chapter 5, we describe a method for adding support to subsets of computable irrationals to these datatypes. We end this chapter showing a performance comparison of the use of these datatypes using the DEVStone benchmark.

4.1 Safe-Float

The main goal of Safe-Float (SF) is detecting the problems described in Chapter 3 with floating-point (FP). If no problem is detected, we can be certain that the simulation has the correct trajectory. Else, the user can rerun the simulation using another datatype (for example, a larger size SF, or other datatype, as those proposed later in this chapter).

SF is designed as a drop-in replacement of the FP datatypes. The datatype is inspired from the safe-numerics library [Ram12] from Robert Ramey providing safe integer for C++14. SF was proposed to the Boost Library. The development of the project was accepted under the scope of Google Summer of Code 2015 while mentored by Robert Ramey.

C++11 features have influenced the design, but it can be ported to other languages with similar language features. The main point of using C++11, and not a previous standard of C++ is the access to Floating Point Unit hardware flags, which was not available before. We can assert about errors produced in an operation with low performance overhead using this environment. The alternative requires intensive bit-manipulation to check for the errors.

4.1.1 Context of Safe-Float

In Section 3.3.1, we described the strengths and limitations of using FP datatypes for representing time. Being unaware of errors while operating with these datatypes affects the correct simulation of DES models.

In addition to the arithmetic problems, some others can result in security concerns. In these regards, the CERT Division of Software Engineering Institute at Carnegie Mellon University publishes guidelines for Secure Programming when using FP datatypes [CER08a].

Using C++ techniques like overloading, template meta-programming, and custom literals, we propose a wrapping datatype for FPs. This datatype behaves exactly like the wrapped one, except that no expression produces arithmetic errors. In case we cannot obtain a correct result, we report it to the user. Errors are detected at compile time when possible, and at runtime otherwise.

4.1.2 Secure coding with Floating-Point

We present here the problems mentioned in the CERT C and C++ secure coding standards [CER08a, CER08b] that we could detect using C++11 language features. For each of them we present a code example of the failure and how our new datatype would handle it.

Domain, Pole, and Range errors

Three errors may occur when operating with C's math library functions: Domain, Pole, or Range errors. The first, domain error, is when the input value is not in the domain of the function. The second, pole error, is when a result is represented as infinity because is approximating to the limit of the function. The last one, range error, is when the result does not fit in the datatype used.

The danger here is that even if the functions are not returning the expected value, a value is returned. Unless the user explicitly checks for combinations of flags and *errno* variable values, the user is unaware of the error. Then, the execution continues using the erroneous result.

In Listing 4.1, and 4.2, we show examples of domain error when using the inverse hyperbolic tangent function from the C math library. In Listing 4.1, single-precision FP is used, and the value of 2.0 is passed to the function producing a silent error. In Listing 4.2, SF of equivalent precision is used throwing an exception because 2.0 is not in the domain of the inverse hyperbolic tangent function. The examples includes an assert line showing where the execution never reaches using each datatype.

```
try {
        float unsafe_var = 2.0;
        float unsafe_result;
        unsafe_result = atanh(unsafe_var);
        cout << "no exception was thrown" << endl;
} catch (exception) {
```

```
        assert(false); // never reached
}
```

Listing 4.1: Example of continue execution unaware of a domain error

```
try {
        safe<float> safe_var = 2.0;
        safe<float> safe_result;
        safe_result = atanh(safe_var);
        assert(false); // never reached
} catch (exception) {
        cout << "exception was thrown" << endl;
}
```

Listing 4.2: Example of detecting a domain error with Safe-Float

We show similar example for the case of the Pole errors evaluating the inverse hyperbolic tangent in its border (1.0). In Listing 4.3, we see how using single-precision FP returns an infinite value, corresponding with the limit of the function, and silently continues execution. In Listing 4.4, we show how using $safe\langle float\rangle$ throws an exception for notifying the operation is undefined at that point.

```
try {
        float unsafe_var = 1.0;
        float unsafe_result;
        unsafe_result = atanh(unsafe_var);
        cout << "no exception was thrown" << endl;
} catch (exception) {
        assert(false); // never reached
}
```

Listing 4.3: Example of continue execution unaware of a pole error

```
try {
        safe<float> safe_var = 1.0;
        safe<float> safe_result;
        safe_result = atanh(safe_var);
        assert(false); // never reached
} catch (exception) {
        cout << "exception was thrown" << endl;
}
```

Listing 4.4: Example of detecting a pole error with Safe-Float

Finally, we show an example of Range errors. In this example, we evaluate the inverse hyperbolic tangent in the largest representable value included in the domain of the function. The result of this operation is extremely large for fitting in the representation used. This error is different from Pole error in the sense that it may be recoverable using a higher precision datatype, while the Pole error is not

recoverable, because the value is not a valid result from the function. In Listing 4.5, we see how using single-precision FP continues operating after the wrong result was assigned. In Listing 4.6, we show how using $safe\langle float\rangle$ throws an exception for notifying the operation is undefined at that point.

```
try {
        float unsafe_var = nexttowardf(1, -INFINITY);
        float unsafe_result;
        unsafe_result = atanh(unsafe_var);
        cout << "no exception was thrown" << endl;
} catch (exception) {
        assert(false); // never reached
}
```

Listing 4.5: Example of continue execution unaware of a range error

```
try {
        safe<float> safe_var = nexttowardf(1, -INFINITY);
        safe<float> safe_result;
        safe_result = atanh(safe_var);
        assert(false); // never reached
} catch (exception) {
        cout << "exception was thrown" << endl;
}
```

Listing 4.6: Example of detecting a range error with Safe-Float

Domain, Pole, and Range errors are encoded as a combination of flags and special variables in C++, starting at 2011 standard. The CERT coding standard rule mandates that these variables need to be checked after each operation. In these regards, the SF checks each condition and throws an exception in case an error is detected. When possible, error recovery can be implemented in a catch statement. Otherwise, the exception stack reaches the top and aborts the program with a system error in place of silently producing incorrect results.

Approximation problems on casting

In the CERT secure coding standard, two new problems related to approximation are described aside from those mentioned in Section 3.3.1. The rules FLP34-C, and FLP36-C focus on the casting approximations. Here, errors are produced by approximation problems when casting between integer types, and FP types, or when casting between FP types of different sizes.

The following is an example of integer to FP casting with approximation problems. We cast an odd integer number larger than the FP mantissa. This kind of integer values cannot be exactly represented in the FP variable. Thus, it is silently rounded when assigned. Below, we show the same example using SF; in this case, the assignation throws an exception.

In Listing 4.7, we show an example of an odd integer number larger than the FP mantissa that is casted into a FP variable. In Listing 4.8, we show how same operation throws an exception when using SF.

```
try {
        int i = (2<<(std::numeric_limits<float>::digits+2))+1;
        float unsafe_var;
        unsafe_var = i;
        cout << "no exception was thrown" << endl;
} catch (std::exception) {
        assert(false); // never reached
}
```

Listing 4.7: Example of continue execution unaware of a int to float approximation

```
try {
        int i;
        i = (2<<(std::numeric_limits<float>::digits+2))+1;
        safe<float> safe_var;
        safe_var = i;
        assert(false); // never reached
} catch (std::exception) {
        cout << "exception was thrown" << endl;
}
```

Listing 4.8: Example of detecting a int to float approximation with Safe-Float

In the case of C++, assigning a large FP value to an integer variable produces undefined behavior. In Listing 4.9, we show the example of assigning the maximum representable number in single-precision FP to a 32-bits integer and continue the execution unaware of the error. In Listing 4.10, the same example is reproduced using SF, which throws an exception at assign time.

```
try {
        float unsafe_var = std::numeric_limits<float>::max();
        int i;
        i = unsafe_var;
        cout << "no exception was thrown" << endl;
} catch (std::exception) {
        assert(false); // never reached
}
```

Listing 4.9: Example of continue execution unaware of a float to int out-of-range assign

```
try {
        safe<float> safe_var = numeric_limits<float>::max();
        int i;
        i = safe_var;
```

```
        assert(false); // never reached
} catch (std::exception) {
        cout << "exception was thrown" << endl;
}
```

Listing 4.10: Example of detecting a float to int out-of-range assign with Safe-Float

Finally, we show the case of narrowing. Here, the variable is casted from a FP datatype to another FP datatype, the second being smaller than the first one, i.e. double to float in C++. When narrowing, it is possible that the original value does not fit in the new variable. In Listing 4.11, we assign the maximum value representable in double-precision FP to a single-precision one, and the execution continues. In Listing 4.12, we show how using SF an exception is thrown under the same scenario.

```
try {
        double unsafe_to_narrow = std::numeric_limits<double>::max();
        float unsafe_var;
        unsafe_var = unsafe_to_narrow;
        cout << "no exception was thrown" << endl;
} catch (exception) {
        assert(false); // never reached
}
```

Listing 4.11: Example of continue execution unaware of an incorrect narrowing

```
try {
        double unsafe_to_narrow = numeric_limits<double>::max();
        safe<float> safe_var;
        safe_var = unsafe_to_narrow;
        assert(false); // never reached
} catch (exception) {
        cout << "exception was thrown" << endl;
}
```

Listing 4.12: Example of detecting an incorrect narrowing with Safe-Float

This is not the only way to produce narrowing errors. For example, a number having all the bits of its mantissa set to 1 will have to truncate the mantissa to fit in a smaller representations. This is true even when the value being casted is smaller than the largest value representable in the target datatype.

Associative operation failures

In Section 3.3.1, we said FP numbers are not associative, a property commonly assumed available, given their intended use as representation for real numbers. In Listing 4.13, we show an example of adding and removing a number that produces a different result than the expected one. In Listing 4.14, we reproduce the same example using SF and show how it throws an exception for notifying the error.

```
try {
        float a, b, c, d;
        a = (8 << std::numeric_limits<float>::digits);
        b = -(8 << std::numeric_limits<float>::digits);
        c = 2;
        d = 2;
        c += a;
        c += b;
        a += b;
        d += a;
        if (d != c){
                cout << "no exception was thrown" << endl;
                cout << "c & d should be equal" << endl;
        }
} catch (exception) {
        assert(false); // never reached
}
```

Listing 4.13: Example of continue execution unaware of an unexpected approximation

```
try {
        safe<float> a, b, c, d;
        a = (8 << std::numeric_limits<float>::digits);
        b = -(8 << std::numeric_limits<float>::digits);
        c = 2;
        d = 2;
        c += a;
        c += b;
        a += b;
        d += a;
        assert(false); // never reached
} catch (exception) {
        cout << "exception was thrown" << endl;
}
```

Listing 4.14: Example of detecting an unexpected approximation with Safe-Float

In CERT secure coding standard, rules FLP01-C/CPP, and FLP02-C/CPP make reference to this issue. It is stated as errors produced by rearrange of operations giving different results or inexact operations. This may produce unexpected behaviors as rearranging events in a simulation state trajectory.

Here, the approach used by SF is to check that every operation produces an exact result (checking information available in the FP environment). This is a stronger requirement than associativity. There might be scenarios where this level of safety should be discarded, i.e. when using FP intended algorithms from numerical methods for matrices factorization. For this purpose, SF allows to silent particular checks

at variables declaration. We cover the details in Section 4.1.3 of how to enable/disable checks in SF when discussing design and implementation details.

Unexpected inputs

In C++, the standard way to read user input is the use of the iostream library; this library reads input from streams coming from different sources and extracts values from them, converting the variables where the values are assigned. In the case of FP variables, numbers are converted, but also some keywords are accepted for special values as NaN, and infinite. The rule FLP04-CPP from CERT secures coding standard states unexpected input converted into FP inputs must be checked. Our idea for dealing with these unexpected values is overloading the iostream functions for SF variables.

A similar problem is described in other rule in respect to scanf family of C functions. Here, we cannot apply the same solution since the scanf family of functions cannot be overloaded.

In Listing 4.15, we show an example reading unexpected input into a single-precision FP variable. In Listing 4.16, we show how SF throws an exception when an unexpected value is introduced.

```
try {
        stringstream ss;
        ss.str("NaN");
        float f;
        ss >> f;
        assert(isnan(f));
        cout << "no exception was thrown" << endl;
} catch (exception) {
        assert(false); // never reached
}
```

Listing 4.15: Example of continue execution unaware of a NaN introduced from stream

```
try {
        stringstream ss;
        ss.str("inf");
        safe<float> sf;
        ss >> sf;
        assert(false); // never reached
} catch (exception) {
        cout << "exception was thrown" << endl;
}
```

Listing 4.16: Example of detecting an infinite being read from a stream with Safe-Float

Undetected denormalization

We call *denormalization* when we obtain a subnormal result from an operation between normalized operands. Here, the exponent is set to a special value, and the mantissa implicit 1 is removed from the representation. Subnormal representation permits writing numbers as 0.00001 with the lowest exponent.

Depending on the CPU, compiler, and compiler options used, subnormal numbers may behave differently. For example, it is possible that they flush to zero creating underflow conditions. Another possibility is that the performance of operations is reduced for them producing bottlenecks. In some cases, subnormal flags are different on operations hiding some errors expected to be detected.

In most cases, it is safer and faster to avoid the use of subnormal values than dealing with them. For example, using a larger size FP datatype. In the case of SF, we decided to detect if any subnormal value is used in any operation and throw an exception when this happens.

In Listing 4.17, we show how subnormal numbers are silently used in FP operations. In Listing 4.18, we show how using SF throws an exception when a subnormal number is passed for an operation.

```
try {
        float f;
        f = numeric_limits<float>::denorm_min();
        assert(fpclassify( f ) == FP_SUBNORMAL );
        cout << "no exception was thrown" << endl;
} catch (exception) {
        assert(false); // never reached
}
```

Listing 4.17: Example of continue execution unaware of operating with denormal values

```
try {
        safe<float> sf;
        sf = numeric_limits<float>::denorm_min();
        assert(false); // never reached
} catch (exception) {
        cout << "exception was thrown" << endl;
}
```

Listing 4.18: Example of detecting the use of a denormal value with Safe-Float

Other Floating-Point concerns

Aside from the requirements motivated by the CERT secure coding standard, we address other concerns following the previously mentioned list of dangers from [Gol91], and we allow throwing any exception defined in C++'s FP environment [C$^+$11].

Some rules and recommendations in CERT's secure coding standard cannot be detected at the library

level. In those cases, intervention of a developer or external lint tool is required. The following are the rules and guidelines of CERT not covered by SF:

- FLP30-C. Do not use FP variables as loop counters.
- FLP37-C. Do not use object representations to compare FP values, i.e. using memcmp.
- FLP00-CPP. Understand the limitations of FP numbers.
- FLP04-C. Check FP inputs for exceptional values in scanf family of functions.
- FLP07-C. Cast the return value of a C function that returns a FP type.

4.1.3 Design and implementation details

The design of the library is based in the Generic Meta-programming policies architecture proposed in [Ale01]. In this architecture, multiple implementations of the same datatype are defined, and one is chosen at compile time. Each combination of template parameters, named policies, determines an implementation of the datatype. Each implementation could define different behaviors, optimizations, and characteristics.

For SF, we define four template parameters, the first being the datatype to be replaced, and the following three, the policies to be applied over it. The policies are named check policy, error-handling policy, and casting policy.

The check policy defines the validations to be used. For every operation, check policy class defines pairs of functions enforcing pre- and post-conditions. The policy chosen predicates about the implementation details of these conditions.

At runtime, each operator calls the pre- and post-conditions defined by the check policy, and in case one of them fails an error is notified. Since post-conditions could depend on the input values, pre-conditions can store values or set flags internally. Then, a post-condition can access those values for evaluating the post-condition. This is possible, because pre- and post-conditions are guaranteed to run in-order and without race conditions.

The check policy provides a method used for generating the message to notify about a check failure. This method can be overridden for customization.

The error handling policy defines how the user would be notified in case a check fails. A method, report-failure, is defined to be called with the message produced on a check failure. The report-failure method defines what to do with this message. I.e. the default implementation report-by-throw, will throw and exception with the received message.

The reason error-handling is being defined as a policy is that in some contexts exception throwing is not the best way to report problems. For example, in the context of critical-time systems, it is common to disable the exceptions stack; in the context of using the datatype for diagnostic purposes, it might be better to continue the run with no throwing, and report the check failures to a log file or console.

Property	Operation	Policy	Example case
Overflow	Addition	check_addition_overflow	max() + max()
Overflow	Subtraction	check_subtraction_overflow	lowest() - max()
Overflow	Multiplication	check_multiplication_overflow	max() * max()
Overflow	Division	check_division_overflow	max() / min()
Inexact Rounding	Addition	check_addition_inexact_rounding	lowest() + min()
Inexact Rounding	Subtraction	check_subtraction_inexact_rounding	max() - lowest()
Inexact Rounding	Multiplication	check_multiplication_inexact_rounding	(2*((2^digits)-1)) *((2^digits)-1)
Inexact Rounding	Division	check_division_inexact_rounding	1.0 / 3.0
Underflow	Addition	check_addition_underflow	2.2250738585072019e-308 + -2.2250738585072014e-308
Underflow	Subtraction	check_subtraction_underflow	2.2250738585072019e-308 - 2.2250738585072014e-308
Underflow	Multiplication	check_multiplication_underflow	min() * 0.5
Underflow	Division	check_division_underflow	min() / max()
Division by zero	Division	check_division_by_zero	1.0 / 0.0
Invalid Result (NAN)	Addition	check_addition_invalid_result	infinity() + (- infinity())
Invalid Result (NAN)	Subtraction	check_subtraction_invalid_result	infinity() - infinity()
Invalid Result (NAN)	Multiplication	check_multiplication_invalid_result	infinity() * 0.0
Invalid Result (NAN)	Division	check_division_invalid_result	infinity() / infinity()
Domain Errors	Math.h	check_math_domain	atanh(2.0)
Pole Errors	Math.h	check_math_pole	atanh(1.0)
Range Errors	Math.h	check_math_range	atanh(nexttowardf(1, -INFINITY))
Input Error Nan	Iostream	check_input_nan	stringstream ss(“nan”); ss >>sf;
Input Error Infinity	Iostream	check_input_inf	stringstream ss(“inf”); ss >>sf;

Table 4.1: Basic set of check policies implemented in Safe-Float

Some error-handling policies included in the library are: report-to-stream, that can be used for reporting to the standard output, the standard error, or a file; report-by-assert producing a crash of the system by using the standard assert command; and report-to-exception, which is the default.

Errors reported by the error handling policy are only those requiring runtime detection. Errors detected at compile time report the error and exit compilation, then they do not require any reporting mechanism.

The cast policy defines how SF plays along with other datatypes in the system. Here, we focus on the approximation concerns defined in CERT secure coding standard when casting or narrowing.

We also propose a way to ensure that the literals used in the source code are representable using user-defined literals. For example, 0.1 is not a representable value in float because of its periodicity on binary systems. The compiler approximates this value to a representable value at compile time. This prevents us from detecting the problem. To avoid this, we propose the _sf suffix for user-literals. When safe-literals is enabled, SF will only accept SFs literals for initialization. These literals could be evaluated at compile time to be representable valid values using a constant-expression.

In Table 4.1, we show a collection of classes currently usable as check policy in SF.

These classes focus on fine-grained conditions. For combining them, it is possible to define a new class

inheriting from those interested in. In some cases, conflicts have to be solved. These conflicts happen when more than one class combined is implementing the same pre- or post-condition. In these cases, when the method is called, the compiler will detect ambiguous dispatching code and fail compilation.

We provide a template class for automatizing the conflict resolution. This class, compose-check, uses variadic templates for re-implementing every pre- and post-condition iterating those in every parent class. In Listing 4.19, we show how compose-check implements the addition pre-condition, each other method is implemented similar to it.

```
template<class FP, template<class> class A, template<class> class... As>
class compose_check :
   public check_policy<FP>, private A<FP>, private As<FP>...
{
public:
        virtual bool pre_addition_check(const FP& lhs, const FP& rhs){
                using expand = int[];
                expand e = { true,
                    ( As<FP>::pre_addition_check(lhs, rhs) )... };
                return A<FP>::pre_addition_check(lhs, rhs)
                    && std::all_of(std::begin(e),
                                   std::end(e),
                                   [](bool i){ return i; });
        }
...
}
```

Listing 4.19: Compose-check implementation for Safe-Float check-policy combinations

Compose-check class receives template parameters for the datatype to be wrapped, and as many check classes as required. The abstract check policy class is inherited publicly to allow using the composed check as a check policy. Each of the check classes being combined is inherited privately for preventing exposure of conflicts.

In each condition method, an array of integers is created. This array is initialized using for each position a result calling the same method in each parent class. The compiler expands the definition and initialization of the array. The method ends returning as the result of the conjunction of evaluating the condition in every parent class.

In Table 4.2, we show a set of commonly used combined policies provided in the library.

For error-handling policy, the set of provided classes is shorter and simpler. Most of these classes trigger mechanisms jumping to other places in the process, i.e. exceptions, or producing termination conditions. For them, it does not make much sense to combine, especially in a sequential way. Then, we do not provide any automatized mean for it.

The provided classes are:

Property	Operation	Policy	Components
Overflow	All Arithmetic	check_overflow	check_addition_overflow check_subtraction_overflow check_multiplication_overflow check_division_overflow
Inexact Rounding	All Arithmetic	check_inexact_rounding	check_addition_inexact_rounding check_subtraction_inexact_rounding check_multiplication_inexact_rounding check_division_inexact_rounding
Underflow	All Arithmetic	check_underflow	check_addition_underflow check_subtraction_underflow check_multiplication_underflow check_division_underflow
Invalid Result	All Arithmetic	check_invalid_result	check_addition_nan check_subtraction_nan check_multiplication_nan check_division_nan
Bothflows	All arithmetic	check_bothflows	check_underflow check_overflow
Math	All math.h	check_math	check_domain check_pole check_range
IO	All iostream	check_io	check_input_nan check_input_inf
All	All arithmetic	check_all	checks every defined condition

Table 4.2: Composed check policies defined for convenience in Safe-Float

- on_fail_throw : Throws a SF exception when the checks fail.
- on_fail_abort : Calls std::abort() when the check fails.
- on_fail_assert : Inserts an assert for each condition, it makes safe only the debugging, these asserts are removed by the compiler when compiling in release mode.
- on_fail_log : This class logs each error into a stream that needs to be declared when initializing the class, after each error is sent into the stream, it silently continues its execution.
- on_fail_cerr : This is a particular case of on_fail_log. Here the log is output to standard error.

Extending the list of classes used for error-handling policy requires only defining a class implementing the *report-failure* method.

Finally, the cast policy is defined as an integral template parameter. The integral parameter is used as a bit-array for flags; each position of the array enables a particular casting. The way the cast is enabled is through the C++11 function enable-if as showed in Listing 4.20.

```
class safe_float <class FP, class CHECK, class ERROR_HANDLING, int CAST> {
        template<typename C = CAST,
                 typename enable_if( CAST & CAST_TO_INT )>
        operator int() { ... }
...
```

```
};
```

Listing 4.20: Example of enabling a cast method in Safe-Float

Here, we show how the operator for casting to integer is enabled when the CAST_TO_INT bit is set in the cast policy. Other cast policy flags include: the cast to wrapped datatype; cast from wrapped datatype; allowing only the initialization by safe literals; and check for narrowing. Some flags are not orthogonal, for example the one allowing only safe literals in the initialization has to block the cast from the wrapped type.

In addition to the classes defined, the math, limits and iostream libraries were specialized. For more details on options available in the SF library, please check the library reference in [Vic15].

4.1.4 Evaluation

The first simulator tested, CDboost, is a DEVS/PDEVS simulator implemented during the development of this thesis. It implements the sequential architecture of PDEVS presented in [VNWD15]. This simulator is part of an effort for proposing a simulation library to the Boost Project. A feature of this library is the ability to use any datatype for time providing addition, subtraction, and comparisons. For this purpose, time datatype is declared as template parameter in each model. At the time of coupling, models are checked for using same representation of time.

The other simulator used for testing was aDEVS 2.8.1. In this one, a template class for a wrapping Time datatypes is provided, but its use is not propagated in the code. Then, propagating the changes of datatype required inspection and manual replacement.

Since SF is a drop-in replacement of FP datatypes, we could replace the use of $float$, *double*, or *long double* by $safe\langle float\rangle$, $safe\langle$*double*$\rangle$, or $safe\langle$*long double*$\rangle$ just by replacing their use in the code.

For both simulators, models were selected from the examples provided in their distribution. And, all of them were run using the SF datatype for time operation.

Some models operated time variables using more operations than those used by the simulator. Then, we decided running the examples using the check-all policy.

The SF datatype proved useful as diagnostic tool. As it was expected, we found out models ending abnormally the execution because of uncaught SF exceptions. Examples of failing models include: the clock model provided with CDboost (mentioned in Section 3.2 as M1), and the checkout-line model provided with aDEVS 2.8.1.

We provide a comparison of the performance of using SF against other time datatypes in Section 4.3.

4.2 Rational-Scaled and Multi-Base Floating Points

In Sections 3.2 and 3.3, we described a set of problems and limitation with the representation and performance of the different datatypes used in current simulators' implementation, and in Section 3.1 we described how these problems may lead to errors in the generation of simulation trajectories of Discrete-Event Simulation (DES).

In this section, we are presenting more adequate datatypes for those simulators surveyed. These new datatypes are intended for models already implemented in those simulators; we do not discuss here how to represent irrational values, which is treated in Chapter 5. For this section we make some assumptions based in our goal to use these datatypes in the context of DEVS simulators, but the concepts can be easily applied to any other DES formalism.

The standard way to represent the measure of a time lapse is the product between its magnitude, a time unit, and a scale-factor for the unit: $t = m \cdot sf \cdot u$. For example, 3 nanoseconds have a magnitude of 3, nano (meaning 10^{-9}) as the scale-factor and second as the unit. Our proposed datatypes aims at finding how to better represent magnitude and scale-factor. When we use these datatypes for representing time, the unit is implicitly assumed to be Second, which is the unit proposed by Bureau International de Poids et Mesures (BIPM) for International Standardization of the Time Unit. Adhering to the international standard, and using the Second as unique unit in the system discards the complexity of defining conversion functions and calling them at runtime.

For proposing the new datatypes, we attempt to cover the following functional requirements derived from the analysis presented in Chapter 3:

- Support multiple scale-factors.
- Provide additive operation not leading to time shifting errors, event reordering errors, or Zeno problems.
- Provide comparison operators (=, <).
- Provide a large enough range of representation to include every value representable in FP and rational variables.
- Provide safe arithmetic such that any operation with unsafe result will produce an error (similar to Safe-Float (SF)).
- Support seamlessly the coupling of models allowing the possibility of operating using multiple different scale-factors in a single simulation.

In addition, we set the following non-functional requirements:

- Do not affect the performance of the simulation significantly.
- Keep a compact representation.
- Encourage optimizing the additive operations and comparisons over any other. These are the only operations used in the simulation algorithms defined for DEVS formalism.

- Delay operations deriving from the composition of models as much as possible. These operations are known to be expensive.

Based in these requirements, we propose two datatypes, Rational-Scaled floating-point (RSFP) and Multi-Base floating-point (MBFP), which are drop-in replacement of FPs and rational datatypes.

4.2.1 Rational-Scaled Floating Point

Rational-Scaled floating-point (RSFP) is composed by four integer variables, called $quantum_n$, $quantum_d$, *magnitude*, and *exponent*. They are combined to form a number as described in Formula 4.1.

$$magnitude \cdot \frac{quantum_n}{quantum_d} \cdot 2^{exponent}$$

Formula 4.1: Rational-Scaled floating-point (RSFP) interpretation

The unit, second, is implicit when using this datatype for time. The exponent and quantum variables take care of scaling the units when needed. We call this group of variables the scale-factor. For example, a time lapse of 3 nanoseconds can be expressed as showed in Formula 4.2.

$$3 \cdot \frac{1}{5^9} \cdot 2^{-9} \cdot s \Rightarrow magnitude = 3, scale\text{-}factor = 10^{-9}, unit = second$$

Formula 4.2: Example of representing 3 ns in RSFP

This new datatype provides a correct representation for all those numbers represented in previously existing datatypes. For them, we define the following rules of conversion:

- An integer can be converted to a Rational-Scaled floating-point (RSFP) assigning the integer value to the magnitude. If the integer was representing seconds, we set the exponent to 0, and both quantum variables to 1. Otherwise, we need to adjust the scale-factor. For example, for milliseconds, we should set $quantum_n = 1$, $quantum_d = 125$, and $exponent = -3$. This equivalence derivation is showed in Formula 4.3.

$$magnitude \cdot ms = magnitude \cdot \frac{1}{1000} \cdot s = magnitude \cdot \frac{1}{125} \cdot \frac{1}{8} \cdot s = magnitude \cdot \frac{1}{125} \cdot 2^{-3} \cdot s$$

Formula 4.3: Example of Integer to RSFP conversion

- A floating-point (FP) value can be converted to a RSFP setting both quantum variables to one, matching the exponents, and assigning the mantissa to the *magnitude*. As in previous case, depending on the unit associated to the original value, some minor adjustments to the quantum and exponent variables may be necessary. For example, the binary FP 1E-3 ($\frac{1}{8}$) associated to milliseconds can be defined in RSFP setting $quantum_n = 1$, $quantum_d = 1000$, $magnitude = 1$, and $exponent = -3$. This equivalence derivation is showed in Formula 4.4.

$$\frac{1}{8} \cdot ms = 1 \cdot 2^{-3} \cdot ms = \frac{1}{1000} \cdot 2^{-3} \cdot s = 1 \cdot \frac{1}{1000} \cdot 2^{-3} \cdot s$$

Formula 4.4: Example of floating-point (FP) to RSFP conversion

- When importing a rational value defined by two integers, we assign the numerator value to the *magnitude*, we assign the denominator as $quantum_d$, we set $quantum_n = 1$, and we set the $exponent = 0$. Here, if the unit needs to be adjusted, the $quantum_d$ or the magnitude may not include all the possible values represented in the original rational. To ensure that all the numbers in the range will fit when units do not match, we can use a larger integer representation for RSFP than the one used in the rational being converted, or in some cases, compensate using fraction simplification on the quantum variables.

  For example, using 32-bits integers, if we need to define the value $\frac{2^{31}}{3}$ minutes, we can do it by defining it as $\frac{3 \cdot 5 \cdot 2^{33}}{3}$ seconds. Here, the $quantum_d$ is 3, but the numerator is too large to be represented by the *magnitude*, even if we simplify the 3 with $quantum_d$. To be able to represent this number, we can use 64-bits integers, or we can set $quantum_n$ to 20 and magnitude to 2^{31}, or set the exponent to 10 and the magnitude to $5 \cdot 2^{23}$.

In addition, RSFP can be used to represent and operate in a large range, covering all previously available representation values (of FP and rational), operating safely, and introducing new numbers not representable on any of the previous datatypes. For example, $0.1 \cdot 2^{128}$ is representable in RSFP, but it has infinite binary periodicity, and therefore it cannot be represent using binary FP datatypes, and it is too large to be represented as rational.

An alternative representation to the one using four integers could be implemented using two integers and a FP. In this case, the integers define the *quantum*, the *magnitude* is set on the mantissa of the FP, and the *exponent* is set in the FP exponent. If using this representation, the use of Safe-Float (SF) is encouraged for the FP component for preventing undesired approximations.

In Chapter 2, we presented the algorithms proposed by Zeigler for the abstract simulator of DEVS. These algorithms only operate on time variables through four operations: addition, subtraction, equality comparison, and lower than comparison. Algorithms use comparisons for deciding which transition to execute next. They use addition for advancing the chronology and compute the schedule of next internal transition. And they use subtraction for obtaining the elapsed time required for executing the external transition function.

Our focus in this chapter is preventing simulators from introducing errors in the timeline. If the modeler requires other functions for operating with time at the model level, the datatype can be extended to support them. Alternatively, the modeler can cast to other datatypes to operate and cast back for returning the result. If a model produces a approximation internally, we expect the modeler to verify that the model behavior is not affected.

In the rest of this section, we note instances of our datatype as $\langle quantum_n, quantum_d, magnitude, exponent \rangle$ to simplify examples reading.

Agreeing in a common scale-factor

For the design of our datatypes, we assume every time-advance resulting from an atomic model has identical scale-factor. This assumption is a relaxed version of the one made by existing simulators. In most current simulators, the common assumption is that every model in the simulation uses the same time unit and scale-factor. For example, OMNeT++ provides a small set of predefined scale-factors, but the selection of one is global and applied to every time variable defined in the simulation. Under our assumption, we have to find a scale-factor for coupled models connecting atomics. We call this search of a common scale-factor for interoperation an "agreement".

Having mechanisms to discover a common scale-factor for a coupled model, allows us to iterate and find a common scale-factor for the simulation. This globally common scale-factor, allow us to reduce the complexity of our simulation. We propose four approaches to reach the scale-factor agreement: static, dynamic-initialization, local, and global.

In the static approach, the modeler needs to declare the scale-factor in advance for each atomic model. Using the declared scale-factors, a preprocessor or compiler, computes the common scale-factor. After finding the common scale-factor, every value is readjusted in the models. This approach requires information that the modeler may not be able to provide. Implementing this approach requires features not available in some modern programming languages. An example of language providing the necessary features for implementing this approach is C++11.

In Listing 4.21, we show the declaration of a possible implementation in C++11 for the static approach.

```
template<int QN, int QD, int E>
class RSFP_static {
    int _qn, _qd, _m, _e;
    bool initialized=false;
public:
    RSFP_static(const int& m);
    RSFP_static & operator+=(const RSFP_static &rhs);
    bool operator==(const RSFP_static &other) const;
    bool operator<(const RSFP_static &other) const;
    template<int QN2, int QD2, int E2>
    RSFP_static(const RSFP_static<QN2, QD2, E2> &rhs);//cast
    template< template <int QNR, int QDR, int ER> class RS, typename... RSS>
    friend void establish_static_scale_factor(RS rs, RSS... rss);
};
```

Listing 4.21: RSFP-static

The implementation for RSFP-static receives three template parameters for declaring the scale-factor. The constructor only takes a parameter for the *magnitude*. For each scale-factor being used by a model in the simulation, a variable has to be declared. The declaration of each variable registers the scale-factor for participating in the agreement. in this particular approach, we can relax our usage assumption of

only using a scale-factor per model. It is possible, by declaring multiple variables in a single model to obtain a global agreement compatible with them all.

Before initiate the simulation a variable for each scale-factor going to be used has to be defined. We call the establish-static-scale-factor function with all variables defined for statically computing the scale-factor based on the template parameters. The *magnitude* is assigned at runtime, this helps on maintaining encapsulation of the datatype.

To avoid the use of variables without agreement, the function ends by setting the initialized flag. Then, after the agreement is reached, it is not possible to introduce new scale-factors. The already defined variables can be operated for producing any result. But, if a new variable needs to be constructed dynamically, it has to be converted to the type of the agreement for adjustment. This operation may fail, and errors need to be handled properly.

The main benefit of this implementation is that it detects the failure to find agreement at compile time. The limitations are the lack of flexibility to evolve the scale-factor on runtime. This is not only a problem for modelers not able to declare scale-factors in advance, but for exploiting redundancies of the datatype to prevent overflows of internal variables balancing the internal representation. Thus, an operation producing an overflow in any internal variable has to throw an error.

In the dynamic-initialization approach, each variable internally keeps a scale-factor. At least one variable of each scale-factor planned to be used has to be declared before the simulation starts. Each of these variables corresponds to an atomic model, and each of them is registered in a global queue when created.

After declaring all the scale-factors, and before the simulation starts, the agreement is negotiated. Using the reached agreement, every variable in the queue is adjusted. Similar to the static approach, it is possible to relax our assumption of a single scale-factor per model as far as every scale-factor is declared before starting the simulation.

The limitation of this approach are similar to those in the static approach. We still need to declare scale-factors in advance, and there is not easy way to evolve preventing the use of balancing. An advantage, compared to the static approach, is that it could be implemented in languages not supporting the evaluation of compile time expressions. For example, this approach can be used in interpreted languages where no compile step is required as Python or PHP.

In Listing 4.22, we show the declaration of a possible implementation in C++11 for the dynamic-initialization approach.

```
class RSFP_init;

namespace {
    int g_qn; int g_qd; int g_e;
    std::vector<RSFP_init*> g_init_queue;
    bool rs_g_initialized = false;
}
```

```
class RSFP_init{
    int _qn; int _qd; int _e;
    M _m;
    bool initialized=false;
public:
    RSFP_init(const int &qn, const int &qd, const int &m, const int &e);
    RSFP_init & operator+=(const RSFP_init &rhs);
    bool operator==(const RSFP_init &other) const;
    bool operator<(const RSFP_init &other) const;
    RSFP_init(const RSFP_init& rs);
    template<class RS, class... RSS>
    friend void establish_init_scale_factor(RS rs, RSS... rss);
};

void establish_init_scale_factor();
```

Listing 4.22: RSFP-init

In RSFP-init, each variable registers on construction into a global queue. In our case, the queue is defined in an unnamed namespace that all RSFP-init variables are able to access. Before starting the simulation, the establish-init-scale-factor function should be called. This function iterates on all the values in the queue for computing the agreement. Here, the difference against RSFP-static version is that scale-factor is computed on runtime. After reaching the agreement, the queue is iterated again for adjusting every variable magnitude.

Similarly to the RSFP-static, a variable is kept indicating if the scale was initialized. And, a casting is required for introducing any dynamically generated variable. In our C++11 implementation, the main advantage compared to the static version is allowing the use of larger integer representations. In C++11, integral template parameters are restricted to native integers only.

In the static and dynamic-initialization approaches, the scale-factor does not change during the simulation. For operating with them, only the magnitude, and some safety checks, require attention. Thus, the expected performance is similar to integer datatype performance.

For the local and global approaches, it is not needed to declare scale-factors in advance. The addition of this feature, allow us to drop completely our previous assumption if needed. However, the assumption allow us to have guarantees about complexity and performance characteristics.

In the local approach, agreements are found between parameters of binary operations. A global agreement may never be reached, but that does not affect the results of the simulation. Interaction between models is the driving force of the propagation. Thus, models with low level or no interaction may create clusters of scale-factors. Here, we call a cluster of scale-factors to all the variables in the system sharing a common scale-factor. If we keep our original assumption of using a single scale-factor per model, after enough interaction between models, the scale-factors reach a stable state.

In Listing 4.23, we show the declaration of a possible implementation in C++11 for the local approach.

```
template<typename QN=int, typename QD=int, typename M=int, typename E=int>
class RSFP_local{
    QN _qn; QD _qd; E _e;
    M _m;
public:
    RSFP_local(QN qn, QD qd, M m, E e);
    RSFP_local & operator+=(const RSFP_local &rhs);
    bool operator==(const RSFP_local &other) const;
    bool operator<(const RSFP_local &other) const;
};
```

Listing 4.23: RSFP-local

In RSFP-local, no casting operation or call to establish-scale-factor operation is required. Each variable keeps record of the scale-factor and magnitude. For binary operations, the scale-factors of operands are checked, and adjusted to a common scale-factor if necessary.

In the global approach, a globally accessible scale-factor variable is defined. This variable could be implemented as a static variable in the datatype. For each operation used, each operand scale-factor is compared with the global one. In case the operand scale-factor does not match the global one, an agreement between them is searched and registered. If our assumption of usage is respected, the agreement eventually reaches a global consensus. In many cases, global approach converges faster than local one, but it has limitations for concurrency. Using global variables may not be suitable for distributed simulators, and it is a performance bottleneck for multi-thread ones.

The reason why global approach may converge faster is because it is enough to operate once with each scale-factor in the system to reach global agreement, and once with each variable in the system after reaching the agreement to propagate the adjustment. In the case of local approach, it is driven by operations, then a chain of binary operations where every scale-factor participates is required for having a global agreement, and after, a chain of binary operations involving every variable is required for propagating the adjustment.

In Listing 4.24, we show the declaration of a possible implementation in C++11 for the global approach.

```
namespace {
    int g_qn; int g_qd; int g_e;
}

class RSFP_global{
    RSFP_global(int qn, int qd, int m, int e){}
    RSFP_global & operator+=(const RSFP_global &rhs) {}
    bool operator==(const RSFP_global &other) const {}
    bool operator<(const RSFP_global &other) const {}
```

```
};
```

Listing 4.24: RSFP-global

The RSFP-global implementation is similar to RSFP-local. In addition, a global scale-factor is defined in an unnamed namespace, or a static variable.

Local and global approaches performance penalty is only significant until the scale-factors of the simulation stabilizes. In the worst-case scenario, every operation produces a penalty in performance for computing Greatest Common Divisor (GCD) and Lowest Common Multiple (LCM) during the agreement. However, under our usage assumption of having a single scale-factor per model, finding agreements is expected to stabilize. The reason why it stabilizes is because each GCD and LCM work as accumulators of information. If we find an agreement C between A and B, the agreements between C and A, C and B, are again C. Depending on the approach chosen, it could take different number of steps to reach the agreement.

In the case of local approach, we do not need to reach a global agreement, we can work with clusters of scale-factors. The worst case is a single cluster which is the case a global scale-factor is reached. Here, we have to operate every scale-factor with each other (directly or not), each time a new participant was added to the local agreement it is computed and adjusted. This requires changes in the local scale-factor as many times as operations adding scale-factors knowledge are being operated. The maximum number of agreements taken by a single variable is one per scale-factor defined in the system. This is the case of a variable interacting once with each variable in the system before any interaction between them happens.

In the case of global approach, after every scale-factor (or model) was operated at least once, a global agreement is reached. Then, we have as many changes to the global scale-factor as scale-factors in the system before stabilize it. After global agreement is reached, the first time each variable participates in an operation adjust its internal scale-factor and it is never changed again.

It is possible, taking an hybrid approach using static or dynamic-initialization ideas to add guesses of scale-factor, and operate using local or global approaches ideas. The hybrid approach allows exploiting partial declarations in advance, and allowing an evolution of the scale-factor during simulation. Compared to static and dynamic-initialization approaches, it adds in allowing evolution of the scale-factor. And, compared to local and global approaches, it adds in giving a head start to the stabilization of the scale-factors.

To check whether an agreement needs to be done, we can check $quantum_n$, $quantum_d$, and $exponent$. In the case of local and global agreements, we propose to include an extra integer variable called scale-version for reducing the quantity of checks required.

In the case of global, we have a global-scale-version and local scale-version of each variable in the system. The global-scale-version is incremented with each new different agreement. And, before each operation, operands' scale-versions are compared against the global one. If the global and local scale-versions match, we know the whole scale-factor matches. Else, we adjust the agreements and, if required, increment the global scale-version.

For example, if we compare equality between $\langle 1, 1, 1, 0\rangle$ and $\langle 2, 2, 1, 0\rangle$, they could agree on using $\langle 1, 1, 0\rangle$ as the scale-factor and call it scale-version 1. In a subsequent operation, if they are compared or added, they will check that they are both in version 1, knowing that $quantum_n$, $quantum_d$ and $exponent$ are the same for both. In the case of transitivity, if we have three numbers, the operation between two of them can produce an agreement, the third may find a new agreement with one of them, lets call it 3. After operating with the other variables again, the three will have the same scale-version.

In the case of local, it is harder to define the version numbers. In this case, incrementing variables is not enough, because they may produce the same number in different clusters. However, it could be implemented following any method allowing defining unique-ids. For example, we can use an identifier based on lists of participants, a global generator of unique identifiers (UIDS), or any other method used for generating UIDS for distributed systems.

RSFP operations

We introduce here the operations required for implementing DEVS simulators: comparison for equality, inequality (lower than), addition, and subtraction. It is easy to extend to other operations, using the same ideas, for use in other formalisms.

If we use the static or dynamic-initialization approaches, every value is known to be using the same scale-factor, then the operations are equivalent to operate with the magnitudes.

In the case of local or global approaches, enforcing the agreement of scale-factor could be part of each operation.

This is unnecessary for some operations, which can find the result without need of a common scale-factor, e.g. comparing a negative number with a positive number,

In order to compare if numbers are equal, checking the four variables is enough to provide a positive answer. If any of the variables differ, we cannot decide yet. And, to answer properly, we try to produce an agreement for the scale-factor and compare their magnitudes. If the agreement cannot be reached, we are certain that the numbers are not equal, since if they were equal, the agreement could be the scale-factor of any of them. Else, we compare the magnitude to decide.

When comparing by lower-than, several things need to be checked. First, we must check if the operands are equal. If they are, we can give a negative answer.

If they are not equal, then one of them is lower than the other. If an agreement can be found, we can compare the magnitude to decide. For example, if we compare $\frac{1}{2}$ to $\frac{1}{4}$, we can represent them in a common scale-factor as $\langle 1, 4, 2, 0\rangle$ and $\langle 1, 4, 1, 0\rangle$. In this case, given the magnitudes are 2 and 1, the second corresponding to $\frac{1}{4}$ is the lower one. It is unnecessary to check the other variables, which are known to be equal after an agreement was found.

It is possible, when the difference between two operands is large, that the agreement cannot be found. If this happens, we can still compute the comparison by lower than. First, we can check if

the signs of both numbers match; otherwise the negative is lower than the positive. Second, we can search for a partial agreement by adjusting the operands to have the same $quantum_d$, and the closest possible exponents without rounding. Having a common $quantum_d$, we can compare numbers looking at their exponents' difference. If the difference between their exponents is larger than two times the size of the integers used to represent the internal values, then large exponent correspond to larger value in positive numbers, and lower exponent corresponds to larger value in negative numbers. Else, we can use a temporary larger datatype for comparing the *magnitude*, $quantum_n$ and *exponents*.

For example, using 32-bit integers, we can compare the largest number representable using the exponent 0, $\langle 2^{32}-1, 1, 2^{32}-1, 0\rangle$, against the smallest representable number using exponent 63 and same $quantum_d$, $\langle 1, 1, 1, 63\rangle$. In Formula 4.5, we show the first number in the example is lower than the second one.

$$\begin{aligned}\frac{2^{32}-1}{1}\cdot(2^{32}-1)\cdot 2^0\cdot s &< \frac{1}{1}\cdot 1\cdot 2^{63}\cdot s\\ (2^{32}-1)\cdot(2^{32}-1)\cdot s &> 2^{63}\cdot s\\ (2^{64}-2^{33}+1)\cdot s &< 2^{64}\cdot s\end{aligned}$$

Formula 4.5: Example of comparing two RSFP variables which scale-factor agreement cannot be found

In this case, a common scale-factor is not produced, and the exponent difference is not enough to decide using the common $quantum_d$. Then, we have to compare $quantum_n$ and *magnitude* values too. Here, we can set the product of *magnitude* by $quantum_n$ of each number in a temporary 64-bits variable. And then we shift-right the temporary variable of the smaller exponent number the difference of the exponents. At this point both temporary numbers are comparable and the smaller *exponent* one has been truncated. In our example, we have a difference of 63, which is lower than two times the integer size (it is 63), the $quantum_d$ is common to both numbers and there is no possible reduction since $quantum_n$ and magnitude are co-primes. Then, we set both temporary variables to 1. In the case the temporary variables were different, the result of their comparison is the final result. In this case, given their equality, we answer that the one with the smallest exponent is the largest. This is because we know the one being truncated had at least a 1 at the last position, else it could be reduced further in the partial agreement.

Finally, the addition operation is usually adding the magnitudes of numbers in agreement. A special case has to be handled when magnitudes' addition result in overflow. For this case we can return a result that is not represented in the same scale-factor using the balancing tactics presented later in this section, or raise an error. The introduction of a new scale-factor may cascade in a global readjustment of variables.

In case the scale-factor of the operands have different exponent, we first check their difference. If the difference is larger than two times the bits of the magnitude, we raise an error. Otherwise, we use a temporary variable to compute the addition using the lower exponent in both variables, and balance the result. If the result does not fit the original variable after balancing, an error is raised. Else, we return the result using the newly balanced representation.

Balancing the representation

The datatype proposed has redundancy of values, similar to rational representation. In Formula 4.6, we show an example of three different representations of one second.

$$1 \cdot \frac{1}{1} \cdot 2^0 \cdot s = 2 \cdot \frac{1}{2} \cdot 2^0 \cdot s = 1 \cdot \frac{1}{2} \cdot 2^1 \cdot s$$

Formula 4.6: Example of RSFP representation redundancy

We can exploit this redundancy to work around operation overflows of any component of the datatype.

For overflow on $quantum_n$, we can balance with $quantum_d$. If simplifying is not enough, some weight can be shared with the *magnitude*. To transfer weight to the *magnitude*, we divide $quantum_n$ by one of its divisors and multiply the *magnitude* by the same divisor. Another alternative, if the binary representation ends in 0, *magnitude* digits can be shifted and the *exponent* adjusted to compensate.

For example, using 32-bit integers, if the result of an operation produces $\langle 2^{33}, 8, 1, 4\rangle$, we cannot fit the value of $quantum_n$ in its variable. Instead, we can increment the *exponent* to 10 and set $quantum_n$ to 2^{27} to make it fit in the representation. Or, we can simplify $quantum_n$ against $quantum_d$, setting $quantum_n$ to 2^{30} and $quantum_d$ to 2^3. Or, we can increment *magnitude*, for instance to 16, and compensate reducing $quantum_n$ to 2^{29}. Prioritizing any of these tactics over others is an implementation detail.

For overflow on $quantum_d$, simplification against $quantum_n$, *maginitude*, or *exponent* can be used. We can simplify $quantum_d$ against the *magnitude*.

For example, using 32-bit integers, if the result of an operation produces $\langle 8, 2^{33}, 4, 0\rangle$, we cannot fit the value of $quantum_d$ in its variable. Instead, we can decrement the *exponent* to -10 and set $quantum_d$ to 2^{23}. Or, we can simplify $quantum_d$ against $quantum_n$, setting $quantum_n$ to 1 and $quantum_d$ to 2^{30}. Or, we can simplify $quantum_d$ against the *magnitude*, setting *magnitude* to 1 and $quantum_d$ to 2^{31}. Prioritizing any of these tactics over others is an implementation detail.

For an overflow on the *magnitude*, we can simplify it against $quantum_d$. Or, we can transfer some representation weight to $quantum_n$. Or, in some cases, we can shift digits and adjust the *exponent*. The options for $quantum_n$ and *magnitude* are similar, since numerically permuting $quantum_n$ and *magnitude* have the same meaning. The difference between these variables is the intended use, which impacts in the design of the operations.

For example, using 32-bit integers, if the result of an operation produces a $quantum_n$ of 1, a $quantum_d$ of 8, a *magnitude* of 2^{33}, and an *exponent* of 4, we cannot fit the value of *magnitude* in its variable. We can increment the exponent to 10 and set the *magnitude* to 2^{27}. We can simplify against $quantum_d$, setting *magnitude* to 2^{30} and $quantum_d$ to 64; or we can increment the $quantum_n$ to some value as 16 and reduce the same factor from *magnitude* setting it to 2^{29}. Prioritizing any of these tactics over others is an implementation detail.

All these simplifications and manipulations of internal representation should be delayed until they are

absolutely necessary to keep complexity of operations low. Running a simplification requires computing expensive algorithms to run as GCD, and LCM.

Handling errors

We stated that comparison operators always return a result, but the addition may throw an error. This error occurs in case of failure to reach an agreement on the result's scale-factor. This is because our datatype does not have addition-closure. This problem previously discussed in the floating-point (FP) datatype, has less chance to happen with RSFP, but it is still possible given the right conditions. The reason why this happens less is because the number line represented by RSFP is denser and balancing can delay the problem. In case the issue arises, we detect and notify the error when adding two numbers that cannot be precisely represented, as opposed to FP that silently approximates and continues.

In the case an error is notified, some options may be evaluated to complete the simulation. First, we can promote the integers used as components of the datatype to larger representations, and retry the last operation. If a proper size cannot be estimated, or the programming language used does not allow this practice, a dynamic arbitrary length integer can be used, i.e. the BigInt provided by the Boost multi-precision library. We do not use this kind of datatypes by default because of their performance penalty. A second alternative would be to introduce approximation algorithms to be run when the error is detected. This would allow the simulation to incorrectly advance the trajectory, but it should still mark in the timeline where and what errors were introduced for after-run analysis.

In Section 4.3, we will show experimental results comparing the implementations proposed for RSFP. The results obtained are compared against previously used datatypes, and other proposed datatypes in this thesis.

4.2.2 Multi-Base Floating Point

The Rational-Scaled floating-point (RSFP) datatype presented in Section 4.2.1 solves basic issues in current simulators. Nevertheless, it still has some usability limitations. Our first concern is a representation restriction over large and small numbers.

Below, we show again the formula for RSFP interpretation for showing this problem.

$$magnitude \cdot \frac{quantum_n}{quantum_d} \cdot 2^{exponent}$$

This representation makes it possible to represent any (binary) FP number. Nevertheless, it prevents the representation of any large, or small, numbers produced by multiplying only co-primes of 2. For example, the largest odd number being represented has to be using exponent 0 (or its simplification against $quantum_d$ be 0), because using any larger value makes the number even. Then, an accumulator repeatedly adding an odd number reaches a point where next addition result is not representable in RSFP. This is a consequence of not having addition-closure in the datatype.

Here, we propose Multi-Base floating-point (MBFP) as an alternative to the use of RSFP. This datatype covers a different set of numbers, and it has a more compact representation. These characteristics make MBFP more appropriate than RSFP in some simulation scenarios. In Formula 4.7, we show how to interpret a number represented using MBFP.

$$mantissa \cdot base^{exponent}$$

Formula 4.7: Multi-Base floating-point (MBFP) interpretation

The MBFP datatype can be implemented using three integer variables representing *base*, *mantissa*, and *exponent*. For example, 3^{-70} (which is a number not representable using 32-bit integers in RSFP) can be represented in MBFP setting *mantissa* to 1, *base* to 3, and *exponent* to -70. Operating is simpler, because only the base requires agreements.

A drawback here is the need to work with non-binary powers, which are not optimized in computers the same as powers of 2. However, we will show in the algorithms that it is never required to power using high exponents. We are limited to use exponents below the size of the *mantissa*. Multiplying for a power of the base is seen as shifting digits (left or right) in the *mantissa* (considering the *base* in use). When we compare or add two numbers, we may have to adjust the *exponent* of both operands to match. The difference between exponents is compensated by shifting digits in the *mantissa*. If we have to shift more digits than the size of the mantissa, the *mantissa* losses any value. Similarly, the resulting *mantissa* wouldn't fit in the result.

In the case the global agreement of the *base* is known in advance, we can pre-compute all the powers of the *base* with exponents between 0 and size of the *mantissa*, and implement the function as a dictionary of constant values. This new datatype is also providing a correct representation for all those numbers represented in the currently used datatypes we reviewed. We define the following rules of conversion between them.

For converting an integer to MBFP, we set the integer value in the *mantissa*. If the integer was representing seconds, we set the exponent to 0, and the base to 2. Otherwise, we need to adjust the unit. For example, for milliseconds, we could set base to 10, and exponent to -3. Every integer having equal or smaller size than the *mantissa* is representable using this conversion.

Conversion from FP datatypes is also simple. We can set the base to 2, and copy the *exponent* and *mantissa*. It is necessary to add the implicit 1 when writing the MBFP *mantissa*. In Formula 4.8, we show an example of converting a half-precision FP to MBFP.

$$\underbrace{0}_{sign}\ \underbrace{10001}_{exponent}\ \underbrace{1100100000}_{mantissa} = 1.1100100000_{bin} \times 2^{17-15} = 1.78125 \times 4 = 7.125 = 2^{-5} \times 57 \times 2^2 = 57 \times 2^{-3}$$

$$MBFP :< \underbrace{57}_{mantissa}, \underbrace{2}_{base}, \underbrace{-3}_{exponent} >$$

Formula 4.8: Example of half-precision floating-point conversion to Multi-Base floating-point (MBFP)

Conversion from Rational can be achieved setting the exponent to -1, the denominator as the base and the numerator as the mantissa.

Conversion from and to RSFP is not always possible, even assuming both use same integer size for internal variables. In Formula 4.9, we show an example of a valid RSFP conversion to MBFP.

$$RSFP :< \underbrace{3}_{quantum_n} , \underbrace{5^5}_{quantum_d} , \underbrace{1000}_{magnitude} , \underbrace{5}_{exponent} >$$

$$\frac{3000}{5^5} \times 2^5 = \frac{3}{3125} \times 1000 \times 2^5 = 3000 \times 4^5 \times 10^{-5} = 3072000 \times 10^{-5}$$

$$MBFP :< \underbrace{3072}_{mantissa} , \underbrace{10}_{base}, \underbrace{-2}_{exponent} >$$

Formula 4.9: Example of a successful Rational-Scaled floating-point (RSFP) conversion to MBFP

In the case the RSFP being converted has a large *exponent*, the denominator cannot be defined as a *base* with a negative exponent. Making it impossible to convert to MBFP if $quantum_d$ is not zero. In Formula 4.10, we show an example of a RSFP value non-convertible to MBFP.

$$RSFP : < \underbrace{1}_{quantum_n} , \underbrace{5}_{quantum_d} , \underbrace{1}_{magnitude} , \underbrace{2^{20}}_{exponent} >$$

$$\frac{1}{5} \times 1 \times 2^{1048576} = 5^{-1} \times 2^{1048576}$$

$$MBFP :?$$

Formula 4.10: Example of an RSFP value non-convertible to MBFP

In this example, if we are setting the *base* to 5 and the *exponent* to -1, the product of the magnitude by the power does not fit in the *mantissa*, and if we are setting the *base* to 2, we are not allowed to set a denominator.

Finally, we showed earlier that some numbers representable in MBFP, as 3^{-70}, cannot be converted to RSFP. This shows that MBFP and RSFP representation domains overlap, but they are not equal. In different application domains we may find a better fit for each of them.

Similar to what we discussed in RSFP, MBFP can be implemented using the same four different agreement approaches. The difference between the datatypes is in the details of how to compute the agreement. Reaching an agreement for the base is based in LCM only, not requiring implementation of GCD. However, common approach is to implement LCM using GCD. The rule for agreement between two numbers A, and B, is the following: the common *base* is the LCM of the bases of A and B, the *exponent* is the minimal exponent between A and B, and the *mantissa* needs to be adjusted for compensation shifting digits.

In Formula 4.11, we show an example of an agreement for addition between two MBFP numbers.

$$A :< \underbrace{2}_{mantissa}, \underbrace{3}_{base}, \underbrace{-2}_{exponent} >$$

$$B :< \underbrace{4}_{mantissa}, \underbrace{4}_{base}, \underbrace{2}_{exponent} >$$

$$LCM(3,4) = 12$$

$$min(-2,2) = -2$$

$$\texttt{After adjustment} : \underbrace{32 \times 12^{-2}}_{A} + \underbrace{11520 \times 12^{-2}}_{B} = \underbrace{11552 \times 12^{-2}}_{Result}$$

$$\texttt{Result} :< \underbrace{11520}_{mantissa}, \underbrace{12}_{base}, \underbrace{-2}_{exponent} >$$

Formula 4.11: Example of adding two MBFPs

Implementation details about handling errors and operating are the same in MBFP than RSFP. Obtaining the agreement and the internal representation are the differences.

In Section 4.3, we will show experimental results comparing the implementations proposed for MBFP. The results obtained are compared against previously used datatypes, and the other proposed datatypes in this chapter.

4.3 Experimental evaluation

To evaluate the proposed datatype empirically, we implemented every proposed datatype in C++14, and compared against native datatypes double and int-32.

We experimented using DEVStone [WGGA11], a benchmark specially designed for comparing implementations of simulators. We included a brief explanation of this benchmark in Section 2.3.5.

For measuring the values, the DEVStone models where implemented for a customized version of aDEVS-2.8. The customizations were restricted to the datatype adevs-time and its uses. The modification allowed us to obtain runs of the benchmark with each of the datatypes being evaluated.

All experiments were compiled using clang 4.1 over the x86_64 version of FreeBSD 10.1-18 operating system. The compiler was used with level 2 optimization. We discussed in Chapter 3, how most compilers use incorrect arithmetic optimizations for floating-point (FP). However, it is a common practice by users to enable them. Thus, we want our benchmark results for double to reflect the users expectations.

In Table 4.3, we present the results obtained for each datatype in four Low-level of Interconnections

(LI) configuration experiments, and in four High-level of Input coupling (HI) configuration experiments. To simplify comparisons, the overhead is measured relatively to the one obtained for double. In these experiments, the period of time-advance used by the DEVStone models was handpicked to produce correct results using any datatypes being compared. Every experiment was ran 20 times, with no significant variance in the results.

In Table 4.4, we show the theoretical number of transitions being executed according to the DEVStone formulas.

From the obtained results, we see RSFP-static and MBFP-local are leading the performance comparison. And, they produce even better results on LI than HI. We attribute this to the fact that HI configurations make more intensive use of comparison than arithmetic operations.

The worst performance obtained in the LI configurations is MBFP-static. This is attributed to power function used for adjusting the mantissa when the exponents are different in the operands. This is not a problem for RSFP-static because it removes any adjustment by fixing the scale-factor at compile time. We believe the difference with others in the MBFP family is due to our implementation using C++ templates preventing some optimization in the compiler.

Contrary to what we expected some datatypes outperformed int-32. We reviewed the results from the simulation, and found them to be correct. We believe this is a result of memory allocation optimization by the compiler, enable the CPU making better use of the internal caches. This is a point that should be looked further to fully understand its implications.

In both datatypes, RSFP and MBFP, the init and global implementations are doing worse than others in performance. We attribute this to the lost of locality affecting the cache. We still believe in the value of these datatypes for context were CPU have lower sizes of cache.

In the case of Safe-Float (SF), we found its performance is slightly better than double in some cases. This is misleading, we cannot trust the results of SF, because the compiler reorders the code. This reorder in current compiler implementation may misplace the flags used for checking safety. A misplaced flag could result in an undetected safety error. In Table 4.5, we present a re-run of the same experiments with no optimizations comparing SF to double. In these results, we see that without optimizations, SF has always a higher overhead than double, as expected.

In the experiments presented, we worked with values representable in every datatype evaluated. Values not representable in a particular datatype are never used in these experiments. Different results are expected for experiments in other simulator architectures and platforms. Proper evaluation of the datatypes on a particular implementation is encouraged.

4.4 Summary

In this chapter we described three new datatypes for being used as time variables for Discrete-Event Simulation (DES): Safe-Float (SF), Rational-Scaled floating-point (RSFP), and Multi-Base floating-

	LI			HI		
datatype	width	depth	overhead	width	depth	overhead
RSFP-static			-7.48 %			-27.89%
RSFP-init			17.25 %			11.28%
RSFP-global			19.56 %			-1.06%
RSFP-local			-5.87 %			-25.17%
int-32			-2.40 %			-10.26%
double	50001	11	0.00 %	100	200	0.00%
MBFP-static			78.97 %			-16.29%
MBFP-init			20.08 %			-3.11%
MBFP-global			8.32 %			-15.74%
MBFP-local			-7.34 %			-30.61%
SF double			-0.87 %			10.21%
RSFP-static			-11.33 %			-21.81%
RSFP-init			19.40 %			31.82%
RSFP-global			19.56 %			8.67%
RSFP-local			-6.80 %			-17.20%
int-32			-3.92 %			-1.40%
double	5001	101	0.00 %	142	100	0.00%
MBFP-static			81.39 %			-10.82%
MBFP-init			18.36 %			4.24%
MBFP-global			6.95 %			-3.91%
MBFP-local			-9.97 %			-17.89%
SF double			-7.13 %			20.75%
RSFP-static			-10.77 %			-36.67%
RSFP-init			12.98 %			-1.12%
RSFP-global			19.56 %			-11.71%
RSFP-local			-7.08 %			-28.91%
int-32			-3.52 %			-21.64%
double	501	1001	0.00 %	205	50	0.00%
MBFP-static			73.23 %			-28.42%
MBFP-init			15.09 %			-18.54%
MBFP-global			7.58 %			-16.01%
MBFP-local			-9.43 %			-34.46%
SF double			-3.34 %			-4.31%
RSFP-static			-12.35 %			-42.77%
RSFP-init			12.55 %			12.71%
RSFP-global			19.56 %			-13.00%
RSFP-local			-4.90 %			-21.14%
int-32			-0.45 %			-23.91%
double	51	10001	0.00 %	720	5	0.00%
MBFP-static			37.99 %			6.15%
MBFP-init			6.62 %			23.50%
MBFP-global			3.95 %			16.35%
MBFP-local			-7.39 %			-40.47%
SF double			-4.25 %			6.79%

Table 4.3: DEVStone comparing time datatypes in aDEVS

configuration	width	depth	$\#\delta_{int}$ transitions	$\#\delta_{ext}$ transitions
LI	51	10001	499999	499999
	501	1001	499999	499999
	5001	101	499999	499999
	50001	11	499999	499999
HI	100	200	1004951	1004951
	142	100	1005148	1005148
	205	50	1034636	1034636
	720	5	1038241	1038241

Table 4.4: DEVStone theoretical transitions on experiments

configuration	width	depth	overhead
LI	51	10001	9.46%
	501	1001	14.75%
	5001	101	14.07%
	50001	11	5.72%
HI	100	200	2.80%
	142	100	28.69%
	205	50	21.83%
	720	5	31.29%

Table 4.5: DEVStone comparing Safe-Float to double without compiler optimizations

point (MBFP). These datatypes do not approximate at the time of operating. This prevents hidden errors in the trajectories of the simulation.

We discussed these datatypes limitations, and presented an empirical evaluation of their performance using the DEVStone benchmark. In Chapter 5, we will show a method for extending these datatypes for adding support for computable irrational values to our timelines.

Chapter 5

Irrational support for datatypes representing time

It is common for different research fields to model and simulate using different formalisms. One of the uses of Discrete-Event Simulation (DES) formalisms is to combine models from multiple formalisms for easing cross-field research collaboration. For instance, this is what DEVSbus [KK98] proposes using Discrete-Event System Specification (DEVS) as the formalism for composing models from multiple formalisms.

It is common to use irrational numbers when defining models in certain fields as physics, mechanics, or chemistry. In addition, it is also common to use irrationals for any model relying on geometrical concepts. For example, modeling wheels may refer to circles leading to usage of π, and modeling traffic using polygons for the map may lead to square roots.

In Chapter 4, we proposed datatypes to identify and prevent timeline errors. The presented datatypes are only usable for models working with rational time values. The purpose of these datatypes is replacing floating-point (FP), or other basic datatypes. We defined them for being replacement for datatypes used in the simulators surveyed in Section 3.2. Those simulators never allowed (syntactically) the definition of models using irrational numbers for time.

In most generic DES formalisms, like DEVS, the domain of the variables that hold the discrete points in time is $\mathbb{R}^+$. Therefore, it is completely legal for a model in such formalisms to produce or process an event at an irrational time, like π seconds.

The problem of including irrational numbers in Time datatypes for DESs is not new. We can categorize previously proposed solutions in three groups: interval arithmetic, fixed-length approximation, and symbolic solving. These solutions solve different subsets of representation issues, but they carry, in some cases, some new concerns or limitations.

In approaches based on interval arithmetic, each number is represented by an interval rather than a discrete value. For example, this is the case for Rational Time Advance DEVS (RTA-DEVS), previously

discussed in Sections 2.3.4, and 3.3.5. The main problem with interval arithmetic is the comparison operations, which may be undefined. If the intervals representing the real numbers do not overlap, we can decide that one is the lowest. But, in the case they do overlap, we do not know which one is the lowest. In addition, equality is undefined, or false, for every pair of numbers that are not both single point intervals. Two equal numbers represented as interval implies an overlap exist. And, if those numbers are irrationals, we are never comparing single point intervals.

In the case of DES, not being able to decide comparisons is equivalent to not knowing which event to process next. And, if special functions for handling simultaneous events are defined, as in DEVS models, we could never decide to use them.

Additionally, summing intervals increments the size of the intervals, but never decrements it. To obtain the current simulation time in DES, a large quantity of additions are used. Thus, the accumulation of interval lengths increases the chances of overlapping in later operations. And such overlapping results in undecidable comparisons. At the point of processing an undecidable comparison, we must crash the simulation to avoid incorrect results. In some cases, we can obtain correct results by restarting the simulation using smaller initial intervals. However, this never guarantees solving the problem, the overlap could be between two equal numbers.

A second approach is to use fixed-length approximations as fixed-point, and floating-point (FP) representations. This is the most popular approach because of its implementation simplicity (usually native datatypes can be used). In addition, using these datatypes implemented in hardware have great performance characteristics.

None of these datatypes offer syntax for declaring irrational values. Then, the programmer approximates them at time of declaration, leaving not trace of its approximation in the code. For example, if π needs to be defined, and we are using fixed-point with 5 decimals, π is defined as 3.14159. Being the values declared as valid rational values, previously proposed datatypes are not able to catch any introduced error.

Finally, in Symbolic DEVS (Sym-DEVS) [ZC92, ZPK00], it was proposed the use of symbolic algebra to represent time. In symbolic algebra, numbers are represented by expressions defining them. The operations over these numbers are defined as composition rules of the expressions defining them. And, solvers are implemented to evaluate comparisons when needed.

This approach looks promising in the sense of accuracy, but as far as we know, Sym-DEVS [ZC92, ZPK00] was the only formalism using it. In Sym-DEVS, expressions are limited to represent roots of polynomials, which restricts its application to only non-transcendental numbers, excluding π, e, and other commonly used irrationals. Other problem is the solver is not specified, so we do not know how to detect equal irrational numbers.

Symbolic algebra approach seems to be the most promising of previous ones. For some well know expressions providing a way to avoid falling into non-decidable operations is possible. Using expression manipulation techniques we can detect relations between two expressions without the need to generate any digit.

Symbolic Algebra is also implemented in some Mathematical applications and libraries. In them, fixed-length approximations are used for operating when symbolical manipulation of an expression is unknown.

An example to show how this method is effective is the implementation of the comparisons of square roots of rational numbers. In fixed-length or intervals it is usually impossible to distinguish the square root of 2 from the square root of a close number, e.g. 2.00000000001. Using interval arithmetic, the results of the square roots could overlap, making them incomparable. Using fixed-length approximation, numbers could be represented equal, when they are not. However, it is easy to compare them using Symbolic Algebra manipulations, for example comparing the two radicands is enough for stating their comparison result without computing any square root.

5.1 Datatype representation

We mentioned, in Section 2.2, there are formal proofs about numbers that are not computable. And, it is proven, in the general case, it cannot be decided neither if one of the computable ones is rational or not, nor deciding the compare of two of them. Using fixed-length approximations is certainly not an acceptable solution. For example, in a one-digit expansion, π and 3 are considered equal, while we know they are different. However, even when it is proven that there is no solution in the general case, we can restrict our set of numbers to obtain satisfactory results.

In this chapter, we present a datatype increasing the set of values represented by previous presented datatypes. This datatype introduces some known subsets of computable irrational numbers. Our approach is inspired of ideas from Computable Calculus, and Symbolic Algebra. And, our main objective is to generate correct trajectories.

Our new datatype represents a real with two components, a rational and an irrational. For the rational component we could use any rational datatype with no approximation, in practice we use Rational-Scaled floating-point (RSFP) or Multi-Base floating-point (MBFP). For the irrational part we use a composition of known subsets of irrationals, and provide a method to construct them uniquely. The reason for the unique construction is being able to decide irrationality and equality by construction. Once we are able to decide these two operations, the other operations can be decided too.

For providing unique construction, we define the structure of the datatype as a tuple $\langle r, I\rangle$ where r is a rational number, and I is a set of tuples $i_k = \langle c_k, A_k\rangle$ with c_k a rational coefficient and A_k an expansion algorithm describing a computable irrational number selected from a predefined set. This representation can be interpreted as shown in Formula 5.1.

$$r + \sum_{i \in I} i.c \cdot i.A$$

Formula 5.1: Irrational datatype interpretation

For example, we can represent π as $\langle 0, \{\langle 1, \pi\rangle\}\rangle$, $\frac{1}{3}$ as $\langle \frac{1}{3}, \emptyset\rangle$, and $\frac{e^\pi}{2}$ as $\langle 0, \{\langle \frac{1}{2}, e^\pi\rangle\}\rangle$. If we need to

add these three numbers, the result can be represented as $\langle \frac{1}{3}, \{\langle 1, \pi\rangle, \langle \frac{1}{2}, e^{\pi}\rangle\}\rangle$.

Initially, we propose two subsets of irrational numbers, rational multiples of π, and rational multiples of e^{π}. We introduce later, the irrational numbers obtained as square root of integer and rational numbers (using integer exponents). We consider these are a large addition to current state of the art. Also, we explain how this set can be enlarged to cover other specific cases when needed by some modeling scenarios. A restriction for our datatype is that it only provides four operations (+, -, =, <); those are the operations needed to implement DEVS simulators' abstract algorithms[ZPK+76, ZPK00]. We discuss how to extend to other relevant arithmetic operations at the end of this chapter.

5.2 Operations

For some operations, we work with interval arithmetic and real number generation algorithms. Here, intervals are used for indicating the precision of the generated digits. If an operation cannot be resolved with current precision, it is detected by overlaps with the other operand and more digits are created incrementing the precision of the operation. In case the precision cannot be incremented further, we report an error. Reasons for this are insufficient memory to generate irrational digits, or overflows in rational components.

The first operation we are interested in is the addition. We define the addition of two numbers as the addition of each of their components. First, we add the rational components. Second, we add the coefficients of each element in the irrational component of the first operand with its matching part in the second operand. This has low impact on the performance of the addition operation. For instance, if we allow a single irrational component, as rational multiples of π, we require adding two rational values, the rational component, and the coefficients associated to π. The addition complexity for this is the complexity of 2 rational additions. In the general case, the complexity of an addition is the complexity of $dim(A) + 1$ rational additions, where A is the set of irrational constants defined. In the case of subtraction, we change the sign of the second operand and add.

The second operation we are interested in is the comparison by equality. The uniqueness of the representation can be used to compare two numbers for equality just by looking at their components. The complexity of this operation is also dependent on the rational datatype used, an equality operation requires $dim(A) + 1$ rational equality operations.

We want to improve our datatype to support other families of irrationals, beside the rational products of π. We can, for example, introduce the rational products of e^{π}. To be certain that we can still distinguish every representable number, we need to prove that the property shown in Formula 5.2 applies to the datatype.

The idea is to be certain that a number defined by an algorithm cannot be described by the linear composition of the others over $\mathbb{Q}$. In the case of A having π and e^{π}, there is no rational number that can be multiplied or added to it in order to make them equal or rational, the proof can be found in [NP01].

While we cover the requirements, we can keep introducing new numbers, for example, we can use

$$\texttt{Given } A\texttt{, the set of all irrational algorithms defined,}$$
$$d = dim(A), \forall i \in \mathbb{N}, 0 < i \leq d, \forall r \in \mathbb{Q}^d \colon \sum_{j=0}^{d} r_j \cdot A_j = 0 \Leftrightarrow \forall j \in \mathbb{N}, 0 < j \leq d, r_j = 0$$

Formula 5.2: Property for irrationals having unique representation

$\langle \pi, e^{\pi} \rangle$, or $\langle \pi, \sqrt{2} \rangle$, or $\langle e, \sqrt{3} \rangle$, or other useful combinations, for defining the set A, and it can be defined in a simulation by simulation basis.

The third operation, compare by greater-than, is the most complex operation requiring multiple steps. The first step is checking for equality, if both numbers are equal, we return false. The second step is discarding the components that are equal, if we obtain a single component that is not equal, which is the common result when operating with rational numbers, we can compare the coefficient of the distinctive component for deciding. In case more than one component are different, this is the only moment we require algorithmic expansions of the distinctive components. The distinctive component are multiplied by the coefficient and compared following the algorithms described in [Abe01]. The complexity varies depending on the expansion algorithms of each component; in case we operate between rational numbers, the complexity is only incremented in $dim(A)$ rational equality operations.

We show in Listing 5.1 a possible implementation of this datatype with support for $\langle \pi, e^{\pi} \rangle$ in C++11. The code is for reference, more sophisticated versions can be written allowing selection by template parameters of the subsets being used, and detecting conflicts between them, this would be a proper approach for a production quality version. Similar code can be implemented in other languages. However, lacking of generic meta-programming features may have an impact in usage flexibility and performance.

```
class iTime{
    rational _q;
    rational _pi_coef;
    rational _e_pi_coef;
    const int MAX_ALLOWED_DIGIT_EXPANSION=1000;
    const int get_digit_k(const int& k) const { ... }
public:
    iTime(): _q{0}, _pi_coef{0}, _e_pi_coef{0}{}
    iTime(const rational& q, const rational& pic,
          const rational& epic):
          _q{q}, _pi_coef{pic}, _e_pi_coef{epic}{}
    iTime(const iTime arg){
        _q = arg._q;
        _pi_coef = arg._pi_coef;
        _e_pi_coef = arg._e_pi_coef;
    }
    iTime operator+=(const iTime& arg){
        _q += arg._q;
        _pi_coef += arg._pi_coef;
```

```
        _e_pi_coef += arg._e_pi_coef;
        return *this;
    }
    bool operator==(const iTime& rhs) const{
        return (_q == rhs._q && _pi_coef == rhs._pi_coef &&
                _e_pi_coef == rhs._e_pi_coef;
    }
    bool operator<(const iTime& rhs) const{
        //working with single coefficients
        if (_pi_coef == rhs._pi_coef && _e_pi_coef == rhs._e_pi_coef)
            return _q < rhs._q;
        if (_q == rhs._q && _pi_coef == rhs._pi_coef)
            return _q < rhs._q;
        if (_q == rhs._q && _e_pi_coef == rhs._e_pi_coef)
            return _q < rhs._q;
        //checking non-decimal part of number
        bri l_rational = _q + _pi_coef * pi.rational()
            + _e_pi_coef * epi.rational();
        bri r_rational = rhs._q + rhs._pi_coef * pi.rational()
            + rhs._e_pi_coef * epi.rational();
        if (l_rational < r_rational) return true;
        if (r_rational < l_rational) return false;
        //checking decimal part
        for (int i=0; i < MAX_ALLOWED_DIGITS; i++){
            if (get_digit_k(i) < rhs._get_digit_k(i)) return true;
            if (get_digit_k(i) > rhs._get_digit_k(i)) return false;
        }
        throw std::exception();
    }
};
```

Listing 5.1: Irrational time

5.3 Extending with parametric subsets

A third group of interesting irrational numbers to include in our datatype are the square roots. Here, we are not referring to a single irrational number multiplied by a rational coefficient as in previous cases, but including a set of constants that can each be multiplied by a coefficient.

We chose to extend with square roots because of their intensive use in several areas, especially those including models using geometry. To introduce these numbers, we need to solve three problems: i) finding how they interact with the previously existing ones, ii) finding how to detect if the numbers are

rational, and iii) finding how to detect if the result of an addition is rational. We will start discussing these problems for roots of integers, and will later generalize it for rational roots.

First, we know these new numbers do not conflict with previously introduced irrationals because they are non-transcendental. All numbers included before are transcendental, and it is impossible to obtain a non-transcendental number as the product of a rational by a transcendental irrational number.

Second, not every square root of an integer is irrational. We need to check if numbers being represented are irrational. For that purpose, we obtain the prime factorization of the number to be represented and check if any factor has an odd exponent; if this is the case, we are in presence of an irrational number, else we should derivate the representation to the rational component.

Obtaining prime numbers factorization is an expensive operation for large numbers. In some programming languages, most of the complexity introduced can be palliated at compile time. For example, using template meta-programming, the prime factorization of the numbers used can be pre-computed.

To represent square roots uniquely, we simplify the expression factorizing the represented number as the product of a rational by a square root of product of primes with exponent 1. The factors extracted outside of the square root are used as coefficient of the irrational constant introduced. In some cases, after simplifying if there is no irrational number at all and we move the coefficient to the rational part of representation. We show an example representing $\sqrt{72}$ in Formula 5.3.

$$\sqrt{72} = \sqrt{2^3 \cdot 3^2} = 3 \cdot 2 \cdot \sqrt{2} \Rightarrow n = \langle 0, \{\langle 6, \sqrt{2}\rangle\}\rangle$$

Formula 5.3: Representation for $\sqrt{72}$

In previous defined subsets $\langle \pi, e^{\pi} \rangle$ the digits-expansion function did not require any parametrization. Here, we are introducing a whole family of irrational constants that will be introduced on demand to the set. The parameter for defining these constants is the integer value in the radicand. We introduce a new pair coefficient and algorithm to the set of irrational constants (A) for each radicand used.

For comparisons we operate as before, we iterate the irrational constants and their coefficients, generating digits only when needed.

For adding, we have three scenarios to consider: first, when the addition produces a rational number; second when the addition produces a result in the same set of irrational square roots; and third, when the addition can not be mapped to any radicand in the set of irrational square roots.

First case is not possible. Addition of two square roots result is rational only using perfect square radicands, we show proof in Formula 5.4. Perfect square radicands are not irrational square roots. Then, they are not defined using irrational constants in our representation.

In second case, if we have addition of equal radicand square roots, we can operate with coefficients only. For example, $2 \cdot \sqrt{2} + 1 \cdot \sqrt{2} = 3 \cdot \sqrt{2}$.

In third case, it is possible to generate irrationals that are not square root of integers using addition. We show an example in Formula 5.5.

$$a, b \in \mathbb{Z}, c \in \mathbb{Q}$$
$$\sqrt{a} + \sqrt{b} = c \Rightarrow a + b + 2\sqrt{ab} = c^2$$
$$2\sqrt{ab} \in \mathbb{Q} \Leftrightarrow \exists d \in \mathbb{Z} : ab = d^2 \Rightarrow a = \frac{d^2}{b}, \frac{d}{\sqrt{b}} + \sqrt{b} = c \Rightarrow d + b = c\sqrt{b}$$
$$c \in \mathbb{Q} \Rightarrow \exists e \in \mathbb{Z} : b^2 = e$$
$$a = \frac{d^2}{b} \Rightarrow \exists f \in \mathbb{Z} : a^2 = f$$
`A and B have to be perfect squares`

Formula 5.4: Proof that addition of square root is rational only if radicands are perfect squares

$$\exists k \in \mathbb{Q} : \sqrt{2} + \sqrt{3} = \sqrt{k} \Rightarrow (\sqrt{2} + \sqrt{3})^2 = (\sqrt{k})^2$$
$$\Rightarrow 2 + 3 + 2 \cdot \sqrt{2 \cdot 3} = k \Rightarrow 5 + 2 \cdot \sqrt{6} = k$$
$$\Rightarrow \frac{k - 5}{2} = \sqrt{6}$$
`Absurd, because` $k \in \mathbb{Q} \wedge \sqrt{6} \notin \mathbb{Q}$

Formula 5.5: Proof that addition of square roots of integers can produce a new class of irrationals

If we do not include any other class of non-transcendental numbers representation, it is safe to operate in a component by component basis, adding matching radicand coefficients. In the case a new class is introduced, as grade 4 roots, special care has to be taken when operating to promote the addition result to the right family of values.

An extension to represent irrationals obtained as square roots of rationals is possible. For extending this way, we use now prime numbers with exponents 1 and -1. Any other exponent requires factorizing and compensating using the rational coefficients. In addition to previous concerns, we need to check what happens when these new irrationals are added. In Formula 5.6, we show proof that adding inverses never produces a rational number.

$$A \subset primes, B \subset primes, |A| < \infty, |B| < \infty, A \cap B = \emptyset, A \cup B \neq \emptyset, p, q, k \in \mathbb{Q}$$
$$a = \prod A \qquad b = \prod B \qquad k = p\sqrt{\frac{a}{b}} + q\sqrt{\frac{b}{a}}$$
$$k^2 = p^2\frac{a}{b} + q^2\frac{b}{a} + 2pq\sqrt{\frac{ab}{ba}} = p^2\frac{a}{b} + q^2\frac{b}{a} + 2pq\sqrt{1} = \frac{p^2a^2 + q^2b^2 + 2pqab}{ab}$$
$$\Rightarrow k^2ab = (pa + qb)^2 \Rightarrow k\sqrt{ab} = pa + qb$$
$$\texttt{absurd} : k\sqrt{ab} \notin \mathbb{Q} \texttt{ and } (pa + qb) \in \mathbb{Q}$$

Formula 5.6: Proof that adding square root of inverses is irrational

In Formula 5.7, we show proof that it is possible outcomes of adding irrational square roots of rationals is not representable as the square root of a rational number.

Similarly to the family of square roots of integers, if we do not include any other class of non-transcendental numbers representation, it is safe to operate in a component by component basis. In the

$$\sqrt{\frac{3}{2}}+\sqrt{\frac{5}{7}}=\sqrt{\frac{3}{2}+\frac{5}{7}+2\sqrt{\frac{15}{14}}}=\sqrt{\frac{31+2\sqrt{15.14}}{14}}=\sqrt{\frac{31+2\sqrt{210}}{14}}$$
$$\sqrt{210}\notin\mathbb{Q}\Rightarrow\frac{31+2\sqrt{210}}{14}\notin\mathbb{Q}$$

Formula 5.7: Proof that addition of square roots of rationals can produce a new class of irrationals

case a new class is introduced, as grade 4 roots, special care has to be taken when operating to promote the addition result to the right family of values.

5.4 Extending the datatype further

We have now basic support for irrational that may be extended or customized to support accurate simulation. The approach we discussed introduced a new subset of irrational numbers at each step. We cannot do this for every irrational. Not every subset is compatible with those we shown before.

For example, it is not proven that $\pi+e$ is irrational, and then we are not certain that we can use both at the same time safely. Two approaches can be taken, first we can remove π from the set of constants when using e; second we can set a limit for digits allowed to be generated, and trigger an error when an operation could not be decided after reaching the generating limit. We encourage the use of this limit even in case that it is theoretically safe.

A stronger property than the one we required named Algebraic Independence over $\mathbb{Q}$ is explained in [NP01]. Several irrational numbers have been proved to be Algebraic Independent over $\mathbb{Q}$; any of them can be used to extend our datatype.

The same reasoning presented for square roots can be applied to any other subset of irrationals that need to be added, for instance introducing cubic roots. We can use similar reasoning for extending the arithmetic operations provided by the datatype to something more than addition. We define them following the algorithms defined in [Abe01], and check the results are always part of the original set. In case the results are not part of the set, a new set needs to be defined dynamically, as we do when we add two irrational square roots not sharing the radicand.

5.5 Summary

In this chapter, we presented a method for supporting irrational time values in the timeline of a simulation. This values are commonly used in certain fields.

We described the limitations of previous approaches from the literature. And, we proposed a new datatype based in concepts from computable calculus and symbolic algebra. This new datatype has the limitation of not allowing arbitrary irrationals to participate in every operation. Only those covering a property described are allowed to interoperate.

The method does not introduce significant complexity unless generating digits for solving is a must during strict inequality operations. The complexity in those cases is related to the complexity of generating the digits.

We presented the four operations required for implementing a DEVS simulator (+, -, <, =). In addition, we explained how to extend to other operations. We introduced the subsets of multiples of π, e^{π}, and the square roots of integer and rational numbers. And, we finalized explaining how to extend the datatype to support further irrational values.

Chapter 6

Uncertainty and Discrete-Event Simulation (DES)

As discussed earlier, one of many usages of Discrete-Event Simulation (DES) is being part of decision-making processes for industrial and experimental research works. In these works, events are collected from real systems using measuring instruments, and procedures. The measurement results obtained could be used as input for feeding simulations.

The Bureau International de Poids et Mesures (BIPM) defines in [BIII08] the proper procedures to obtain measurement results and the basic concepts behind them. In metrology, it is impossible to determine a unique true value for a measurand [BIP08]. Every measurement result is a set of quantity values being attributed to a measurand together with any other available relevant information. Then, measurement results are generally represented as a single measurement quantity and a measurement uncertainty summarizing all the relevant information collected from the measurement process. These values can be combined to give an Uncertainty Interval and assign reasonable values to the measurand [BIP08] (we will not enter in the details of how the measurement result is obtained, but rather focus on how to use it for simulation).

In this chapter, we focus on studying the introduction of uncertainty on the time component of the events; we are interested in this because time is advanced using the simulator algorithms, and affects directly their definition. In the context of DES, we will consider the result of a simulation to be a trajectory of states, or outputs, describing the dynamic behavior of the model. The input for the simulation is a set of events, each event generally noted as a pair $\langle t, s \rangle$ describing a change of state in the system; the t component is the time at which the state is changed; and the s component is the new state reached.

In some cases, the state component can also have an associated uncertainty; if scheduling events in the timeline is independent of the uncertainty of the state, the state can be encoded as an interval (representing the uncertainty), and the model can have uncertain states without requiring any change to the simulation algorithms or the modeling language.

Mathematical tools exist for modeling and simulate Continuous Systems considering input uncertainty. The goals of these tools are to derive uncertainty intervals of the results. The obtained intervals correspond to the propagation of uncertainty in inputs through the simulation.

In the case of Discrete-Time Simulation (DTS), any finite period contains a finite number of time values. Thus, we can simulate using input with uncertainty in its time occurrence using brute-force. For this purpose, we simulate each value in the input's uncertainty interval. After simulating all possibilities, we combine the obtained results for knowing their uncertainty quantification.

None of these approaches can be used for simulating DES because a small change in the input can lead to a completely different trajectory. The Continuous Systems techniques cannot be used because, in general, DES cannot be translated into Continuous Systems, and operations from calculus, as min-max or derivatives, are undefined for them. The brute-force approach of DTS cannot be used neither, because in DES, the uncertainty interval could include infinite time values. Thus it is impossible to use a brute-force algorithm to explore all the results.

Therefore, we propose a method for the simulation of DES models in which input events can have uncertainty in their time of occurrence. Our method aims at obtaining all the possible trajectories considering such inputs. To the best of our knowledge, no previous research in DES has explored this way.

6.1 Method for handling uncertainty quantifications in DES

We have a special interest in models described using the Discrete-Event System Specification (DEVS) formalisms family for the reasons discussed in Chapters 1 and 2. Thus, we propose new abstract algorithms for simulating DEVS models. These algorithms can process external events having uncertainty quantification in their time of occurrence.

We describe the uncertainty in the time of occurrence of the events using intervals. Thus, we need interval operations in our algorithms for advancing the simulation. However, this extension does not affect the models specification. Therefore, we can reuse previously defined DEVS models in new contexts in which the information about the uncertainty is valuable, but was not taken into account.

In Chapter 3, we discussed how any change in the time of occurrence of events affects the resulting trajectories. We can associate to each possible occurrence of an event a branching point of execution in the simulation. These branching points form a tree-like summary of all the possible trajectories. In this tree, each path from the root to a leaf represents a trajectory corresponding to discrete inputs and their respective uncertainty intervals. Our simulation algorithms generate this kind of results.

Then, the introduction of a single event with uncertainty can produce infinite trajectories. The event in the timeline could be at any time on the interval, and even if it always reaches the same state, it could be at different (infinite) points of time. So, we need to analyze trajectories-trees when uncertainty intervals include an infinite number of values in $\mathbb{R}$. In those cases, the tree could have an infinite number of edges going out of a branching point, but these edges do not necessarily reach an infinite number of

states. Indeed, they could all reach the same finite set of states but at infinite different placements on the timeline.

For example, let us consider the case of a model for a non-Schedule-Preserving (non-SP) pedestrian traffic light. The simulation of this model can generate infinite number of trajectories per simulation step, but they all converge to reach the same state in a single step.

The following is the expected behavior of the model:

- The model state is defined by the tuple $\langle color, timeout \rangle$. The possible colors are RED, ORANGE, and WHITE. The timeout tells how long to wait before switching the color if no external event is received.
- When a pedestrian pushes the button while the traffic light is on RED, the color switches to ORANGE, and the timeout is set to 5 seconds.
- When a pedestrian pushes the button while the traffic light is on WHITE, it stays WHITE, and the timeout is set to 15 seconds.
- When a pedestrian pushes the button while the traffic light is on ORANGE, it stays ORANGE, and the timeout is set to 5 seconds.
- The initial state of the simulation is $\langle$RED, 45 sec$\rangle$.
- While the button is not pushed, the traffic light loops over RED, ORANGE and WHITE, respectively for 45, 5, and 15 seconds.
- The output is what the pedestrian sees. There are two possible outputs for this model, WALK and WAIT. When switching state from WHITE to RED, the output is WAIT. When switching state from ORANGE to WHITE, the output is WALK. Since ORANGE is only an intermediary state, switching to it does not generate any output.

In Algorithm 6.1, we show how the internal and external transitions can be implemented for this model.

In Figure 6.1, we show the state trajectory when pushing the button at exactly 1 second of simulation. In this case, the initial state $\langle red, 45 \rangle$ is switched to $\langle orange, 5 \rangle$ after a second, and then, states loop indefinitely following the predefined timeouts.

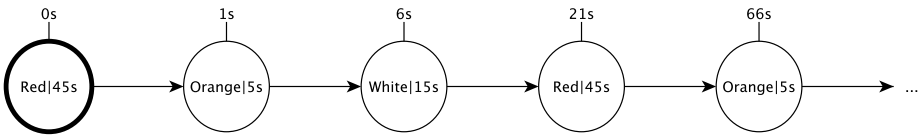

Figure 6.1: State trajectory of non-Schedule-Preserving (non-SP) pedestrian traffic light with input at exactly 1s

Introducing the button event at 45s, we obtain the same sequence of states showed in Figure 6.1, but with different occurrence times. Same happens for introducing the event at any time before 45s.

State: $\langle color, timeout \rangle, color \in \{red, orange, white\}$
Output: $y \in \{wait, walk\}$

Function δ_{int}*(State s)* $\rightarrow$ *State*
  if $s.color = red$ **then** return $\langle orange, 5 \rangle$;
  if $s.color = orange$ **then** return $\langle white, 15 \rangle$;
  if $s.color = white$ **then** return $\langle red, 45 \rangle$;
End

Function δ_{ext}*(State s, Time e, Message x)* $\rightarrow$ *State*
  if $s.color = white$ **then** return $\langle white, 15 \rangle$;
  if $s.color = orange$ **then** return $\langle orange, 5 \rangle$;
  if $s.color = red$ **then** return $\langle orange, 5 \rangle$;
End

Function λ*(State s)* $\rightarrow$ *Output*
  if $s.color = red$ **then** return ;
  if $s.color = orange$ **then** return $walk$;
  if $s.color = white$ **then** return $wait$;
End

Algorithm 6.1: Model of a non-Schedule-Preserving pedestrian traffic light

Introducing the event between 45 and 50 seconds produces a different sequence of states. In this case, introducing the event produces two consecutive $\langle orange, 5 \rangle$ states, the first at 45s, and the second at the event occurrence. And, introducing the event between 50 and 65 seconds produces two consecutive $\langle white, 15 \rangle$ states, the first at 50, and the second at the event occurrence.

If the event being introduced has uncertainty in its time of occurrence, we want to evaluate all reachable state sequences. For example, if we know the event could occur between 40 and 60 seconds, the simulation has to produce the three sequences of states mentioned before. In addition, we associate a time of occurrence interval to each state in the sequences. This intervals are subinterval of the external input occurrences.

The state trajectories of the simulation include, in addition to the sequence of states, the occurrence time of each state. In our example, we introduce an event at any time in the $[40, 60]$ interval. Choosing any value in the interval produces a different trajectory, since the time the event is introduced is the time the state is changed. Since the interval is in $\mathbb{R}$, it has infinite points, and then the number of trajectories is infinite. However, we can characterize every of them as representing one of the three state sequences and a time interval when state changes can occur.

Based on these concepts, we propose a new set of algorithms to deal with these issues. The main idea behind the algorithms is detecting the times when the same sequences of events are generated. The result is a set of continuous subintervals, which are derived from the uncertainty intervals of the events. For this purpose, we analyze the overlap of the uncertainty intervals occurring between pairs of input events, or between them and the schedule of the next internal transition. We do this because uncertainty can affect the schedule of the internal transition as consequence of a previous external one.

In addition to analyzing the overlapping of events, we need to check if the result of processing a transition is unique. For this purpose, we explore the definition of transition functions looking for continuous input intervals producing a common state. Each such interval determines a single sequence

of states. Again, having a sequence of states could be associated to infinite trajectories.

When using these algorithms, it is possible, for some combinations of model and input, that a step of simulation reaches an infinite number of states. As a consequence, it would require an infinite number of branches to be executed.

For example, the classic processor DEVS model described in [ZPK00] could produce a single sequence of states, or it could reach an infinite number of states in a single step of simulation, depending the input. The Processor model receives jobs to be executed; each job takes a fixed execution time. In case the processor is busy and a new job is received, the new job is queued until the current job is completely processed. After processing each job, an output is generated to inform the processing was completed. The state of this model can be represented by a pair (n, l) where n is the number of jobs to be processed and l the time left to finish processing current job. The external transition function for this model always increments n, the number of jobs by one; it uses a piece-wise function to set the value of l, the time left to finish: if the queue is empty it sets l to the fixed period declared in the model, else it set the new value of l to the difference between the previous value of l and the time elapsed since the last transition (parameter e of the external transition function). The internal transition, removes a job from the queue: it decrements n by 1, and if queue is not empty sets the time to finish current job to the period declared in the model, else it passivates the model.

For example, we introduce two jobs to a processor with a declared job processing time of 2s, first job at 1s, and second at 2s; both jobs occurrence with an uncertainty of ± 0.1s. Each occurrence uncertainty can be described as an interval, first one as $[0.9, 1.1]$, and second one as $[1.9, 2.1]$. When the first event is introduced, the state is changed to $\langle 1, 2\rangle$. After, when we introduce the second event, the first job is still being processed, and then we use the piece of δ_{ext} function dependent on the elapsed time. Computing the elapsed time is the difference of the interval of occurrence of previous event and the one being inserted, in this case $[0.8, 1.2]$. When evaluating $\delta_{ext}(\langle 1, 2\rangle, [0.8, 1.2], \text{QUEUE-JOB})$, we compute the new state as $\langle 2, x\rangle$ where x is the difference between 2 and the elapsed time e. Since, the x takes infinite values, $\langle 2, x\rangle$ takes infinite values, and we obtain an infinite number of states.

The model in previous example is not always producing infinite states. For example, if we delay the second input to happen after 2s, e.g. 5s, the job is finished at 3s, and when the second event is introduced, with the queue empty, will produce the single state $\langle 1, 2s\rangle$ again. Thus, in this case, we have a finite number of state sequences, but still infinite trajectories. This shows that, for some models, the input participate in deciding the finiteness of the generated state sequences.

Every simulation starts in a single state. A trajectory never branches until it receives an external event. We showed the introduction of an event with uncertainty produces infinite trajectories. These trajectories can share, or not, the state sequence. In the case, they do not share the state sequence, there is a point in the sequence were multiple states are reachable. At this point we can branch our sequence of states, producing a states-tree. We branch when uncertainty time intervals overlap, or when many states are produced by processing an event. For keeping track of every trajectory, we attach to each node of the tree the interval associated to its creation. This tree with annotations is useful for having a global visualization of all trajectories.

In Figure 6.2, we show an example using this visualization with the state sequences it represents.

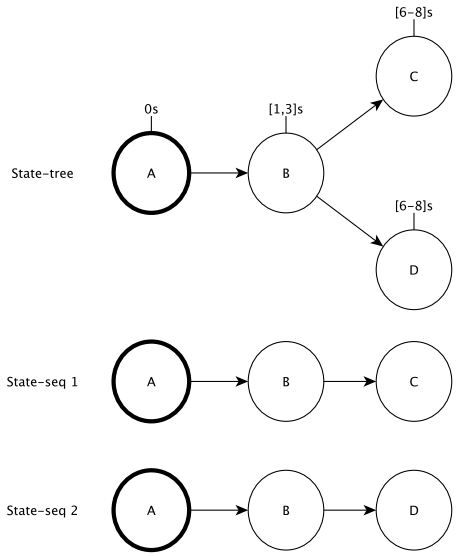

Figure 6.2: Example of a states-tree and its expansion to state-sequences

Here, the set of state sequences that could be obtained from the states-tree are two, State-seq 1, corresponding to the upper path of the tree, and State-seq 2 corresponding to the lower path of the tree. The intervals wrote over the nodes in the tree represent the uncertainty of the time of occurrence of those states. Each trajectory is described by one of the state sequence associated to any combination of time occurrences chosen from the annotated intervals. For example, following the state sequence $\langle A, B, C\rangle$ we could find the trajectories $\langle\langle A, 0s\rangle, \langle B, 2s\rangle, \langle C, 7s\rangle\rangle$, or $\langle\langle A, 0s\rangle, \langle B, 3s\rangle, \langle C, 8s\rangle\rangle$; and following the state sequence $\langle A, B, D\rangle$, we could find the trajectory $\langle\langle A, 0s\rangle, \langle B, 1s\rangle, \langle C, 6s\rangle$. The combination of time occurrences and states is infinite in this case and not all combinations are valid trajectories.

For all values in the occurrence uncertainty interval associated to a state, an event can be defined. We are certain this event belongs to at least one possible trajectory. However, obtaining a complete trajectory from the tree is not an easy task. Choosing an event from the tree constraints the selection of every other events participating in the trajectory.

For example, we define a model doing echo of input events one second later in the output. If we input an event at [0.9, 1.1] seconds, the model outputs same event at [1.9, 2.1] seconds. For every possible occurrence of the output event (in [1.9, 2.1]) there is a possible input triggering the output. Then a trajectory including every occurrence of the output exists. However, when choosing a particular event, e.g. the output at 2s, only one input participates of same trajectory (1s).

6.1.1 Abstract simulator for DEVS atomic models

In Algorithm 6.2, we present the Abstract Simulator for atomic models accepting events with uncertainty intervals for their time of occurrence. The simulator structure is similar to the Classical-DEVS abstract simulator presented in [ZPK00].

Data: Atomic model A
`// simulator variables:`
Interval t_{last} `// time of last event`
Interval t_{next} `// time of next event`
Atomic model A `// the simulated atomic model`
A::state s `// current state`

Function *init-message(Time t)* $\rightarrow$ *void*
  $t_{last} := [t, t]$
  $t_{next} := t_{last} + [A.ta(s), A.ta(s)]$
  return
End

Function **-message(Interval t)* $\rightarrow$ *Y*
  if $\neg(t \subseteq t_{next})$ **then** raise an error ;
  Y $y := A.\lambda(s)$
  $s := A.\delta_{int}(s)$
  $t_{last} := t$
  $t_{next} := t_{last} + [A.ta(s), A.ta(s)]$
  return y
End

Function *x-message(X x, Interval t)* $\rightarrow$ *void*
  Interval t_{local}
  $t_{local}.upperend := t.upperend - t_{last}.lowerend$
  $t_{local}.open\_upperend := t.open\_upperend \vee t_{last}.open\_lowerend$
  if $t \cap t_{last}$ **then**
    $t_{local}.lowerend := 0$
    $t_{local}.open\_lowerend = False$
  else
    $t_{local}.lowerend := t.lowerend - t_{last}.upperend$
    $t_{local}.open\_lowerend := t.open\_lowerend \vee t_{last}.open\_upperend$
  end
  Function $f(a) := A.\delta_{ext}(s, a, x)$
  Set⟨Interval⟩ $primg :=$ all pre-image pieces of $f(k), k \in t_{local}$
  forall the $p \in primg$ **do**
    FORK
    if *On forked child* **then**
      Time $v := \frac{p.lowerend + p.upperend}{2}$
      $s := f(v)$
      $t_{last} := (t_{last} + p) \cap t$
      $t_{next} := t_{last} + [A.ta(s), A.ta(s)]$
    end
    WAIT(every branch is created)
    EXIT
  end
  return
End

Algorithm 6.2: Abstract Simulator for atomic models allowing input with occurrence uncertainty

Internal variables

Every variable of the simulator representing time is an interval. Four properties define each interval. The properties are two numbers for the upper and lower end, and two booleans stating if each end is open or not. Real numbers are only used for calling the functions of the model. Exceptionally, the *init-message* function receives a real value by parameter to set the initial time.

The input for constructing a simulator is an atomic model, which is kept as reference in the variable A. Two internal variables for tracking the simulation time are defined, t_{next} and t_{last}, corresponding to the time of the last event processed by the simulator, and the next scheduled internal transition. Both time tracking variables are of type interval to include the uncertainty quantification that could be associated to them.

Initialization

The *init-message* function provides the means for initializing the simulation. It receives a real number in parameter t used for setting the t_{last} variable. We construct a closed interval assigning t to both endpoints. And, we set t_{next} to the addition of t_{last} to the result of executing the *time-advance* function of A.

Internal transitions

The **-message* function advances the simulation time when an internal transition occurs. The parameter of this function is an interval (t) corresponding to the time of occurrence of the transition. The time at which the internal transition occurred is saved in t_{last}.

The function first generates the output using the λ function of A. Then, it gets the new state of the model using the δ_{int} function. Then, it sets the t_{last}, and computes the new t_{next} using the *time-advance* function of A. Finally, it returns the output previously computed.

An external event being prepared for input can overlap its uncertainty interval with the scheduled internal transition. The information available in the **-message* context is not enough for detecting when this happens. Thus, we delegate the responsibility of solving this kind of conflicts to the caller (defined in Section 6.1.2).

The caller has to find subintervals having a single reachable state. A new branch is created for each subinterval found. Each branch gets passed the parameter t. The δ_{int} function used internally only needs t to update t_{last}.

External transitions

The *x-message* function processes the execution of an external event. For this purpose, it has to call the δ_{ext} function, which uses the elapsed time since last transition. The *x-message* function starts by saving

all the elapsed times requiring evaluation in the variable t_{local}. In this variable we assign the difference between t and t_{last}. This difference corresponds to the differences of every possible last event occurrence time to every possible current time.

Since the δ_{ext} function is based on the elapsed time in current state, which includes every possible time value in the continuous t_{local} interval, we need to explore every possible state reached for every possible time of occurrence of the event in the uncertainty interval, which could lead to a number of alternate trajectories in the simulation.

For studying all possible trajectories, we use a branching approach. A branch is created to simulate each trajectory.

The second step is defining a constrained version of the δ_{ext} function, named f, in which the state and message variables fixed to those passed by the parameter x, and current state. Afterward, if we restrict the domain of f to the uncertainty interval t, the image of f is the set of all reachable states after this simulation step.

Having the set of reachable states, we split the initial time interval into a set of segments. Each segment is such that it can only lead to a single state. We also put the constraint that the size of each segment is maximized and therefore no two contiguous segments can reach the same state. We call this set of segments the *pre-image*.

We fork the simulation for each subinterval in the *pre-image*. In each branch created, we set the corresponding subinterval as the t_{last} of the simulation, and the state as the one obtained by using any point in the interval for f. After all the branches have been created, the current branch is exited. This termination is safe because the combination of all the new branches explores all the values in the original uncertainty interval t.

Notice that in the algorithm, we use the middle position between the endpoints of the uncertainty subinterval as the elapsed time value passed to f. This value is chosen to avoid handling border cases and simplify the case of zero-length pseudo-intervals (intervals reduced to a single discrete point), but every value in the interval produces the same state.

It is possible, in case t_{last} and t overlap, that the addition of t_{last} and the time interval obtained in the pre-image for current branch extends after the original t interval. In this cases, the value assigned as new t_{last} for current branch needs to be restricted to the points intersecting with t.

6.1.2 Main loop for DEVS simulators handling uncertainty quantification

Here, we define the main loop for advancing a simulation fed with external events collected as measurement results and defined with time intervals. This loop runs until every input in the queue is consumed and the model reaches a passive state.

The input is an ordered queue of events. Each event contains a message and a time uncertainty interval. The message component is any element from the model's input set X, and the time uncertainty

interval is any non-empty interval in $\mathbb{R}^+$.

The input queue order is defined by the following criteria: the interval having a value lower than any value in the other interval goes before the other, and in the case of a draw, the one having a value greater than any value in the other goes after the other.

For example the interval $[1, 3]$ is before $[2, 4]$, because 1 is lower than any value in $[2, 3]$. And, $[1, 3]$ is before $[1, 4]$ because 4 is higher than every value in $[1, 3]$.

Every time main-loop iterates, we advance a simulation step. For advancing, we can call the *x-message* function, or the **-message* function. What we chose depends on the current schedule of internal transition, and the events in the input queue.

In some cases, it is easy to decide, for example if the queue is empty, and an internal transition was scheduled, we call the **-message* function. In other cases, overlaps and order of events can make the choice more difficult. Then, we classify each combination of current queue of input events, and the next scheduled internal transition, as belonging to any of three possible scenarios: *No-Collision*, *Input-Collision*, or *Scheduled-Collision*.

In the No-Collision scenario, a scheduled transition or an input event is ready to be processed, and its time uncertainty interval does not intersect with any other event. Formally, we characterize the scenario as shown in Formula 6.1.

$$No\_Collision(Simulator\ s, EventQueue\ q) \equiv q \neq \emptyset$$
$$\Rightarrow ((q.front.time \ll s.next \wedge (\forall x \in q : q.front.time \ll x.time) \vee s.next \ll q.front.time)$$

Formula 6.1: No-Collision predicate

Where ($\ll$), the "strictly before" operator is defined as: $a \ll b \equiv (a \cap b = \emptyset) \wedge a.upperend \leq b.lowerend$. This auxiliary predicate defines an order on time intervals, where an interval is only "strictly before" another if every value in the first interval is strictly lower than every value in the second one.

The *No-Collision* predicate is true if the queue of events is empty, or if the first event in the queue is scheduled strictly before any other (including the scheduled internal transition), or if the scheduled internal transition ($s.t_{next}$) is expected strictly before any event in the input queue.

For advancing a step of simulation under No-Collision scenario, we use the Algorithm 6.3.

Data: Simulator s, Queue⟨Event⟩ input
if $input \neq \emptyset \wedge input.front.time \ll s.t_{next}$ **then**
  $s.x\text{-}message(input.front.message, input.front.time)$
  $input.pop()$
else
  $s.*\text{-}message(s.t_{next})$
end

Algorithm 6.3: Main-loop advancing on a No-Collision scenario

In this algorithm, if the uncertainty interval of the first event in the input queue comes strictly before

the scheduled internal transition, we pass this event to the simulator using the *x-message* function and remove it from the queue of inputs to be processed; else we advance the simulation using the **-message* function for running the internal transition on its scheduled interval.

In the Input-Collision scenario, the first event in the input queue overlaps its time uncertainty interval with another event in the queue, or with the scheduled internal transition. In addition, at least one value in the uncertainty interval of the first event of the queue is before any value in the scheduled internal transition. Formally, we characterize the scenario as shown in Formula 6.2.

$$
\begin{aligned}
&Input\_Collision(Simulator\ s, EventQueue\ q) \equiv q \neq \emptyset \\
&\wedge (\exists p \in q.front.time, \forall r \in s.next : s.next \cap q.front.time \neq \emptyset \wedge p < r) \\
&\vee (size(q) \geq 2 \wedge (\exists x \in q, \exists p \in x.time, \forall r \in s.next : x \neq q.front \\
&\wedge q.front.time \cap x.time \neq \emptyset \wedge p < r)))
\end{aligned}
$$

Formula 6.2: Input-Collision predicate

The *Input-Collision* predicate is true if the input queue is not empty, and its first event overlaps its uncertainty interval with the scheduled internal transition one, and at least one possible occurrence time of the input is lower than any value in the scheduled transition. In addition, the predicate is true if there are at least two events in the queue of events overlapping their uncertainty interval, and there is at least one possible occurrence in the uncertainty interval of the first event that is lower than every value in the interval of the next internal transition that is scheduled.

For advancing a step of simulation under Input-Collision scenario, we use the Algorithm 6.4.

The algorithm has two parts, first it searches for an interval of time to advance the simulation in which there is only one conflict. We call this interval the *bound*. Second, the simulation is branched to explore the paths resulting from each conflict resolution.

If the conflict detected is between events in the input-queue, the *bound* is between the start of the first event in the input queue until the end of the event ending first in the queue or the start of the scheduled transition, whichever happens first.

For every event intersecting the *bound*, we branch the simulation, we adjust the event upper endpoint to be into the *bound* and we call the *x-message* function running it as the first received event in a new branch, then we remove the event from the input queue and adjust the lower endpoint of every other event intersecting the *bound* so their intervals do not start earlier than the executed event.

After every branch is created, we decide if it is possible to advance the simulation without introducing any event during the *bound* interval, this happens when every event uncertainty interval extends beyond current *bound*. If it is possible to advance without input any event, we maintain the current branch in same state and adjust the start of every event in the queue for being introduced in next simulation step. Otherwise, if it is not possible to advance without introducing at least one of the conflicting events, we exit current branch, since the created branches obtain all possible trajectories.

By slicing the timeline this way, we can study all permutation of event placements in overlapping

Data: Simulator s, Queue⟨Event⟩ input

`// simulator variables:`
Interval *bound* `// time slice for advancing simulation`
if $\exists x \in input : x \neq input.front \wedge x.time \ll s.t_{next}$ **then**
  Real $limit := x.time.upperend : (x \in input \wedge \forall y \in input, x.time.upperend \leq y.time.upperend)$
  $bound := [input.front.time.lowerend, limit]$
  $bound.open\_upperend := (\exists x \in input, bound.upperend = x.upperend \wedge x.open\_upperend)$
else
  Real $limit = s.t_{next}.lowerend$
  $bound := [input.front.time.lowerend, s.t_{next}.lowerend]$
  $bound.open\_upperend := \neg s.t_{next}.open\_lowerend$
end
$bound.open\_lowerend := input.front.time.open\_lowerend$
forall the $x : x \in input \wedge x.time \cap bound \neq \emptyset$ **do**
  FORK
  if *On forked child* **then**
    remove x from *input*
    forall the $y : y \in input \wedge y.time \cap bound \neq \emptyset$ **do**
      Interval $i := [max(x.time.lowerend, y.time.lowerend), y.time.upperend]$
      $i.open\_upperend := y.time.open\_upperend$
      if $x.lowerend = y.lowerend$ **then**
        $i.open\_lowerend := x.open\_lowerend \wedge y.open\_lowerend$
      else if $y.time.lowerend < x.time.lowerend$ **then**
        $i.open\_lowerend := x.open\_lowerend$
      else
        $i.open\_lowerend := y.open\_lowerend$
      end
      replace y on *input* by Event$(i, y.message)$
    end
    $s.x\text{-}message(x.message, x.time \cap bound)$
  else
    WAIT(every branch is created)
    if $\exists y \in input : y.time \subseteq bound$ **then**
      EXIT
    else
      forall the $y : y \in input \wedge y.time \cap bound \neq \emptyset$ **do**
        replace y on *input* by Event$(y.time \setminus bound, y.message)$
      end
    end
  end
end

Algorithm 6.4: Main-loop advancing on a Input-Collision scenario

intervals. The case of a scheduled internal transition is treated in a separate scenario because its scheduling is sensitive to the introduction of external events, given that the input of an external event changes the state used for computing the time-advance required for scheduling the next internal transition.

In the Scheduled-Collision scenario, the uncertainty interval of the first event in the queue overlaps with the one for the scheduled internal transition. In addition, there is at least one value in the uncertainty interval of the scheduled internal transition before any value in the first event of the input queue. Formally, we characterize the scenario as showed in Formula 6.3.

$$Scheduled\_Collision(Simulator\ s, EventQueue\ q) \equiv q \neq \emptyset$$
$$\wedge\, q.front.time \cap s.next \neq \emptyset \wedge (\forall x \in q, \exists p \in s.next, \forall r \in x.time : p \leq r)$$

Formula 6.3: Scheduled-Collision predicate

The *Scheduled-Collision* predicate is true if the input queue has at least one event, the first event intersects its uncertainty interval against the scheduled internal transition interval ($s.t_{next}$), and there is at least one value in the interval of the scheduled internal transition that is lower than any value in the uncertainty interval of any event in the input queue. It is enough to check if the scheduled internal transition intersects with the first event in the queue, we know the order of the queue ensures that not intersecting the first event in the queue is equivalent to not intersecting any other event in the queue when the scheduled input transition uncertainty interval has values before the first event in the queue.

For advancing a step of simulation under *Scheduled-Collision* scenario, we use the Algorithm 6.5.

The structure of this algorithm is similar to Algorithm 6.4 we first look for a *bound*, and then we branch the simulation.

After the *bound* is found, a new branch of the simulation is created for each event in the queue intersecting the bound interval; the upper endpoints of these events are adjusted to fit into the bound interval. On each branch, the event that leads to the branching is executed using the *x-message* function, and the lower endpoint of all the remaining events is adjusted to not be before the executed event.

After all branches are created, we continue the execution of the current one running the scheduled transition using the **-function* in the child simulator, and the bound as t parameter. Here, t is being adjusted for covering the responsibility delegated by Abstract Simulator of safely partitioning the scheduled internal transition uncertainty intervals before calling the **-function*, as mentioned in Section 6.1.1.

For obtaining the *bound*, the lower endpoint of the scheduled transition is used as lower endpoint; and for the upper endpoint, we use the upper endpoint of the event in the queue ending first, or the upper endpoint of the scheduled transition, whichever happens the first.

For combining the three algorithms proposed for the three scenarios, we provide in Algorithm 6.6 the *main-loop*.

The *main-loop* receives two parameters, a model and a queue of events to be fed in. A simulator for the model is created (s), and the loop is iterated until the queue of input is empty and the model is passivated.

Data: Simulator s, Queue⟨Event⟩ input

`// simulator variables:`
Interval *bound* `// time slice for advancing simulation`
if $\exists x \in input : x.time \subseteq s.t_{next} = x.time$ **then**
  Real $limit := x.time.upperend :\ (x \in input \wedge \forall y \in input, x.time.upperend \leq y.time.upperend)$
  $bound := [s.t_{next}.lowerend, limit]$
  $bound.open\_lowerend := s.t_{next}.open\_lowerend$
  $bound.open\_upperend := (\exists x \in input, bound.upperend = x.upperend \wedge x.open\_upperend)$
else
  $bound := s.t_{next}$
end
forall the $x : x \in input \wedge x.time \cap bound \neq \emptyset$ **do**
  FORK
  if *On forked branch* **then**
    remove x from *input*
    forall the $y : y \in input \wedge y.time \cap bound \neq \emptyset$ **do**
      Interval $i := [max(x.time.lowerend, y.time.lowerend), y.time.upperend]$
      $i.open\_upperend := y.time.open\_upperend$
      if $x.lowerend = y.lowerend$ **then**
        $i.open\_lowerend := x.open\_lowerend \wedge y.open\_lowerend$
      else if $y.time.lowerend < x.time.lowerend$ **then**
        $i.open\_lowerend := x.open\_lowerend$
      else
        $i.open\_lowerend := y.open\_lowerend$
      end
      replace y on *input* by Event$(i, y.message)$
    end
    $s.x\text{-}message(x.message, x.time \cap bound)$
  else
    WAIT(every branch is created)
    $s.*\text{-}message(bound)$
  end
end

Algorithm 6.5: Main-loop advancing on a Scheduled-Collision scenario

Data: Atomic model *top*, Queue⟨Event⟩ input, Real t_{init}
Simulator s `// the simulator of the model being simulated`

$s := Simulator(top)$
s.init-message(t_{init})
while $input \neq \emptyset \vee s.t_{next} \neq \infty$ **do**
  if $No\_Collision(top.t_{next}, input)$ **then** Check Algorithm 6.3;
  if $Input\_Collision(top.t_{next}, input)$ **then** Check Algorithm 6.4;
  if $Scheduled\_Collision(top.t_{next}, input)$ **then** Check Algorithm 6.5;
end

Algorithm 6.6: Main-loop for coordinating simulation using measured input

Per iteration, a scenario is chosen using the defined predicates and the corresponding algorithm is used for advancing a simulation step. The three algorithms used for advancing the simulation always consume at least one event.

However, to ensure that the simulation does not stale and produces the correct result, we need to prove that every combination of input and state matches one and only one predicate for detecting the scenario.

We propose then the following theorems:

- Theorem 1: For any simulation, every combination of input and states renders true at least one of the predicates: *No-Collision*, *Input-Collision*, or *Scheduled-Collision*.
- Theorem 2: For any simulation, every combination of input and states renders true at most one of the predicates: *No-Collision*, *Input-Collision*, or *Scheduled-Collision*.

For proving Theorem 1, we want to show the following first order logic expression in Formula 6.4 is valid.

Theorem 1:

$$\forall s, \forall q : No\_Collision(s, q) \vee Input\_Collision(s, q) \vee Scheduled\_Collision(s, q)$$

Formula 6.4: Every simulation step is characterized by at least one scenario (Theorem 1)

This equivalent to prove Formula 6.5.

Theorem 1 (expanded):

$$\begin{aligned}
&\forall s, \forall q : \\
&(q \neq \emptyset \Rightarrow ((q.front.time \ll s.next \\
&\wedge (\forall x \in q : q.front.time \ll x.time) \vee s.next \ll q.front.time)) \\
&\vee (q \neq \emptyset \wedge (\exists p \in q.front.time, \forall r \in s.next : s.next \cap q.front.time \neq \emptyset \wedge p < r) \\
&\vee (size(q) \geq 2 \wedge (\exists x \in q, \exists p \in x.time, \forall r \in s.next : x \neq q.front \wedge q.front.time \cap x.time \neq \emptyset \wedge p < r)))) \\
&\vee (q \neq \emptyset \wedge q.front.time \cap s.next \neq \emptyset \wedge (\forall x \in q, \exists p \in s.next, \forall r \in x.time : p \leq r))
\end{aligned}$$

Formula 6.5: Every simulation step is characterized by at least one scenario (Theorem 1 expanded)

We break the proof in three cases: the input queue is empty; the input queue has a single element; the input queue has at least two elements.

In the first case, if the input queue is empty then *No-Collision* is true and as consequence the disjunction of the three predicate is true.

In the second case, if the input queue has a single element, it either intersects with the scheduled transition or not. In case they intersect, if the interval of the event in the queue includes earlier values than any value in the scheduled transition interval, the *Input-Collision* is true; else the *Scheduled-Collision* is true. In case they do not intersect, *No-Collision* is true because one interval comes strictly

before the other. As consequence the disjunction of the three predicate is true for every input queue having a single event.

In the third case, the input queue has at least two events and,

- *No-Collision* is true either if the scheduled internal transition comes strictly before the first event in the input queue, or if the first event in the input queue comes strictly before the scheduled internal transition and every other event in the input queue.
- If the scheduled internal transition intersects with the first element of the queue, and its interval starts before or simultaneously to the first element in the queue, *Scheduled-Collision* is true because the queue order guarantees the lowest value in the uncertainty interval of the first event of the input queue is lower than or equal to every value in the uncertainty interval of any other event in the queue.
- If the scheduled internal transition intersects with the first element of the queue, and its interval starts after the interval of the first event in the queue starts, or if the scheduled internal transition does not intersect the first element, but first two elements of the queue intersects, *Input-Collision* is true.
- Then, the disjunction of predicate is true for every input queue having more than one element.

Adding the three cases, we can say the disjunction of the three predicates is true for every possible $\langle s, q \rangle$, proving the Theorem 1.

For proving Theorem 2, we want to show the Formula 6.6 cannot be satisfied.

$$\begin{aligned} &\exists s, \exists q, (No\_Collision(s,q) \wedge Input\_Collision(s,q)) \\ &\vee (No\_Collision(s,q) \wedge Scheduled\_Collision(s,q)) \\ &\vee (Input\_Collision(s,q) \wedge Scheduled\_Collision(s,q)) \end{aligned}$$

Formula 6.6: A simulation step can be characterized by more than one scenario ($\neg$ Theorem 2)

To prove Theorem 2, we show that never two predicates are True at the same time, then the algorithm has no ambiguity about what case to chose in each simulation step.

Here, the predicate is true if there exist a combination of simulation (model and state) and input queues, for which at least two predicates are simultaneously true from the following: *No-Collision*; *Input-Collision*; and *Scheduled-Collision*.

Proving this cannot be satisfied means no such a combination exist, and therefore every combination of simulation and input queues renders true at most one predicate.

We break in three cases, assuming *No-Collision* is true, assuming *Input-Collision* is true, and assuming *Scheduled-Collision* is true.

In the first case, if *No-Collision* is true, it could be because the input queue is empty or because the expression $q \neq \emptyset \wedge ((q.front.time \ll s.next \wedge (\forall x \in q : q.front.time \ll x.time) \vee s.next \ll q.front.time)$

is true.

If the queue is empty, the expression being proved is false because *Input-Collision* and *Scheduled-Collision* both require the input queue not to be empty for being true.

If the queue has events, *Scheduled-Collision* and *Input-Collision* are false, because there is no intersection between the first event in the input queue and the second, or because the scheduled internal transition comes strictly before any event in the input queue.

In the second case, if *Input-Collision* is true, it could be because the first two events in the input queue intersection includes values before the scheduled transition. Other option is that first event intersects with the scheduled transition, and the input event interval starts earlier than the scheduled internal transition one.

If the first two events intersect in the input queue before the scheduled transition interval starts, *No-Collision* is false because the queue is not empty and there exist and intersection of events in the input queue before the start of the scheduled transition interval; and the *Scheduled-Collision* is false because the first event in the input queue has earlier values in its interval than those in the scheduled internal transition one.

If the first event intersects the scheduled internal transition, *No-Collision* is false because of it; and *Scheduled-Collision* is false because first event in the input queue has earlier values in its interval than those in the scheduled transition one.

In the third case, if *Scheduled-Collision* is true, *No-Collision* is false because of the intersection between the first event in the input queue and the scheduled transition; and *Input-Collision* is false, because there is intersection between the first event of the input queue and the scheduled transition, but no value in the interval of the first event of the queue is earlier than every value in the scheduled internal transition one.

Adding the results from the three cases, we obtain that there does not exist a combination of a simulation (in a particular state), and an input queue rendering two predicates true simultaneously.

As corollary, from proving Theorems 1 and 2, we can state that one and only one scenario is chosen per simulation step.

6.2 A subclass of DEVS for preventing infinite branching of the simulation

We presented algorithms in Section 6.1 for simulating all possible trajectories for a DEVS model when the external events introduced have uncertainty quantification in the time component. And we showed an example of how certain combinations of models and input can require infinite branches for advancing a single step of simulation.

In this section, we present Finite-Forkable DEVS (FF-DEVS) [VDW15], a subclass of DEVS models guaranteed to be simulated for any input, with the algorithms presented in Section 6.1, requiring only finite branches per simulation step.

For obtaining the subset of models we study the branching points of the proposed algorithms.

In the case of the main-loop, the branching points are placed in Algorithms 6.4, and 6.5, used for simulating the branches of multiple events that overlap their intervals with the next event to be processed. Here, each algorithm branches at most one branch per conflicting event in the input. Assuming we never introduce infinite conflicting events, we always create finite branches per simulation step. This is a reasonable assumption since input is collected from real systems.

In the case of the abstract simulator, the only branching point in the algorithms is in the *x-message* function. The time component of the input event being an uncertainty interval could provide infinite continuous values that need to be evaluated by the external transition to obtain all possible states.

In the *x-message* function, the algorithm breaks a given continuous interval into smaller subintervals for which the image of the external transition function is a unique state value, and creates a new branch per subinterval. We want the number of branches required for advancing the simulation to be finite, as consequence, we want these subintervals to be finite.

The first restriction to apply is including in the subclass only models where the δ_{ext} function produces finite states for a given current state, an input message and any finite uncertainty interval of elapsed time.

However, this restriction is too weak to satisfy the requirement that the number of branches of the simulation per simulation step is finite. It is possible for a δ_{ext} function to reach only two states, but having infinite intervals of each one in a finite interval. This should not to be confused with a Zeno problem state, which is not possible for DEVS models. Here, each simulation branch is effectively advancing, but it is not possible to have finite intervals for the t_{last} variable tracking when the simulation is advanced.

The number of reachable states of a δ_{ext} function does not limit the number of subintervals in a pre-image of a derived f. For example, in Formula 6.7, we show a δ_{ext} function having only two possible states to reach.

$$\delta_{ext}(s, e, x) = \begin{cases} 1 & \texttt{if } e \texttt{ is irrational} \\ 2 & \texttt{otherwise} \end{cases}$$

Formula 6.7: A two states δ_{ext} with infinite discontinuities

However, every f derived from this δ_{ext} function has infinite number of subintervals in its pre-image when evaluated in $[0, 1]$.

Using this restriction alone, and using any interval including all the intervals producing each state, we could modify the algorithms to advance the simulation obtaining a superset of the trajectories of the model.

For obtaining the exact set, we have to add the restriction for the states being reached from finite subintervals of the elapsed time interval. Then, we restrict the subclass to those models for which evaluating the δ_{ext} function in every state and input message combination, describes a constant piece-wise function with finite steps for every finite interval of elapsed time. In the context of this thesis, we consider the empty function is not a piece-wise function.

This restriction does not prevent the models from producing infinite sequences of states in an infinite trajectory. For example, a model counting transitions can be modeled as an integer counter and an external δ_{ext} function adding one to the counter in each call, this model produces infinite states, but it will never produce more than one in a single step of simulation.

It is important to notice that branching the simulation finite times is not the same as having finite trajectories. FF-DEVS models are allowed to describe infinite trajectories, similar to Classical-DEVS models.

Formally, we say a FF-DEVS atomic model is the tuple $A = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$ where:
S is the set of states,
X is the set of input values,
Y is the set of output values,
$\delta_{int} : S \to S$ is the internal transition function,
$\delta_{ext} : Q \times X \to S$ is the external transition function where: $Q = (s, e) : s \in S, 0 \leq e \leq ta(s)$ is the total set, e is the time elapsed since last transition, and $\forall s' \in S, \forall x' \in X, \forall k < \infty^+ : e < k \to \delta_{ext}(s', e, x')$ is a constant piece-wise function described by finite pieces,
$\lambda : S \to Y$ is the output function, and
$ta : S \to \mathbb{R}^+$ is the time-advance function.

Notice that any model defined in FF-DEVS is necessarily a DEVS model and it requires no change at all in its definition to be used as such. The other way around does not work and we will show examples of DEVS models in co-FF-DEVS later in this section.

We propose FF-DEVS coupled models being any DEVS coupled model coupling only FF-DEVS models. We want then to be certain we could transform any FF-DEVS coupled model into a FF-DEVS atomic equivalent.

We know DEVS models are closed under coupling, meaning that for every coupled DEVS model it exists an atomic DEVS model reproducing the same dynamic behavior. Given that FF-DEVS models are valid DEVS models, we know then that a single DEVS atomic model can represent any coupled FF-DEVS model.

For asserting the FF-DEVS subclass is also closed under coupling, we need to validate that the atomic model produced by the closure is a valid FF-DEVS atomic model.

We can see the state of the equivalent atomic model of a coupled FF-DEVS model as the tuple having in each position a corresponding state to the one of each atomic model in the original model. The external transition function of the equivalent atomic model can work as a two step process, first processing the routing of the input to know which state components will be modified and then calling

the corresponding original external transition functions to modify each of the state components.

Since DEVS and FF-DEVS specification of atomic models are so similar, we can trivially assert every item in the definition tuple, but the δ_{ext} function, is correct specification of a valid FF-DEVS model. For the δ_{ext} function, we need to check the δ_{ext} function of the atomic equivalent model still covers the restriction of being described by a set of piece-wise constant functions, each of them corresponding to a combination of current state, input message, and the functions having finite steps for every finite interval of elapsed time.

In fact, the new δ_{ext} function (δ'_{ext}) has constant piece-wise behavior too for being a composition of constant piece-wise functions.

The quantity of pieces of the atomic equivalent may be larger than any function from any participant model when the pieces endpoints do not match as the models presented in Formula 6.8.

$$\texttt{Model A} : \delta_{ext}(s,e,x) = \begin{cases} 1 & 0 \leq e \leq 1 \\ 2 & 1 < e \end{cases}$$
$$\texttt{Model B} : \delta_{ext}(s,e,x) = \begin{cases} 1 & 0 \leq e \leq 2 \\ 2 & 2 < e \end{cases}$$

Formula 6.8: External transition functions of the example atomic models A and B

A composed version, where both models (A and B) are connected to the EIC and no internal couplings are defined would be the model shown in Formula 6.9.

$$\texttt{Model C} : \delta_{ext}(s,e,x) = \begin{cases} \langle 1,1 \rangle & 0 \leq e \leq 1 \\ \langle 1,2 \rangle & 1 \leq e \leq 2 \\ \langle 2,2 \rangle & 2 < e \end{cases}$$

Formula 6.9: External transition function of model C, a closure equivalent atomic model

While the original δ_{ext} functions have each a max of two intervals of pre-image, the new composed δ'_{ext} function has the possibility of three. We can bound the quantity of intervals produced by the composition, and then since they are constant piece-wise and finite when evaluated in a finite interval, we can say the atomic equivalent covers the requirements to be a FF-DEVS model too.

To get a better idea of the practical limitations of FF-DEVS modeling, we will present a list of classical models used in the literature and discuss for each of them if they belong or not to the FF-DEVS models subclass.

- Passive [ZPK00]: This model never produces an output. Since an output is never produced, there is no need to have more than a single state. If the State set has a single element, the external function has to be a constant function. A constant function (a constant piece-wise function of a single piece) covers the requirements to be in FF-DEVS.
- Storage [ZPK00]: This model is used to simulate a memory cell capable of storing a single value. The model can receive two kinds of events, one asking for the last stored value and the other asking to store a new value. In case a request for the stored value is received, the state is changed

to a new one having the value to be output and the next internal transition is scheduled after a zero-time delay. In case a store event is received, the value in the message is stored as the state of the model and the model becomes passive. The processing of both types of events is independent of the elapsed time variable. Thus, all the f functions derived from the δ_{ext} function are constant functions, which covers the requirements for belonging to FF-DEVS.

- Counters: binary counter, n-ary counter, and infinite counter are described in [ZPK00].
  - The n-ary counter only outputs a predefined message after receiving n messages. To do so, the state represents a counter between zero and N. Every time a message is processed by the external transition, the counter is incremented. If the current state is below N, the model is passive, and when it reaches N next internal transition is scheduled after a zero-time delay. The internal transition resets the counter to zero before the output is sent. In any case, the external function is independent of the elapsed variable; therefore the model belongs to the FF-DEVS class.
  - The binary counter is a particular case of the n-ary counter, on which n is equal to two.
  - The infinite counter is similar to the storage model, in the sense it can receive two kinds of events, one incrementing the counter and the other asking for the current count and resetting the counter. Here, the counter state is a pair of current count and if the value was requested, when the value is requested a zero-time transition is scheduled to output and reset the counter. This model is also a FF-DEVS model for the same reason as previous ones, external transition is independent of the elapsed time variable.

- Accumulator: This is a slight variation of the infinite counter model, in place of counting the messages, it receives messages with numbers and it sums them into the current state. This more interesting model to study is also into the FF-DEVS models class.

- Generator [ZPK00]: This model is used to generate outputs of predetermined values at fixed points of time. The model input set is empty, and then the external transition function domain is empty. Since the empty function is not a piece-wise function, this is not a valid FF-DEVS model. An alternative definition of the Generator model is receiving input and ignoring it. In this case, the external transition function reschedules the next internal transition at the same absolute time it was scheduled before, but to do so, the time-advance needs to be defined as the difference between the previous time-advance and the elapsed time. After fixing the state and the input in the external transition function, the function is linear in the elapsed time, which is not a suitable behavior for a valid FF-DEVS model. Allowing this kind of behavior would require infinite forks if using our algorithms.

- Processor [ZPK00]: This model receives jobs to be performed; processing each job takes a fixed period. In the case, the processor is busy and a new job is received, the new one is queued until current one is completely processed. After processing each job, an output is generated to inform the processing was completed. The external transition function for this model is also linear in the elapsed time for some combinations of input and state values. Here, when the queue is not empty, the introduction of a new job requires the use of the difference between current time-advance and elapsed time as in the generator. Therefore, this model does not belong to the FF-DEVS class.

DEVS Model	FF-DEVS	FD-DEVS	SP-DEVS
Passive	YES	YES	YES
Storage	YES	NO	NO
n-ary counter	YES	YES	NO
Binary counter	YES	YES	NO
Infinite counter	YES	NO	NO
Accumulator	YES	NO	NO
Generator	NO	YES	YES
Processor	NO	NO	NO
SP-DEVS traffic light	NO	YES	YES

Table 6.1: Summary of DEVS models and the DEVS subclasses they belong to

- Schedule Preserving DEVS (SP-DEVS) Traffic Light [HC04]: This model represents a traffic light with a pedestrian call button that never changes a scheduled internal transition. When the button is pressed by, the event is registered into the state and when the previously scheduled internal transition is reached, its value is used for deciding the next state, for example to decide next light to be turn on. This is similar to what is done in the previous two examples, and has the same characteristic of being linearly dependent on the elapsed time for external transitions, which means this model does not belong to the FF-DEVS class.

In Table 6.1, we show the summary of which models are included in each of the following subclasses of Discrete-Event System Specification (DEVS): Finite-Forkable DEVS (FF-DEVS), Finite & Deterministic DEVS (FD-DEVS), and Schedule Preserving DEVS (SP-DEVS).

Comparing with other subclasses of DEVS, we can say that FF-DEVS is not strictly included in SP-DEVS [HC04] or Finite & Deterministic DEVS (FD-DEVS) [HZ06] since neither of them include models with infinite set of states as the infinite counter included in FF-DEVS. Also, we can say that neither SP-DEVS nor FD-DEVS are strictly included in FF-DEVS since we can find models, like the Generator, that are included in both of them but not in FF-DEVS. However, the intersection between these classes is not empty. Some models, like the binary-counter for example, belong to both FF-DEVS and FD-DEVS, while some others, like the Passive model, belong to all three of them.

6.3 Study of trajectories with uncertainty quantification of a FF-DEVS model

In Section 6.1, we presented the non-Schedule-Preserving (non-SP) traffic light model and discussed the different trajectories-tree results expected according to different inputs proposed. We show now how our algorithms obtain the expected trajectories-tree for this model in the particular case of pressing the call button two times, at 2 and 8 seconds with an uncertainty of ± 1 second, meaning the events could be input at any time between 1 and 3 seconds and 7 and 9 seconds from starting the simulation.

In Figure 6.3, we show the state, and output trajectories-tree obtained by the simulator.

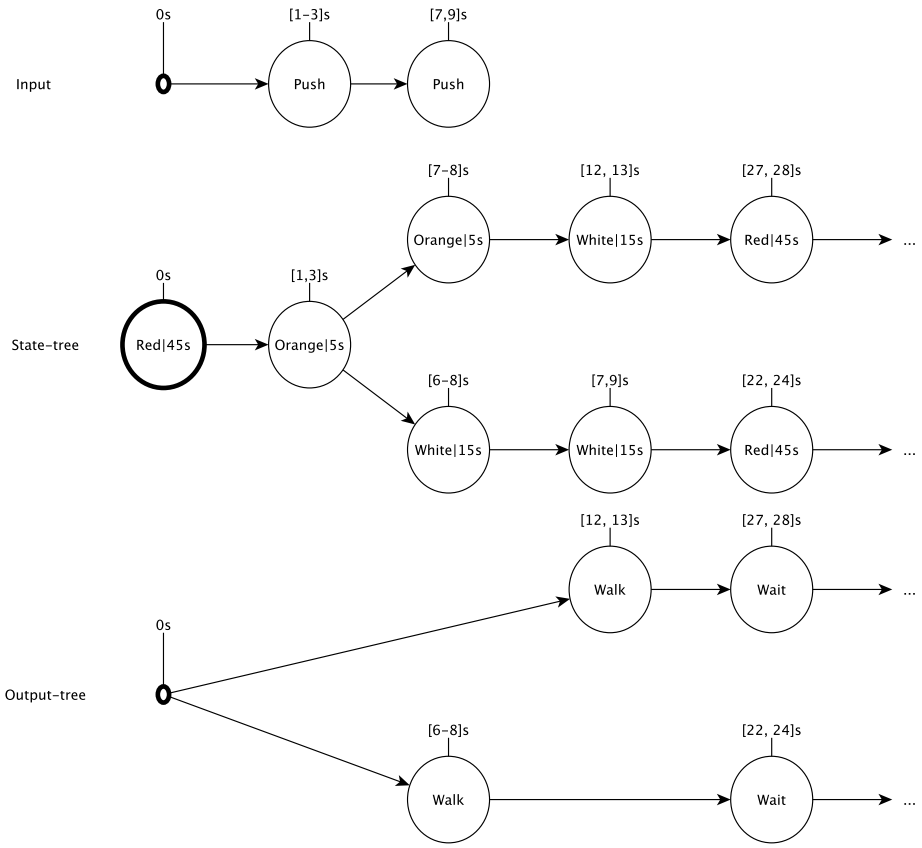

Figure 6.3: Example of Output and State trajectories-tree of a non-SP pedestrian traffic light

In the example, we introduce the first event at $[1, 3]$ seconds; this is in the scenario of a *No-Collision*, since the other event in the queue, and the scheduled internal transition (45s) both are strictly after this event. At this point we already have infinite resulting trajectories, one for each possible placement of the event in the interval $[1, 3]$, but at this point we do not require any branching of the simulation.

The next event process could be the scheduled internal transition at $[6, 8]$, or the next event in the input queue at $[7, 9]$, this is the scenario of a *Scheduled-Collision*. Here, we branch our simulation for exploring the possibility of the input event being executed first (possibly) rescheduling the internal transition (upper branch), or the internal transition being executed first (lower branch).

In the upper branch, the Orange state is renewed, and the simulation delays the output of the Walk light until $[12, 13]$ seconds, while in the lower branch, the state is changed to white, outputs a Walk light, and the input makes it extend the time it is staying on this state until $[22, 24]$.

After being in walk state, both branches go to the red state and output Red light, the upper branch does it at $[26, 28]$ seconds, and the lower one at $[22, 24]$ seconds, afterward both branches loop between the three states indefinitely.

Here, we can answer decision questions based in the obtained trajectories-trees. For example, we can ask what light is on at 14s. The answer is Walk, because in both branches of the simulation, we output

Walk before 14s and not change back to Wait happens in any branch before 22s.

6.4 Summary

In this chapter, we presented a method for using measured events as input to the simulation of DEVS models.

The method detects branching points in the sequence of states, and branches the simulation for exploring them all. The proposed algorithms keep track of every possible occurrence time of those branching points. Combining the sequence of states and the occurrence times, trajectories-trees are constructed. These trees describe every possible State and Output trajectories corresponding to the possible placement of the input events with uncertainty.

The method presented is not suitable for every combination of DEVS model and input. In some cases, it may require infinite branches in a simulation step. We proposed a subclass of DEVS, which every model could be simulated with finite branches per simulation step for any input. We called this subclass Finite-Forkable DEVS (FF-DEVS).

We compared this new class to other classes in the bibliography. The FF-DEVS class is not included in FD-DEVS, neither in SP-DEVS. The FD-DEVS class is not included in FF-DEVS, neither is the SP-DEVS included in FF-DEVS.

Chapter 7

A new sequential architecture for Parallel-DEVS simulators

In previous chapters, we described new datatypes and algorithms proposed for Discrete-Event Simulation (DES). We want to study the performance impact of their use in multiple formalisms based on Discrete-Event System Specification (DEVS), empirically.

In this regard, we define a simulator architecture in which it would be easy to add and remove any of the proposed changes. This architecture allows users to easily switch the formalism used. We want to reuse the same architecture and code base as much as possible, in order to reduce the bias introduced by implementation details during experimentation. In addition, we want to ensure that an execution is repeatable to simplify the analysis of any unexpected result.

7.1 Sequential algorithms for PDEVS

To produce repeatable executions without adding complexity, we choose a sequential simulation approach for the architecture based on the Parallel-DEVS (PDEVS) formalism. In Chapter 2, we described the abstract algorithms defined to execute PDEVS models. The Abstract Simulator uses two kinds of components to run the simulation: Simulators in charge of simulating atomic models and Coordinators in charge of simulating coupled models. The mapping between models and simulation components is one to one.

The approach used in the PDEVS abstract algorithms' follows a parallel design: all the components are considered to be running and waiting for a message at the start. The Root Coordinator starts the simulation by sending an advance request to all of its children (Coordinators and Simulators) and waits until a response is received from all of them. Each Coordinator receiving one of these messages does the same: it sends the message to each of its children and waits for the replies (done and output messages). When a request reaches a Simulator, it runs sequentially the simulation of its atomic model

and sends results to its parent Coordinator. Once all the replies are collected at the Root level, the Root Coordinator sends the messages to advance the simulation once again, until the end-time is reached or the simulation reaches a passive state.

To provide a sequential, single threaded Coordinator, we have removed the synchronization sets, simultaneous executions, and locks, defined in the abstract algorithms. We have also replaced the top down asynchronous message passing with synchronous calls to functions and the bottom up (asynchronous) response messages with return to functions called. An additional benefit of this approach is that children are not aware of their parent coordinators. Likewise, we are sure that the call hierarchy is limited to a fixed value that is linear in the height of our model hierarchy since calls are only initiated by parent coordinators and they only happen in a top down fashion.

Because multiple message types are supported, replies can carry more than the confirmation of execution. We have encoded the returned message type so that different kind of messages – done and output messages – can be received from the children. These message types are identical to those described in Chapter 2. Afterward, we use the message type to process the reply and to route messages accordingly. For instance, if a child sends an output message, the parent will pass this message to all other children that are internally coupled to the one that generated this output. These messages are received from the parent through two functions: advance-simulation and collect-outputs as shown in Algorithm 7.1.

We added a structure to each Coordinator called *inbox*. The *inbox* collects the messages returned by *collect-output*. After all messages are in distributed to each *inbox*, the next call to *advance-simulation* is used to process them. This procedure is safe because we know the order these functions are called is fixed by the Root Coordinator main loop.

In *collect-outputs*, the coordinator verifies if it has reached its next state change time. If not, an empty reply is sent; otherwise, the outputs of each imminent coordinator are collected and added to the output bag. Hence, all the *Y-messages* are collected first and then sent together.

For *advance-simulation*, the time t is verified to ensure that it is between the last change and the next scheduled change. This variable is assigned as the last executed event time, and then the *external-imminent* set is defined with the models receiving an input event. This is done by adding each receiver of the external coupling set to the external imminent set and adding the content of the *inbox* to the *receiver*'s *inbox*. The previous steps are performed if the coordinator *inbox* is not empty (an input message was received) and the *receiver*'s next state change is not at t. If it is time for the next state change ($t = next$), the outputs of each imminent model are collected and carried out to any linked coupled model that is then added to the external imminent set.

In all the above cases, the coordinator calls *advance-simulation* for each coordinator in the imminent and external-imminent set and their next state change time is added to the *Future Events List (FEL)*. If the *FEL* is empty, the next state change is set to infinity; otherwise, the next change time is picked from the *FEL*. Finally, all the imminent are retrieved from the *FEL*.

For the Simulator, only the return reply mechanism is required. Since, Coordinators and Simulators do not know their parents, all communications are initiated in a top down fashion, and the replies are collected using the method returned values. If a request expects more than one message as a reply, all

```
// Coordinator variables
Time next// time of next event
Time last// time of last event
Bag⟨Message⟩ inbox // inbox for incomming messages
Set⟨Simulator⟩ imminents // set of imminent simulators (or coordinators)
Coupled-model m // model being coordinated
Function collect-outputs(Time t) → Bag⟨Message⟩
    if t ≠ next then
        return ∅
    else
        Bag⟨Message⟩ outputs := ∅
        foreach Coordinator c in imminents do
            if c.m in m.EOC then
                outputs := outputs ∪ c.collectOutputs(t)
            end
        end
        return outputs
    end
End
Function advance-simulation(Time t)
    assert t in [last, next]
    last := t
    Set⟨Simulator⟩ external-imminents := ∅
    foreach receiver in m.EIC do
        if inbox ≠ ∅ ∧ receiver.next ≠ t then
            add receiver to external-imminents
        end
        add inbox contents to receiver.inbox
    end
    if next = t then
        foreach coupling ic in IC of all imminents do
            temp := collect-outputs(ic.first)
            if not empty temp ∧ ic.second.next ≠ t then
                add coupling.second to external-imminents
            end
            add temp to ic.second.inbox
        end
    end
    foreach Coordinator c in ⋃(imminents,external-imminents) do
        c.advance-simulation(t)
        if c.next ≠ ∞ then
            FEL.insert(c.next, c)
        end
    end
    if FEL is empty then
        next := ∞
    else
        next := FEL.top.time
    end
    imminents = all Coordinators scheduled in FEL for next
    remove imminents from FEL
End
```

Algorithm 7.1: Sequential PDEVS Coordinator

the required messages are wrapped in a tuple that is returned as the response. The function names are the same as in the Coordinator: *advance-simulation* and *collect-outputs.* The algorithms for the Simulator are shown in Algorithm 7.2.

```
// Simulator variables
Time next// time of next event
Time last// time of last event
Bag⟨Message⟩ inbox // inbox for incomming messages
Atomic-model a // model being simulated
a::State s// current state
Function collect-outputs(Time t) → Bag⟨Message⟩
    if t ≠ next then
        return ∅
    else
        return a.λ(state)
    end
End
Function advance-simulation(Time t)
    assert t in [last, next]
    if inbox = ∅ ∧ t = next then
        state := a.δ_int(state)
        next := last + a.ta(state)
    end
    if inbox = ∅ then
        if t = next then
            state := a.δ_conf(inbox, t − last, state)
        else
            state := a.δ_ext(inbox, t − last, state)
        end
        next := last + a.ta(state)
    end
    last := t
End
```

Algorithm 7.2: Sequential PDEVS Simulator

The *collect-outputs* method verifies the parameter time *t*. If this time is different from the next scheduled event, an empty bag is returned; otherwise, the output generated by the model is returned. For *advance-simulation*, we verify if the time *t* is legitimate by making sure it is within the last change and next expected change. If *advance-simulation* was called with a valid time *t*, the *inbox* content is checked. If the *inbox* is empty and it is time for processing the next event, the internal transition function is executed and the *next* variable is set to the result of adding *last* to *ta*. When *inbox* is not empty, meaning an input has been received; we execute the external, or confluence, transition function. Which transition function to run depends of the timing of the input being simultaneous to an scheduled internal event or not.

Similar sequential algorithms had being developed to simulate Classical-DEVS models. In the case of Classical-DEVS, the messages need to be passed individually, and a mechanism for applying the SELECT function is defined.

7.2 Architecture

The architecture of the new simulator consists of modules that mainly correspond to the modeling and simulation execution engines. This modular design has the great advantage of providing simple interfaces to the modeler, who can easily transition from models specification to implementation. The simulation mechanism and the abstract simulator details are hidden, and they do not need to be manipulated by the users, but could be extended by an expert user. The architecture components can be grouped into three categories: *Model* classes, *Execution* classes and *Utility* classes as shown in Figure 7.1

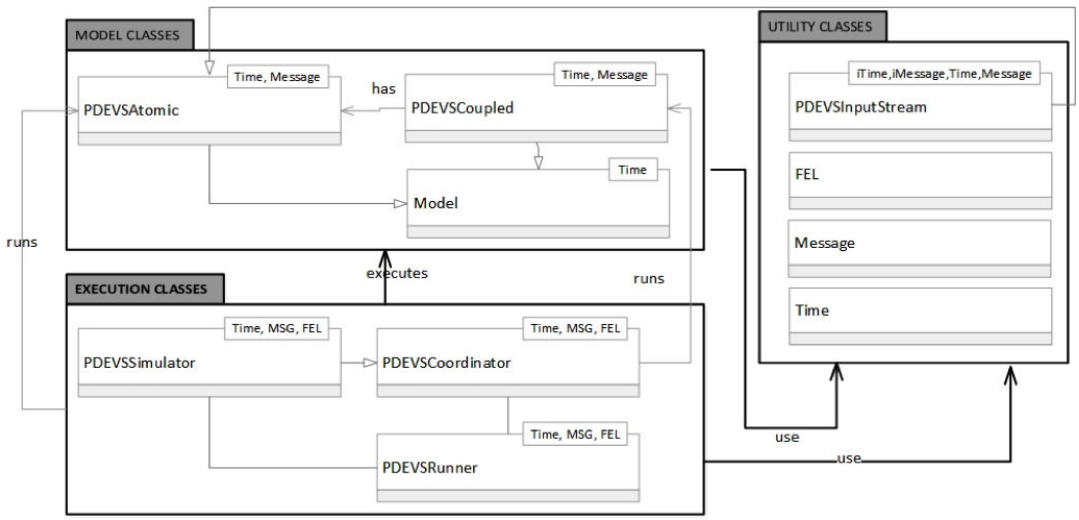

Figure 7.1: Architecture view of the simulator showing the PDEVS simulation components

The *Utility* classes are not explicitly defined in the PDEVS formalism or the abstract simulator, but they have been included as they provide useful functionalities. These classes are used to control the representation used for Messages, Time, and FEL. In addition, utility classes include a model reading streams of events and injecting them into the simulation.

Following, we discuss implementation details about each one of these categories.

7.2.1 Model classes

For model classes (shown in Figure 7.2), we will first present the PDEVSAtomic class. It is used to define new atomic models. The constructor requires two template parameters: Time and Message. The functions provided by PDEVSAtomic correspond to those described in the formalism: internal, external, confluent, time-advance and output functions. The time-advance, which is commonly included in the internal and external function in various simulation implementations, is here clearly separated and has its own dedicated function.

Coupled models are defined using the PDEVSCoupled class. This class constructor receives pointers of the components to be coupled. Three types of pointers can be passed: pointers to External Input Couplings (EIC) for models that receive inputs from outside of the coupled model, pointers to Internal

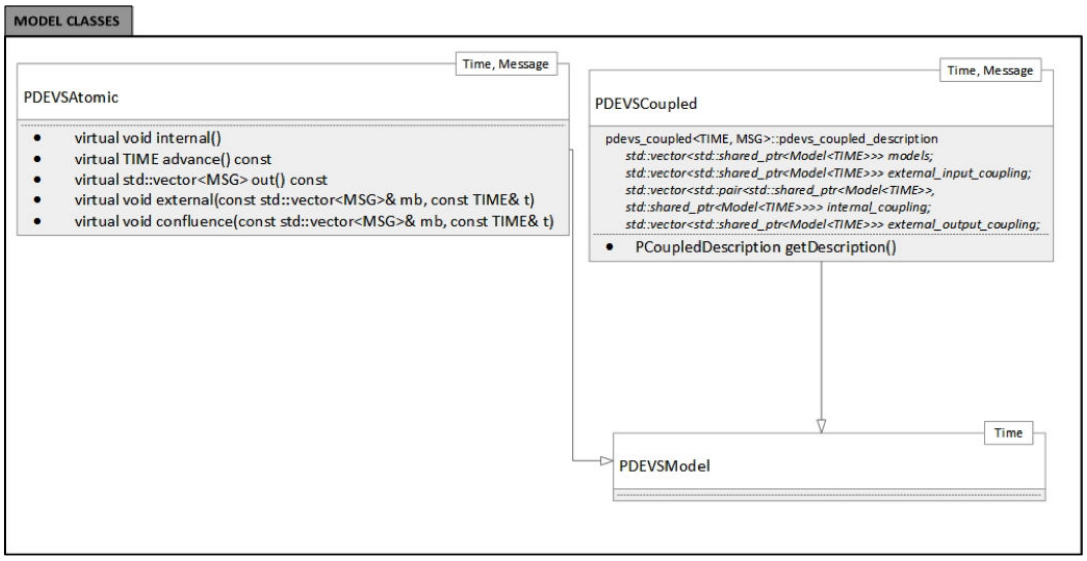

Figure 7.2: Model classes

Couplings (IC) for models that are connected internally; and External Output Couplings (EOC) for models that send outputs outside of the coupled model. PDEVSCoupled provide two implementations of this constructor: one that takes initialization lists and another with vectors. This is particularly useful since certain compilers have not implemented initialization lists.

Both PDEVSAtomic and PDEVSCoupled classes inherit the model class that allows coupled and atomic models to be connected easily through couplings debugged with ease since they share a common model interface.

7.2.2 Execution classes

Execution classes, illustrated in Figure 7.3, implement the abstract simulator algorithms and execute models. The PDEVSCoordinator class, in charge of managing coupled models, requires three template parameters: Time, Message and FEL. These three parameters will be detailed in the utility classes. Constructing coordinator objects is complex, as it requires the coupled model components to be extracted and embedded in the coordinator. For instance, when the coordinator is built, all the children are constructed, and the couplings between components that communicate are saved. The coordination algorithms described previously in Algorithm 7.1 are implemented in this class.

The PDEVSSimulator class implements the simulator's algorithms presented in Algorithm 7.2. Therefore, this class is in charge of calling the state transition functions at the appropriate times, and of returning the outputs of the atomic models to their coordinators. In addition to the described function, an init function setups the model initial time-advance.

Apart from the PDEVSSimulator and PDEVSCoordinator classes, a PDEVSRunner class was implemented. This class advances the simulation for the Root coordinator and defines the end time of the simulation. It also provides mechanisms for customizing output and debugging.

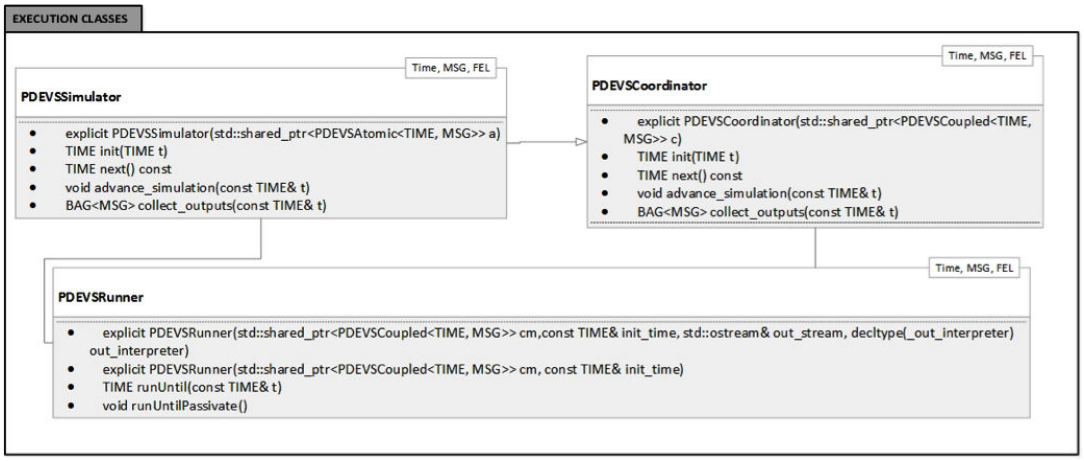

Figure 7.3: Execution classes

7.2.3 Utility classes

The utility classes provide essential data structures to run the simulation properly. The first class in the utility category is called Message. Boost::any is used by default as the message type, and it allows the exchange of any type of messages in our models. This datatype enables us to define type agreements between any pair of connected models without restricting the user to utilize a single datatype for every communication in the simulation.

For the Time, any type having assignation, equality, order, addition and definition of infinite can be used, i.e., double is accepted. As alternative, we also implemented the datatypes proposed in Chapters 4 and 5.

For the FEL, we accept any structure that matches the *std::priority_queue* signature. Hence, the user can define customized schedulers and increase performance if needed. The default provided FEL is the standard priority queue. is also provided as part of the utility classes. Using an effective FEL is essential in order to achieve good performance, as showed in [HU07a]. Other FEL designs considered in [HU07a] should be implemented in the future for evaluation.

In addition to the datatypes provided by the above-described classes, an input stream model is provided. Its role is to allow reading and processing events that originate from an external source. Indeed, in most of the cases, models receive inputs that come from the external environment. A common approach used by many simulators is to add file-reading capabilities to the Root coordinator. However, we believe that having a specialized module that assumes this responsibility is better. Indeed, having the event file processed in the root coordinator requires all inputs to be centralized in the same location and involves additional routing. In contrast, a separate input stream model is more flexible and allows the inputs to be placed close to the model of interest. Therefore, the new simulator provides a standard atomic model capable of processing standard input stream. This model receives two constructor parameters: the input stream to be read and the function to process events in the stream.

The architecture proposed is strongly based on generic and template programming. This approach

allows encapsulating features in a way that replace introduces zero overhead. We implemented a simulator written in C++11 following this architecture. Our intention is submitting it for standardization into the Boost Library. Therefore, the Boost coding standards were used, and only dependencies are on Boost and C++11 standard libraries. Consequently, we were able to compile and test our code on multiple platforms, including various distributions of Linux, DragonFlyBSD, FreeBSD, MS Windows, OSX, and on an ARM platform (using STM32F2 Evaluation Board). Tool chains available in each platform were used, namely clang, gcc, Microsoft and Intel compilers.

To provide multi-formalism, similar execution and model classes are defined for other formalisms. At this point, we have implemented Classical-DEVS and PDEVS in our simulator.

7.3 Performance evaluation

In order to evaluate the performance of the new simulator, we have designed a set of experiments using the DEVStone benchmark previously described in Chapter 2. These experiments run different tests on both High-level of Input coupling (HI) and Low-level of Interconnections (LI) configurations.

The same experiments are also run using the flattened model version of the new simulator and the results are compared to ADEVS for each case. In the following graphs, we will identify the simulators as ADEVS, CDEVS and CDFLAT where CDEVS refers to the new simulator and CDFLAT its flattened version.

In Figures 7.4 and 7.5, we show results of the execution of two models in which events are introduced sequentially, one after the other. The first, shows the results obtained for a LI configuration of width 1000 and height 5 while the second is a HI configuration of width 100 and height 5. ADEVS and CDEVS/CDFLAT performances are close; their difference is never higher than 1%. From this experiment, we can conclude that the new simulator has a similar performance to ADEVS for sequential events.

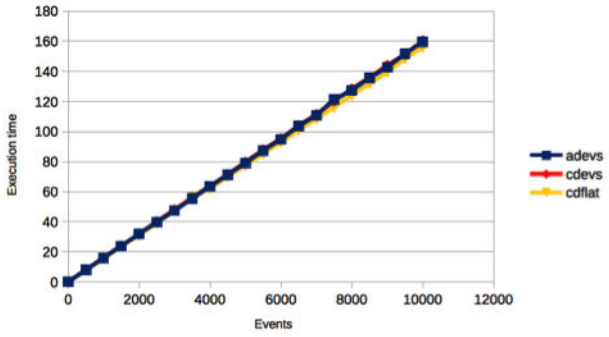

Figure 7.4: Experiment on LI configuration with Width = 1000, Height = 5, and Individual Events

For the next experiment, we use the same models but introduce events simultaneously. This is mainly done in order to verify if our implementation handles simultaneous events as well as ADEVS that uses the closure property.

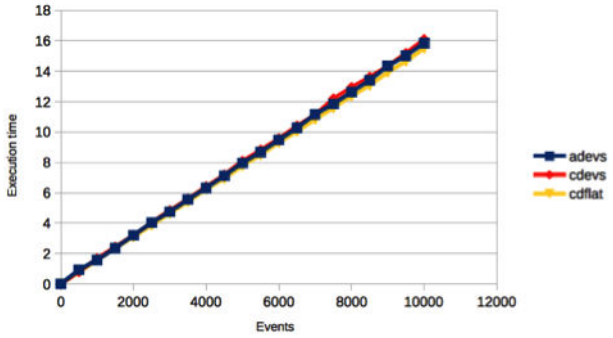

Figure 7.5: Experiment on HI configuration with Width = 100, Height = 5, and Individual Events

In both cases, using LI and HI configurations, the new simulator clearly performs better and handles simultaneous events faster than ADEVS as shown in Figures 7.6 and 7.7. Moreover, the difference increases linearly with the number of injected messages.

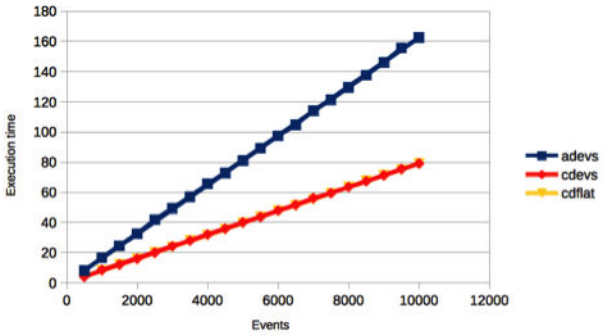

Figure 7.6: Experiment on LI configuration with Width = 1000, Height = 5, and Simultaneous Events

Figures 7.8 and 7.9 shows an experiment using LI configuration with different depths and 5000 input events. The first experiment introduces the events sequentially. The second, introduces them simultaneously.

We observe a similar pattern as in the previous plots. As illustrated, injecting simultaneous events produces a difference linear to the depth of the model. This experiment demonstrates therefore that the difference is exhibited when simultaneous events are involved and is proportional to the depth of the model. This can be explained by the fact that ADEVS has to construct atomic models equivalent to coupled models as per the closure property.

Figure 7.10 are experiments using a LI fixed configuration. Each experiment, we introduced packages of simultaneous events sequentially. For every experiments the total set of events was 5000. The messages were first delivered one at the time, then two at the time, then five at the time, until 500 at the time.

For the first few increments in the packet's size, the execution time was reduced logarithmically. Anyway, for packages of more than twenty events, the benefit does not increase.

From the above experiments, we have observed similar results for HI and LI configurations in the

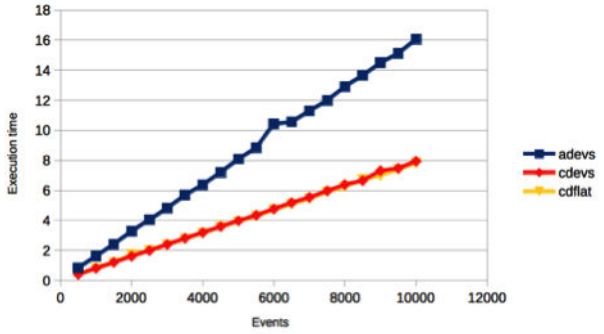

Figure 7.7: Experiment on HI configuration with Width = 100, Height = 5, and Simulataneous Events

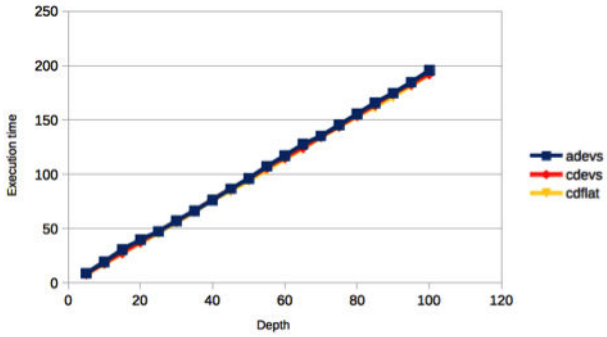

Figure 7.8: Experiment on LI configuration with Width = 100, variable Depth, and Individual Events

presence of sequential events. When simultaneous events are introduced, the new simulator has an obvious advantage that increases when the set of simultaneous events is increased. For LI configurations, the difference also increases proportionally to the width. Hence, the new simulator provides an elegant modular architecture that preserves the coupled/atomic structure and allows having a best-in-class performance.

7.4 Summary

In this chapter, we described a new architecture for implementing PDEVS simulators using a sequential approach. This new architecture allows to easily change the datatypes and algorithms used for working as an experimental workbench.

We presented a performance comparison using the DEVStone benchmark, showing the flexibility added does not have an impact in its usability.

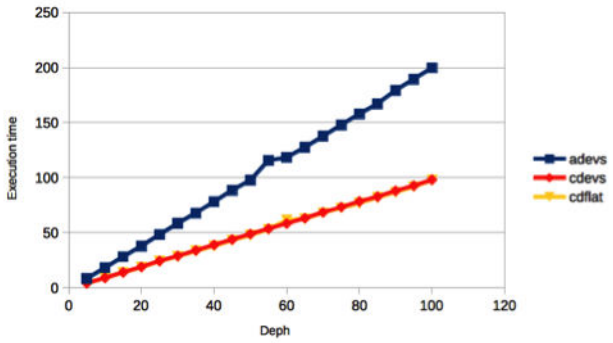

Figure 7.9: Experiment on LI configuration with Width = 100, variable Depth, and Simultaneous Events

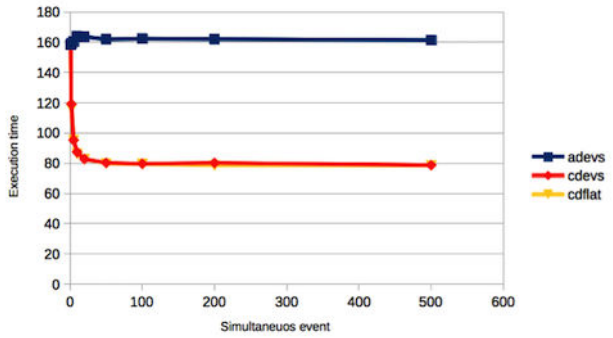

Figure 7.10: Experiment on LI configuration with variable groups of Simultaneous Events

Chapter 8

Conclusions

In this thesis, we reviewed time representation datatypes used for Discrete-Event Simulation (DES) Simulators. We started surveying the implementation of eleven DES Simulators. We found most of them use floating-point (FP) datatypes for their time variables. Yet, a few of them use structures with integers and units.

We described the problems of using datatypes approximating the results of the operations. We classified these problems in three groups: time shifting errors, event reordering errors, and Zeno problems.

We reviewed the limitations found in the literature about the use of FP. We explained how these limitations translate to problems on the chronology of the simulation. We exemplified how these problems can make the simulation produce incorrect results. And we discussed why these problems are impossible to detect in practice.

We reviewed the strengths and limitations of other classic datatypes. And, we checked alternative approaches found in the literature.

We designed a new datatype, Safe-Float (SF), for using as diagnostic tool on old simulations. This datatype wraps FP and reports when the result of an operation is different from the expected one.

We implemented the SF datatype in C++ and experimented with it replacing the time datatype of two Discrete-Event System Specification (DEVS) simulators. We ran every example simulation provided by each simulator distribution. Some of the examples reported errors, showing their results diverge from the theoretically expected ones. This proves the relevance of the research topic.

We proposed then two new datatypes for replacing the use of the old ones and providing correct results. The first, Rational-Scaled floating-point (RSFP), inspired from rational and FP datatypes. The second, Multi-Base floating-point (MBFP) addresses the limitations of FP not being able to represent binary-periodic numbers. Both datatypes represent significantly more values than previous ones. Every operation returning a result produces precise results. If the result of an operation does not fit in the provided space, an error is reported. Similarly to SF, the error prevents the simulation to advance with

incorrect values.

Using the RSFP or MBFP datatypes, we can produce correct results for simulating models defined in any of the surveyed simulators.

In formalisms defining DES models, any real could be used as a time value, including an irrational. Irrationals are commonly used in areas like mechanics, kinetics, and electricity. Using concepts from Computable Calculus, we presented how the previous datatypes can be extended to work with subsets of computable irrational numbers.

We also evaluated the proposed datatypes empirically. For this purpose, we implemented them, ran a set of tests using the DEVStone benchmark, and compared the execution time obtained against the classic datatypes.

We described the problems of using input for simulation with temporal uncertainty quantifications. This input corresponds to data collected using measuring instruments and procedures. The data collected measuring is commonly used in engineering and experimental research. We presented algorithms for simulating models described using the DEVS formalism, and introducing the measurement results as input of the simulation. These algorithms produces output and state trajectories-trees with uncertainty intervals associated to the nodes. These trees summarize all possible trajectories corresponding to every possible occurrence combination of the uncertain input.

For some combinations of models and inputs, the proposed algorithms require infinite branching. To handle this case, we defined a subclass of DEVS model that always use finite branches of simulation for any input received.

We proposed a new sequential architecture for Parallel-DEVS (PDEVS) simulators. This architecture is based in generic meta-programming policies design pattern. We implemented a simulator following this architecture in C++11. The simulator implemented allows us to easily experiment using different datatypes and simulation algorithms. The encapsulation proposed introduces low bias on experiments, and the sequential approach adds in reproducibility. We compared our simulator performance empirically against other simulators using the DEVStone benchmark. The results of the benchmark positioned our simulator at same level as top of the class simulators.

8.1 Future research

We propose continuing our research exploring the following topics related to simulating with uncertainty. First, we want to formally define a new subclass of DEVS called Partially-Forkable Discrete-Event System Specification (PF-DEVS). This new class includes DEVS models in which the external transition results in finite states for every finite interval of elapsed time. This is a weaker condition than the one proposed in Finite-Forkable DEVS (FF-DEVS), and then this is a super class of FF-DEVS. For simulating these models, we plan on defining simulation algorithms that never adjust the t_{last} variable. This approach should produce a superset of the simulation trajectories. The trade-off here is between obtaining the exact set of trajectories, versus simulating more models.

Second, we want to formally define a new subclass of FF-DEVS called Sequence Preserving Discrete-Event System Specification (SQP-DEVS). This class will include the models such that for every finite interval of elapsed time, their external transition results in a unique state. This condition enforces that every trajectory obtained corresponds to a unique sequence of events. We want to study the properties of this new class and compare it against others proposed in the literature.

Third, we want to extend the use of data with uncertainty to the definition of the models. For this purpose, we propose to extend the modeling language, and define new simulation algorithms. This new class called Uncertainty-Aware Discrete-Event System Specification (UA-DEVS) uses real intervals for specifying time variables. These intervals affect the definition of external transition and time-advance functions. Using UA-DEVS we could model uncertain behavior, for example modeling sensors for robots. Other possible usage could be the evaluation of changes in supply chains. For example, we could use UA-DEVS for deciding between two clock manufacturers. In industry, manufacturers guarantee uncertainty limits for their products. A manufacturer could be cheaper, but its clock could have higher uncertainty. A UA-DEVS simulation can help us understand the impact of the change on the clock in the traffic handled by a network router constructed using these clocks, and take a decision on changing or not the provider.

Finally, the algorithms proposed for simulation with uncertainty use branching of the simulation. In some cases, multiple branches will produce similar advances but shifted in the timeline. In DEVS, the transitions are independent from the global time. The transitions process depends on the elapsed time since previous transition. We want to study the use of a cache for optimizing the simulation. The plan is to save in a cache the states obtained from each combination of input passed to the transitions. If the simulation reaches a states cycle or multiple branches follow a similar path the cached results could be used.

Bibliography

[Abe01] Oliver Aberth. *Computable Calculus*. Academic Press, 2001.

[AD94] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[Ale01] Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley, 2001.

[All81] James F Allen. An interval-based representation of temporal knowledge. In *IJCAI*, volume 81, pages 221–226, 1981.

[All83] James F Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[And09] Charles André. Syntax and semantics of the clock constraint specification language (CCSL). *INRIA Research Report 6925*, 2009.

[BIII08] IEC BIPM, ILAC IFCC, IUPAP IUPAC, and OIML ISO. The international vocabulary of metrology—basic and general concepts and associated terms (vim), 3rd edn. jcgm 200: 2012. *JCGM (Joint Committee for Guides in Metrology)*, 2008.

[BIP08] BIPM. Guide to the expression of uncertainty in measurement, (1995), with supplement 1, evaluation of measurement data, jcgm 101: 2008. *Organization for Standardization, Geneva, Switzerland*, 2008.

[BNO03] Roberto Beraldi, Libero Nigro, and Antonino Orlando. Temporal uncertainty time warp: an implementation based on java and actorfoundry. *Simulation*, 79(10):581–597, 2003.

[BV02] Jean-Sébastien Bolduc and Hans Vangheluwe. A modeling and simulation package for classic hierarchical DEVS. *MSDL, School of Computer McGill University, Tech. Rep*, 2002.

[C+08] IEEE Standards Committee et al. 754-2008 ieee standard for floating-point arithmetic. *IEEE Computer Society Std*, 2008, 2008.

[C+11] C++ Standards Committee et al. Iso/iec 14882: 2011, standard for programming language c++. Technical report, Tech. rep., ISO/IEC, 2011. http://www. open-std. org/jtc1/sc22/wg21, 2011.

[CER08a] SEI CERT. *The CERT C secure coding standard*. Pearson Education, 2008.

[CER08b] SEI CERT. *The CERT C++ secure coding standard*. Pearson Education, 2008.

[Cha75] Gregory J Chaitin. A theory of program size formally identical to information theory. *Journal of the ACM (JACM)*, 22(3):329–340, 1975.

[CZ94] Alex Chung Hen Chow and Bernard P Zeigler. Parallel devs: A parallel, hierarchical, modular, modeling formalism. In *Proceedings of the 26th conference on Winter simulation*, pages 716–722. Society for Computer Simulation International, 1994.

[CZK94] AC Chow, Bernard P Zeigler, and Doo Hwan Kim. Abstract simulator for the parallel devs formalism. In *AI, Simulation, and Planning in High Autonomy Systems, 1994. Distributed Interactive Simulation Environments., Proceedings of the Fifth Annual Conference on*, pages 157–163. IEEE, 1994.

[DN66] Ole-Johan Dahl and Kristen Nygaard. Simula: an algol-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[FB02] J-B Filippi and Paul Bisgambiglia. Enabling large scale and high definition simulation of natural systems with vector models and jdevs. In *Simulation Conference, 2002. Proceedings of the Winter*, volume 2, pages 1964–1970. IEEE, 2002.

[FB04] Jean-Baptiste Filippi and Paul Bisgambiglia. Jdevs: an implementation of a devs based formal framework for environmental modelling. *Environmental Modelling & Software*, 19(3):261–274, 2004.

[FBT+14] Romain Franceschini, Paul-Antoine Bisgambiglia, Luc Touraille, Paul Bisgambiglia, David Hill, Rumyana Neykova, and Nicholas Ng. A survey of modelling and simulation software frameworks using discrete event system specification. In *2014 Imperial College Computing Student Workshop*, page 40, 2014.

[Fuj99] Richard M Fujimoto. Exploiting temporal uncertainty in parallel and distributed simulations. In *Proceedings of the thirteenth workshop on Parallel and distributed simulation*, pages 46–53. IEEE Computer Society, 1999.

[GAW08] Marcelo Gutierrez-Alcaraz and Gabriel Wainer. Experiences with the devstone benchmark. In *Proceedings of the 2008 Spring simulation multiconference*, pages 447–455. Society for Computer Simulation International, 2008.

[GCQ08] Valerio Gheri, Giovanni Castellari, and Francesco Quaglia. Controlling bias in optimistic simulations with space uncertain events. In *Distributed Simulation and Real-Time Applications, 2008. DS-RT 2008. 12th IEEE/ACM International Symposium on*, pages 157–164. IEEE, 2008.

[GHJ97] Vineet Gupta, Thomas A Henzinger, and Radha Jagadeesan. Robust timed automata. In *Hybrid and Real-Time Systems*, pages 331–345. Springer, 1997.

[Gly89] Peter W Glynn. A gsmp formalism for discrete event systems. *Proceedings of the IEEE*, 77(1):14–23, 1989.

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1):5–48, 1991.

[GW05] Ezequiel Glinsky and Gabriel Wainer. Devstone: a benchmarking technique for studying performance of devs modeling and simulation environments. In *Distributed Simulation and Real-Time Applications, 2005. DS-RT 2005 Proceedings. Ninth IEEE International Symposium on*, pages 265–272. IEEE, 2005.

[HC04] Moon Ho Hwang and Su Kyoung Cho. Timed behavior analysis of schedule preserved devs. In *Proceedings of 2004 Summer Computer Simulation Conference*, pages 26–29, 2004.

[HF97] Maria Hybinette and Richard Fujimoto. Cloning: a novel method for interactive parallel simulation. In *Proceedings of the 29th conference on Winter simulation*, pages 444–451. IEEE Computer Society, 1997.

[HF01] Maria Hybinette and Richard M Fujimoto. Cloning parallel simulations. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 11(4):378–407, 2001.

[HF02] Maria Hybinette and Richard M Fujimoto. Scalability of parallel simulation cloning. In *Simulation Symposium, 2002. Proceedings. 35th Annual*, pages 275–282. IEEE, 2002.

[Hoa78] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hoo99] William Graham Hoover. *Time reversibility, computer simulation, and chaos*. World Scientific, 1999.

[HSKP97] Joon Sung Hong, Hae-Sang Song, Tag Gon Kim, and Kyu Ho Park. A real-time discrete event system specification formalism for seamless real-time software development. *Discrete Event Dynamic Systems*, 7(4):355–375, 1997.

[HU06] Jan Himmelspach and Adelinde M Uhrmacher. Sequential processing of pdevs models. *Proceedings of the 3rd EMSS*, pages 239–244, 2006.

[HU07a] Jan Himmelspach and Adelinde M Uhrmacher. The event queue problem and pdevs. In *Proceedings of the 2007 spring simulation multiconference-Volume 2*, pages 257–264. Society for Computer Simulation International, 2007.

[HU07b] Jan Himmelspach and Adelinde M Uhrmacher. Plug'n simulate. In *Simulation Symposium, 2007. ANSS'07. 40th Annual*, pages 137–143. IEEE, 2007.

[Hwa07] Moon Ho Hwang. Devs++: C++ open source library of devs formalism, 2007.

[Hwa09] Moon Ho Hwang. *DEVS++: C++ Open Source Library of DEVS Formalism*. On-line resource at http://odevspp.sourceforge.net/ [Last checked: Oct 31st 2013], v.1.4.2 edition, April 2009.

[HZ06] Moon Ho Hwang and Bernard P Zeigler. A modular verification framework based on finite & deterministic devs. *SIMULATION SERIES*, 38(1):57, 2006.

[HZ08] Xiaolin Hu and Bernard P. Zeigler. *The Architecture of GenDevs: Distributed Simulation in DEVSJAVA*. On-line manual at http://acims.asu.edu/wp-content/uploads/2012/02/The-Architecture-of-GenDevs-Distributed-Simulation-in-DEVSJAVA-.pdf [Last checked: Nov 1st 2013], January 2008.

[JK06] Vladimír Janoušek and Elöd Kironskỳ. Exploratory modeling with smalldevs. *Proc. of ESM 2006*, pages 122–126, 2006.

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

[KK98] Yong Jae Kim and Tag Gon Kim. A heterogeneous simulation framework based on the devs bus and the high level architecture. In *Simulation Conference Proceedings, 1998. Winter*, volume 1, pages 421–428. IEEE, 1998.

[KKSS00] Kihyung Kim, Wonseok Kang, Bong Sagong, and Hyungon Seo. Efficient distributed simulation of hierarchical devs models: transforming model structure into a non-hierarchical one. In *Simulation Symposium, 2000.(SS 2000) Proceedings. 33rd Annual*, pages 227–233. IEEE, 2000.

[Lac10] Mathieu Lacage. *Outils d'Expérimentation pour la Recherche en Réseaux*. PhD thesis, Nice, 2010.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[Lee14] Edward A Lee. Constructive models of discrete and continuous physical phenomena. Technical report, Technical Report UCB/EECS-2014-15, EECS Department, University of California, Berkeley, 2014.

[LF00] Margaret L Loper and Richard M Fujimoto. Pre-sampling as an approach for exploiting temporal uncertainty. In *Proceedings of the fourteenth workshop on Parallel and distributed simulation*, pages 157–164. IEEE Computer Society, 2000.

[LF04a] Margaret L Loper and Richard M Fujimoto. A case study in exploiting temporal uncertainty in parallel simulations. In *Parallel Processing, 2004. ICPP 2004. International Conference on*, pages 161–168. IEEE, 2004.

[LF04b] Margaret L Loper and Richard M Fujimoto. Exploiting temporal uncertainty in process-oriented distributed simulations. In *Proceedings of the 36th conference on Winter simulation*, pages 395–401. Winter Simulation Conference, 2004.

[LW04] Alejandro López and Gabriel A. Wainer. Improved Cell-DEVS model definition in CD++. In *Proceedings of ACRI. Lecture Notes in Computer Science*, volume 3305, Amsterdam, Netherlands, 2004. Sloot, P.; Chopard, B.; Hoekstra, A. Eds.

[Mil80] Robin Milner. A calculus of communicating systems. *Lecture Notes in Computer Science*, 92, 1980.

[MMP92] Oded Maler, Zohar Manna, and Amir Pnueli. From timed to hybrid systems. In *Real-time: theory in practice*, pages 447–484. Springer, 1992.

[MN05] Alexander Muzy and James J Nutaro. Algorithms for efficient implementations of the devs & dsdevs abstract simulators. In *1st Open International Conference on Modeling & Simulation (OICMS)*, pages 273–279, 2005.

[Moo66] Ramon E Moore. *Interval analysis*, volume 4. Prentice-Hall Englewood Cliffs, 1966.

[MP92] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems: specifications*, volume 1. Springer Science & Business Media, 1992.

[MZRM09] Saurabh Mittal, Bernard P Zeigler, and Jose L Risco-Martin. Implementation of formal standard for interoperability in m&s/systems of systems integration with devs/soa. *International Command and Control C2 Journal, Special Issue: Modeling and Simulation in Support of Network-Centric Approaches and Capabilities*, 3(1), 2009.

[Nan81] Richard E Nance. The time and state relationships in simulation modeling. *Communications of the ACM*, 24(4):173–179, 1981.

[NP01] Yuri Valentinovich Nesterenko and Patrice Philippon. *Introduction to algebraic independence theory*. Number 1752 in Lecture Notes in Mathematics. Springer Science & Business Media, 2001.

[Nut03] J Nutaro. ADEVS: User manual and API documentation. *On-line document, http://www. ece. arizona. edu/˜nutaro/adevs-docs/index. html*, 2003.

[Pet81] James L Peterson. *Petri net theory and the modeling of systems*. Prentice Hall PTR, 1981.

[PM07] Patrick Peschlow and Peter Martini. A discrete-event simulation tool for the analysis of simultaneous events. In *Proceedings of the 2nd international conference on Performance evaluation methodologies and tools*, page 14. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007.

[QB04] Francesco Quaglia and Roberto Beraldi. Space uncertain simulation events: some concepts and an application to optimistic synchronization. In *Parallel and Distributed Simulation, 2004. PADS 2004. 18th Workshop on*, pages 181–188. IEEE, 2004.

[Ram12] Robert Ramey. Safe numerics library. `https://github.com/robertramey/safe_numerics`, 2012. Accessed: 2015-09-20.

[Saa12] Hesham Saadawi. *Verification Methodology for DEVS Models*. PhD thesis, Carleton University Ottawa, 2012.

[SQK11] Mayerlin Yuleydy Uzcategui Sulbarán, Jacinto Alfonso Dávila Quintero, and Kay Andrés Tucci Kellerer. Galatea: una historia de modelado y simulación. *Ciencia e Ingeniería*, pages 85–94, 2011.

[SW10] Hesham Saadawi and Gabriel Wainer. Rational time-advance devs (rta-devs). In *Proceedings of the 2010 Spring Simulation Multiconference*, page 143. Society for Computer Simulation International, 2010.

[Tur36] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[VDW14] Damián Vicino, Olivier Dalle, and Gabriel Wainer. A data type for discretized time representation in devs. In *Proceedings of the 7th International ICST Conference on Simulation Tools and Techniques*, pages 11–20. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2014.

[VDW15] Damián Vicino, Olivier Dalle, and Gabriel Wainer. Using finite forkable devs for decision-making based on time measured with uncertainty. In *Proceedings of the 8th International Conference on Simulation Tools and Techniques*, pages 89–98. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2015.

[VH08] András Varga and Rudolf Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, page 60. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.

[Vic15] Damián Vicino. Safe-float library. `https://github.com/sdavtaker/safefloat`, 2015. Accessed: 2015-09-20.

[VNWD15] Damián Vicino, Daniella Nayunkuru, Gabriel Wainer, and Olivier Dalle. Sequential pdevs architecture. In *Proceedings of the TMS/DEVS 2015*. SCS (The society for modeling and simulation international), 2015.

[VTV14] Yentl Van Tendeloo and Hans Vangheluwe. The modular architecture of the python (p) devs simulation kernel. In *Proceedings of the 2014 Symposium on Theory of Modeling and Simulation-DEVS*, pages 387–392, 2014.

[Wai09] Gabriel A Wainer. *Discrete-event modeling and simulation: a practitioner's approach*. CRC Press, 2009.

[Wei84] Reinhold P Weicker. Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, 1984.

[WGGA11] Gabriel Wainer, Ezequiel Glinsky, and Marcelo Gutierrez-Alcaraz. Studying performance of devs modeling and simulation environments using the devstone benchmark. *Simulation*, page 0037549710395649, 2011.

[ZC92] Bernard P Zeigler and Sungdo Chi. Symbolic discrete event system specification. *Systems, Man and Cybernetics, IEEE Transactions on*, 22(6):1428–1443, 1992.

[ZPK+76] Bernard P Zeigler, Herbert Praehofer, Tag Gon Kim, et al. *Theory of modeling and simulation*, volume 19. John Wiley New York, 1976.

[ZPK00] Bernard P Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems*. Academic press, 2000.

[ZS03] Bernard P Zeigler and H Sarjoughian. Introduction to DEVS modeling & simulation with JAVATM: Developing component-based simulation models. *Arizona State University*, 2003.