Exploration of Direct Synchronization Approaches for a High-Level and Unified Simulation of Discrete-Event/Continuous-Time Systems

Breytner Joseph Fernandez Mesa

Université Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présenté par

Breytner Joseph FERNANDEZ MESA

Thèse dirigée par **Frédéric PETROT (MSTII)**, Professeur Grenoble INP/Ensimag, Université Grenoble Alpes
et codirigée par **Liliana Lilibeth ANDRADE PORRAS**, Maître de conférence UGA / Polytech, Université Grenoble Alpes

préparée au sein du **Laboratoire Techniques de l'Informatique et de la Microélectronique pour l'Architecture des systèmes intégrés** dans **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Exploration des approches de synchronisation directes pour la simulation unifiée et de haut niveau des systèmes continus/discrets

Exploration of Direct Synchronization Approaches for a High-Level and Unified Simulation of Discrete-Event/ Continuous-Time Systems

Thèse soutenue publiquement le **14 octobre 2021**,
devant le jury composé de :

Florence Maraninchi
Professeur des universités, Université Grenoble Alpes, France, Présidente
Christoph Grimm
Professeur, Technische Universität Kaiserslautern, Allemagne, Rapporteur
Robert de Simone
Directeur de recherche, INRIA Sophia Antipolis, France, Rapporteur
François Pêcheux
Professeur des universités, Sorbonne Université, LIP6, France, Examinateur
Nicolas Ventroux
Ingénieur HDR, Thales Research & Technology, France, Examinateur
Rainer Dömer
Professeur, University of California, Irvine, États-Unis, Examinateur
Frederic Pétrot
Professeur des universités, Université Grenoble Alpes, Directeur de thèse

Abstract

Current integrated cyber-physical systems combine digital and analog components that interact and create complex system behavior. The digital components can be hardware and software; the analog components can be electrical, mechanical, etc. Their integration has safety-critical applications in aerospace, automotive, defense, and other industries. Designers must assure safe and predictable functioning within constrained design budgets and short time-to-market intervals. To this end, they rely on modeling and simulation.

However, cyber-physical system simulation is prone to accuracy and speed problems. The simulation of digital components is based on discrete events (DE) and it progresses by discrete timesteps. The simulation of analog components is based on continuous-time (CT) differential equations and it progresses in a time continuum. Despite different time representations, data must be transmitted between both domains at precise instants (synchronization). One solution is fixed-step synchronization, which occurs at regular and statically user-defined timesteps. But it raises a dilemma: either the timestep is small, the simulation accurate but slow, or the timestep is large, the simulation possibly inaccurate, but fast. CT/DE synchronization has a vast impact on simulation accuracy and speed.

The purpose of this thesis is to accelerate the simulation of combined CT/DE models. To this aim, we propose a sequential direct CT/DE synchronization algorithm that makes the CT and DE domains interact by events. Synchronization takes place only at the event notification times, preserving accuracy and improving speed when compared to fixed-step approaches.

However, direct synchronization increases the modeling costs: it requires the CT components to communicate only by events. To address this issue, we also propose a modeling interface that can be automatically defined by a CT model of computation. This enables designers to specify models only by interconnecting primitives. Some primitives can consume and generate events. Such models take a graphical representation form, e.g., electrical circuits, and simulate on top of our algorithm with high accuracy and speed.

Last, based on our sequential approach, we propose a parallel direct CT/DE synchronization algorithm for the simulation of models containing multiple CT components. Our solution groups these components, synchronizes their execution, and simulates them in parallel. It improves simulation speed when compared to our sequential algorithm, which already shows an important speed-up when compared to fixed-step approaches.

Combined CT/DE modeling and simulation with high accuracy and speed is an aid to the design of high confidence and performant complex cyber-physical systems while maintaining short design cycles and reduced costs.

Résumé

Les systèmes cyber-physiques intégrés actuels combinent des composants numériques et analogiques qui interagissent et permettent de créer des comportements de systèmes complexes. Les composants numériques peuvent être matériels et logiciels tandis que les composants analogiques peuvent être électriques, mécaniques, etc. Leur intégration a des applications critiques pour la sécurité dans l'aérospatiale, l'automobile, la défense et d'autres industries. Les concepteurs doivent assurer un fonctionnement prévisible dans le cadre de budgets de conception limités et de courts intervalles de mise sur le marché. À cette fin, ils s'appuient sur la modélisation et la simulation.

Cependant, la simulation de systèmes cyber-physiques est sujette à des problèmes de précision et de vitesse. La simulation des composants numériques est basée sur des événements discrets (ED) et elle progresse par pas de temps discrets. La simulation de composants analogiques est basée sur des équations différentielles en temps continu (CT) et progresse dans un continuum temporel. Malgré des représentations temporelles différentes, les données doivent être transmises entre les deux domaines à des instants précis (synchronisation). Une solution est la synchronisation à pas fixe, qui se produit à des pas de temps réguliers et statiquement définis par l'utilisateur. Mais cela pose un dilemme : soit le pas de temps est petit, la simulation précise mais lente, soit le pas de temps est grand, la simulation peut-être imprécise, mais rapide. La synchronisation TC et ED a un impact considérable sur la précision et la vitesse de la simulation.

L'objectif de cette thèse est d'accélérer la simulation de modèles combinés en TC et à ED. Dans ce but, nous proposons un algorithme de synchronisation séquentielle directe CT/DE qui fait interagir les domaines TC et ED par événements. La synchronisation n'a lieu qu'aux moments de notification des événements, préservant la précision et améliorant la vitesse par rapport aux approches à pas fixes.

Cependant, la synchronisation directe augmente les coûts de modélisation : elle oblige les composants en TC à communiquer uniquement par événements. Pour répondre à ce problème, nous proposons également une interface de modélisation qui peut être définie automatiquement par un modèle de calcul en TC. Cela permet aux concepteurs de spécifier des modèles uniquement en interconnectant des primitives dont certaines gèrent la communication par événements. De tels modèles prennent une forme de représentation graphique, par exemple, sous forme de circuits électriques, et sont simulés sur notre algorithme avec une précision et une vitesse élevées.

Enfin, sur la base de notre algorithme séquentiel, nous proposons un algorithme parallèle pour la simulation de modèles combinés contenant plusieurs composants en TC. Notre solution regroupe ces composants, synchronise leur exécution et les simule en parallèle. La vitesse de simulation est améliorée par rapport à notre algorithme séquentiel, qui fournit déjà une accélération importante par rapport aux approches à pas fixes.

La simulation en TC et à ED avec une précision et une vitesse élevées est une aide à la conception de systèmes cyber-physiques complexes hautement fiables et performants.

Acknowledgements

I would like to express my profound appreciation to all those who have helped me to attain this goal. In particular, I would like to thank:

Prof. Christoph Grimm and Prof. Robert de Simone, for having accepted to report on this thesis and for their constructive comments during their revision.

Prof. Florence Maraninchi, Prof. François Pêcheux, Dr. Nicolas Ventroux, and Prof. Rainer Dömer, for having kindly taken part as members of the jury.

Frédéric Pétrot, for his trust, pertinence, and acuteness in the direction of this thesis.

Liliana Andrade, for the diligence and meticulousness of her supervision from the day we prepared my candidature to the present.

Olivier Müller, Laurence Pierre, Frédéric Rousseau, Arthur Perais, Marie Badaroux, Bruno Ferres, Arthur Vianès, and all other members of the System Level Synthesis team of TIMA Laboratory, for the warm working environment. Luc Michel, Clément Deschamps, and Damien Hedde, members and former members of GreenSoCs.

Gerard Paez, for his continuous advice since my early engineering school days and for his recommendation to Liliana and Fréderic to accept me as a Ph.D. student.

Claudia Gómez, Anna Patete, Mario Spinetti, Iñaki Aguirre, and Richard Márquez, members and former members of the Control Engineering department of the School of Systems Engineering of Universidad de Los Andes—Venezuela, for their rigorous instruction and for feeding my interest in teaching and research since the first semesters of my engineering degree.

My mother, Bertha Fernández, for her tenderness and care: you have taught me to give my best since I was little, this thesis is the fruit of your fervor in my education. I also praise my sisters and family, for all their love and support.

Thibaud Mautuit, for your noble temper, pristine spirit, and exquisite wit. Such rare integrity is a strength I cannot but cherish and wish to share.

To all those who have cared for me and whose name I leave unmentioned.

Contents

List of Figures

List of Tables

Chapter 1: Introduction

Contents

Perhaps that is what the role of an artist relies on—giving a foretaste of something that could exist, and thus causing it to become imaginable. And being imagined is the first state of existence

— Olga Tokarczuk, Nobel Lecture: The Tender Narrator

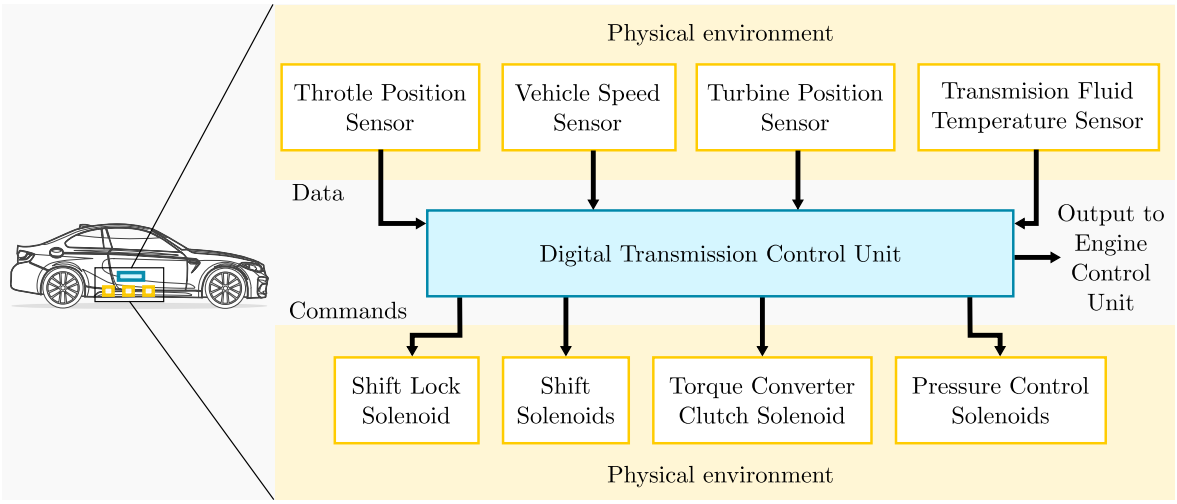


Figure 1.1: Example of an automotive system integrating digital and analog components, adapted from [1, 2]

1.1 Context

Current integrated cyber-physical systems spread rapidly and promise to support major upcoming technological advances, e.g., the Internet of Things, the Smart Grid, Industry 4.0, etc. [11]. They combine digital and analog components to provide computation and networking in their physical environment, in which they sense data and command actuators [12]. Figure 1.1 shows the example of the transmission control unit of a vehicle: it senses positions, speeds, and temperatures; computes control actions digitally; and carries out these actions in their environment via actuators, such as shift, torque, and pressure control solenoids. It also communicates with other digital components, for instance, the engine control unit. In general, the digital components can be hardware and software, such as processors, firmware, and control algorithms. The analog components can be electrical (e.g., a power converter circuit), mechanical (e.g., a car suspension mechanism), chemical (e.g., a carbon monoxide detector), or from any other physical discipline. They interact and create complex system behavior.

Assuring safe component interaction is an unavoidable challenge for designers. These systems have safety-critical applications in the aerospace, automotive, defense, and other industries where small failures could cause great losses—human, economic, etc. In addition, their design, partitioning, manufacture, integration, and validation have to be achieved within constrained budgets and short time-to-market intervals. To meet these goals, *system designers* rely on modeling and simulation.

However, cyber-physical system simulation is prone to accuracy and speed problems. Digital components are typically modeled by processes that produce and consume *discrete events* (DE) in time order at variable timesteps; some DE languages and tools are VHDL [13], Verilog [14], and SystemC [15]. Conversely, analog components are typically modeled by *continuous-time* (CT) differential equations that are solved by symbolic or numerical inte-

gration algorithms over a continuous time interval; some CT languages and tools are MATLAB/Simulink [16] and Mathematica [17]. While CT simulation advances in a time continuum, DE simulation does so by discrete steps; the main difficulty of combined CT/DE simulation is to find suitable instants at which to transmit data between both *time domains* in an accurate and performant manner (*synchronization*).

There are two major views with respect to the integration of CT and DE simulation languages and tools: *co-simulation* and *unified simulation*. CT/DE co-simulation combines community-received tools following standards such as Functional Mockup Interface (FMI) [18] and High Level Architecture (HLA) [19]. But these tools were never conceived to interoperate and their integration requires intermediary mechanisms that add computation and communication overhead. Unified CT/DE simulation eliminates this overhead. It also enables designers to share a single language and modeling approach for all components. Although it might require converting models to a new language, unified simulation significantly reduces the simulation costs and proves useful in the long run.

Some unified CT/DE modeling and simulation languages and tools integrate both time domains, but their models are overly detailed for cyber-physical system design. For example, VHDL-AMS [20] and Verilog-AMS [21] work as extensions to the widely used hardware design languages VHDL [13] and Verilog [14], but their digital component models are at the *Register-Transfer Level* (RTL) of abstraction and take the form of registers, logical operations, and data transactions. This level of detail is likely too slow for system simulations containing many components that interact in intricate manners [22]. It is then necessary to support the simulation of models at a *high level of abstraction*, i.e., models that are just detailed enough to make complex system design a manageable task.

As an attempt to solve this problem, the Electronic System Level (ESL) design community has adopted the SystemC standard [15]. It supports high-level modeling and simulation of digital hardware and embedded software systems via its *Transaction Level Modeling* (TLM) extensions. It also supports high-level modeling and simulation of physical systems via its *Analog and Mixed-Signal* (AMS) extensions. However, SystemC AMS defines a CT/DE synchronization strategy based on statically user-defined simulation timesteps [23]. This strategy faces the designer with a dilemma: either the timestep is small, the simulation accurate but slow, or the timestep is large, the simulation possibly inaccurate but fast. As an alternative, *direct CT/DE synchronization* is based on events that are generated when the component variables meet certain conditions. Since the CT/DE synchronization strategy has a vast impact on the simulation accuracy and speed, we consider that proposing a direct strategy could help to circumvent the trade-off introduced by fixed-step approaches.

The implementation of a new direct CT/DE synchronization strategy in SystemC would extend this language to the modeling and simulation of combined CT/DE systems with high accuracy and speed to meet current cyber-physical integrated system design needs.

1.2 Thesis Objective and Outline

The purpose of this thesis is to accelerate the unified simulation of CT/DE models to support system design use cases that demand high simulation accuracy and speed. To this aim, we explore sequential and parallel direct synchronization algorithms.

After having defined in this chapter the context and purpose of our work, we organize this thesis as follows:

In Chapter 2, we review CT and DE modeling and simulation. We focus our attention on the problem of CT/DE synchronization and the accuracy and speed trade-off of fixed-step approaches. We also discuss and motivate parallel simulation. At the end of the chapter, we present our research questions on direct synchronization algorithms, their support for the simulation of models from different physical disciplines, and their parallel execution as the main axes guiding our work.

In Chapter 3, we review the state-of-the-art literature on the domain. We study direct CT/DE interactions and their modeling requirements, we describe some languages and tools for unified modeling and simulation, and we give an overview of parallel simulation. In this manner, we identify the available paths to answer our research questions.

In Chapter 4, we propose a sequential direct CT/DE synchronization algorithm. Synchronization takes place at the interaction event notification times. We demonstrate the causality, completeness, and liveness properties of our algorithm, important to ensure simulation correctness. We experimentally confirm that our solution speeds up simulation when compared to fixed-step approaches.

However, our algorithm requires the CT models to support direct interactions by events. We gather these requirements in a set of CT interface methods. In Chapter 5, we show that this interface can be automatically defined to facilitate the modeling and simulation of systems from different physical disciplines. We implement abstract-mathematical and electrical models and demonstrate their simulation on top of our algorithm with high accuracy and speed.

Then, based on our sequential solution in Chapter 4, we propose a parallel direct CT/DE synchronization algorithm for the simulation of models containing multiple CT components. It overcomes typical simulation challenges such as the preservation of causality, the prevention of race conditions, and the efficient use of parallel resources. It also preserves the causality, completeness, and liveness properties of the sequential algorithm. Our parallel solution further accelerates simulation.

Finally, in Chapter 7, we present our conclusions on direct CT/DE synchronization and our perspectives for future research.

Chapter 2: Motivation and Definition of the Problem

Contents

There is always a curious tie at some point between the fall and the creation. Taking this ghastly risk is the condition of there being life. You see, for all life is an act of faith and an act of gamble

— Alan Watts, Falling in Love

2.1 Introduction

Current integrated systems mix up components from the discrete-event and continuous-time domains. Their industrial applications require high-confidence and performant systems with short design cycles and reduced costs. System modeling and simulation help to meet these goals. However, the combination of domains and tight coupling of components create complex behaviors that can compromise simulation accuracy and speed.

To identify the constraints and open possibilities to solve this issue, we begin in Section 2.2 by describing modeling and simulation and its use cases in system design. Then, we review modeling and simulation for discrete-event systems in Section 2.3 and continuous-time systems in Section 2.4; we emphasize the levels of abstraction in behavior and structure and the simulation algorithms, factors that affect simulation accuracy and speed. In Section 2.5, we expose fixed-step synchronization and we show how it introduces an accuracy and speed trade-off. In addition, in Section 2.6 we present parallel execution for simulation acceleration. Finally, we conclude our chapter in Section 2.7.

2.2 Modeling and Simulation in System Design

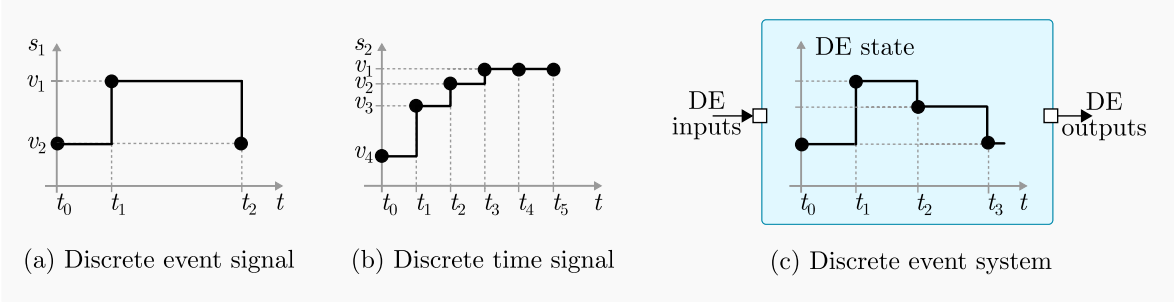
A major task in integrated system design is to maintain high confidence in these systems while working near to their performance limits [24]. Take, for example, safety-critical applications like automated avionics, robotic surgery, braking devices, etc; any failure can cause lives to be lost. The CT and DE domains have well-developed design methods to fulfill this task. However, the combination of domains and tight coupling of components make these isolated methods insufficient. Designers seek to study the system rather than the individual parts [25–27]. Unified modeling and simulation are a means to study system behavior.

A *model* is an abstraction of the system behavior. Simulation uncovers this behavior. System modeling and simulation should support different design use cases [28]:

- **The creation of *executable specifications***, which are functional or algorithmic models of behavior. Designers specify the system requirements in executable specifications that are more rigorous, congruent, and unambiguous than what a paper description could be. Then, during development, designers refine the specifications to more detailed models or convert them into integration tests [28, 29].

- **The development of *virtual prototypes***, which are high-level hardware models that enable the development and test of embedded software before the actual hardware is available. Designers need high simulation speed to handle complex prototypes.

- **The exploration of architectures**, which is the refinement of behavior to a set of architectural elements, communication interfaces, and channels. The detailed models include non-ideal properties and implementation constraints. To this aim, designer search for a satisfactory solution regarding the required behavior and constraints. This process is iterative and simulation must be fast [30].



(a) Discrete event signal (b) Discrete time signal (c) Discrete event system

Figure 2.1: Discrete-event signals and systems

- **Verification and validation**, which refer to testing designs before and after tape-out, respectively [31]. Designers *verify* models during development to confirm functional correctness and other non-functional properties such as power consumption, security, safety, etc. Some verification methods include formal methods, FPGA prototyping, and simulation-based coverage-driven verification, which is currently the most commonly used [32]. After tape-out, designers *validate* component integration to guarantee that behavior meets expectations. Accurate interface modeling allows actual components to be coupled with simulated models to constitute real-time hardware-in-the-loop validation frameworks. Accuracy and speed are key.

Beyond the technical need for modeling and simulation, the business angle calls for it to shorten design cycles, risks, and costs.

DE and CT modeling and simulation have evolved independently and have given rise to unrelated languages and tools. Supporting both time domains under a unified, accurate, and fast simulation environment proves challenging.

2.3 Discrete-Event Modeling and Simulation

We deal with *dynamic systems* whose behavior evolves in time. They take a set of inputs, process them, update their state, and produce a set of outputs. Inputs, outputs, and states are valued functions in time, also known as *signals*. Time allows ordering behavior to distinguish between past, present, and future. Signals can be classified as discrete-event, discrete-time, and continuous-time depending on how time is modeled (*time base*).

In this section, we are interested in *discrete-event signals*, whose time base is the set of integer numbers—or any isomorphic set to the integers [33]. Figure 2.1 (a) is an example over the interval that starts at t_0 and ends at t_2: the signal changes its value only in the discrete set of time points $\{t_0, t_1, t_2\}$ (*timestamps*) that are irregularly spaced. We call each timestamp and value pair an *event*. These signals are typical of digital hardware and software, e.g., the value of a variable in a program and the content of a flip-flop in a digital circuit.

A related type are *discrete-time (DT) signals*, which are discrete-event signals in which the distance between time points is fixed [33]. Figure 2.1 (b) is an example. It changes its value only in the discrete set of time points $\{t_0, t_1, ..., t_5\}$ that are regularly spaced. These

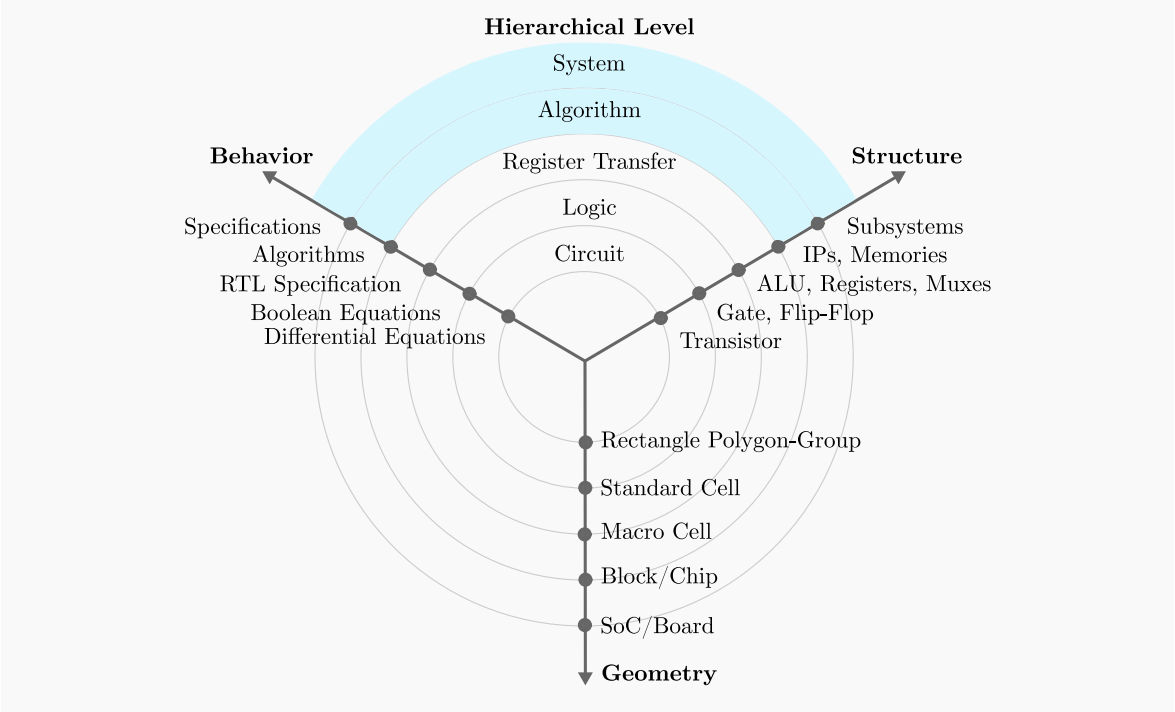


Figure 2.2: Gajski Kuhn Y-chart: levels of abstraction in digital systems adapted from [3], with the levels at which we place our contributions in blue

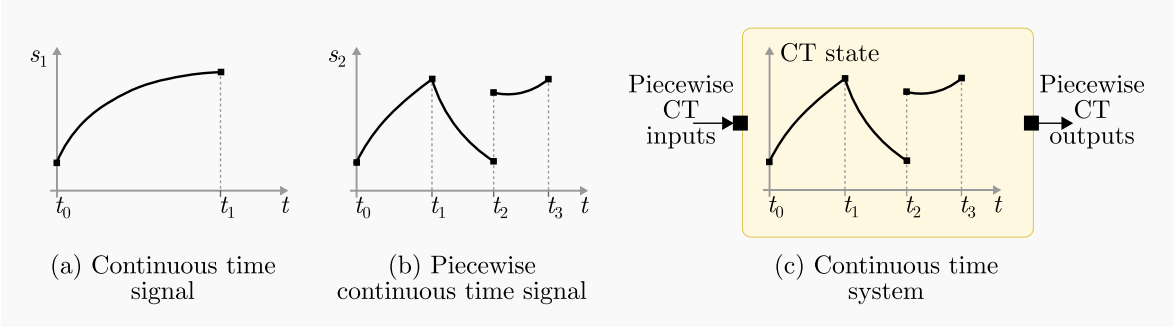
signals are typical of digital signal processing applications, where audio, image, and other signals are sampled at regular intervals.

Discrete-event systems are systems that take in DE inputs, process them, update their DE state, and produce DE outputs. Figure 2.1 (c) is an example. Discrete-event systems can process discrete-time signals because they are a special type of discrete-event signal. In cyber-physical systems, the digital components are discrete-event. These systems are *causal*, meaning that outputs depend only on past and present inputs, and not on future ones.

The behavior and structure of DE digital hardware and software models can be specified with more or less detail depending on the designer's goal.

2.3.1 Levels of Behavioral and Structural Abstraction in Discrete-Event Models

The Gajski Kuhn Y-chart in Figure 2.2 shows the levels of DE behavioral and structural abstraction [3]. Each level is defined in three axes: *behavior*—functioning in time—, *structure*—constitutive elements and interconnections—, and *geometry*—size and shape properties relevant for physical placement. High-level modeling and simulation covers the system and algorithmic levels on the behavior and structure axes. In this thesis we deal with high-level DE models and their simulation.



(a) Continuous time signal

(b) Piecewise continuous time signal

(c) Continuous time system

Figure 2.3: Continuous-time signals and systems

2.3.2 Simulation of Discrete-Event Models

DE simulators monitor and maintain several variables: the DE simulation time (t_{DE}), the DE state, a list of pending events, and a map from the events to a set of *sensitive processes*. They advance time from event to event, processing first those with the earliest timestamp. An event triggers the execution of its sensitive processes. A process can update the system state and generate other events. The simulation ends when the list of events is empty. Execution in time order is critical to prevent *causality errors* where output events depend on future input events.

DE simulators do not advance time to intermediary points between events because, by definition, nothing happens between events. This is an important difference to continuous-time simulators, which divide the simulation intervals into steps to ensure accuracy, as we will see in Section 2.4.

We give more details on DE simulation based on specific languages and tools in Chapter 3.

2.4 Continuous-Time Modeling and Simulation

Continuous-time and discrete-event signals differ in their time bases. In *continuous-time signals*, the time base is the set of real numbers. Figure 2.3 (a) and (b) are two examples. Signal (a) is continuous over the interval that starts at t_0 and ends at t_1. Signal (b) is piecewise continuous over the interval that starts at t_0 and ends at t_3: it suddenly changes its tangent direction at t_1 (*corner*) and its value at t_2 (*discontinuity*).

Continuous-time systems are dynamic systems that take in a set of piecewise continuous-time inputs, process them, update their piecewise continuous-time state, and produce a set of piecewise continuous-time outputs. Figure 2.1 (c) is an example. In cyber-physical systems, the analog components are continuous-time. These systems are causal.

2.4.1 Levels of Behavioral Abstraction in Continuous-Time Models

The electrical, mechanical, chemical, and other physical disciplines have each their particular design methods and tools. There is no agreed way to define the levels of behavioral and structural abstraction in CT systems as clearly as in DE systems. However, in general terms,

we can classify their behavior as linear and nonlinear [34]. *Linear systems* are simple and meet the superposition principle: we can study their response to several simultaneous inputs by dealing with one input at a time. Nonlinear systems are more complex and fail to meet this principle. One example of linear systems are electronic filters that include only linear elements, such as ideal resistors, capacitors, and inductors; their mathematical models can be solved easily both analytically and by simulation. One example of nonlinear systems is fluid flows; their mathematical models, with a few exceptions, can only be solved by simulation with big computational cost. Whenever possible, designers abstract the behavior of physical systems by neglecting their nonlinearities.

The process depends to a big extent on the application and use case. Without consideration of these factors, designers are unable to decide which variables and relations are negligible and which are crucial [34, 35]. Some methods for the abstraction of behavior are:

2.4.1.1 Linearization

Nonlinear systems can be considered linear around given operating points [36]. For example, classical tank models that relate the tank level from bottom to top to the rate of discharge of the tank's fluid are nonlinear. However, this relation is approximately linear around each level. For some applications, such as level regulation, it is possible to work with a linear model around the level of interest. Linearization works by approximation of the behavior.

2.4.1.2 Macro-modeling

Designers can focus only on the major features and neglect part of the behavior [35]. If needed, they can later add these parts to the design. An electrical circuit model, for example, can comprise only linear electrical primitives and omit parasitic currents. Macro-modeling works by selection of the relevant behavior.

2.4.1.3 Discrete-event abstraction

Some physical systems present fast physical phenomena, e.g., collisions in mechanical systems and switching in electrical circuits [8]. They give rise to nonlinear and stiff mathematical models whose parameters are often undetermined, unknown, and ununderstood. Designers can abstract away these phenomena by discrete events that split the model into two: one valid before and another valid after the phenomena. For example, instead of modeling the localized plastic deformations of two colliding balls, designers might model the collision as a discrete event in which the two balls transfer to each other the kinetic energy they had. The discrete-event abstraction works by zooming out the behavior.

These are not the only abstraction procedures, they can vary from domain to domain, and they do not always work, but when they do, they result in high-level models. These models are just accurate enough to let designers gain insight into the system rather than getting lost in the intricacies of its details. Besides, simpler models allow for performant simulations.

As it is not always possible to obtain linear or simplified models, high-level simulation must support linear and nonlinear CT models and their discrete-event abstraction.

2.4.2 Levels of Structural Abstraction in Continuous-Time Models

Given a behavior, designers describe the system structure with more or less granularity. Although there is little consensus on the levels of structural abstraction, we can list the mathematical and graphical representations that provide some structural information.

2.4.2.1 Mathematical Representations

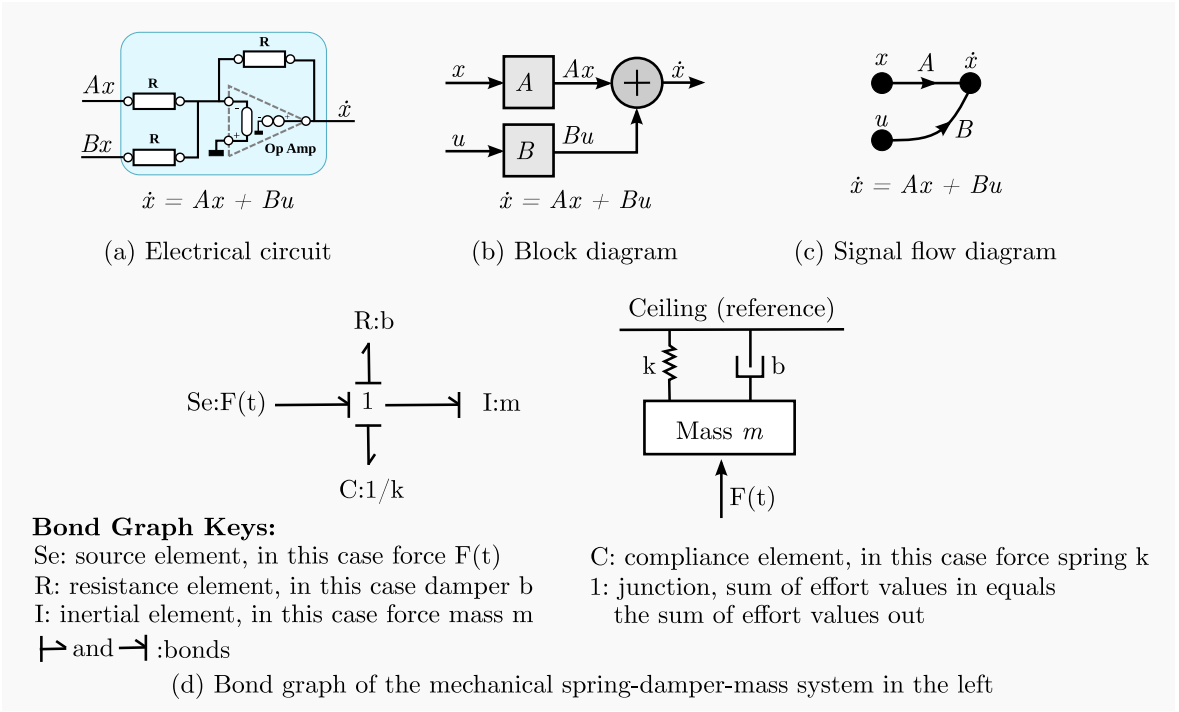
From the mathematical pespective, we can distinguish between models that only describe input/output relations and models that give information about the system state, internal constraints, and spatial distribution. The following representations are often used:

- **Transfer Functions**: they model input/output behavior. They are restricted to linear systems whose parameters do not vary with time (time-invariant). More formally, a transfer function is the ratio of the Laplace transform of the output of the system to the Laplace transform of the input of the system [34]. In this context, the Laplace transform is a mathematical transformation of a differential equation in time to an algebraic equation in frequency that is easier to analyze. However, transfer functions only relate inputs to outputs and omit information on the system structure. Different systems can have the same transfer function.

- **State-Space Representation**: it models not only the input/output behavior but also the internal system state [36]. It supports linear and nonlinear systems with time-invariant and time-varying parameters [34]; so it is more general and gives more information than transfer functions. This representation consists of a set of ordinary differential equations (ODEs) on the system state, and an output equation as a function of the state. There is a rich set of numerical methods to solve ODEs, which makes this representation ideal for simulation. We give their general form in Section 2.4.3.

- **Differential Algebraic Equations (DAEs)**: they model the input/output behavior and the internal state in the form of implicit ODEs that describe the state dynamics. These ODEs are mixed with algebraic equations that describe physical or performance constraints on the system variables. They support linear and nonlinear time-invariant and time-varying systems. DAEs are easily obtained by applying physical laws and they are more general than ODEs, but they are also more difficult to simulate [37]. Whenever possible, ODEs are preferable.

- **Partial Differential Equations (PDEs)**: they model the input/output behavior and the internal state not only as functions of time but also as functions of other variables, typically position. They give information about the effect of the spatial distribution of



(a) Electrical circuit (b) Block diagram (c) Signal flow diagram

(d) Bond graph of the mechanical spring-damper-mass system in the left

Figure 2.4: Graphical representations of continuous-time systems

the system components on the system behavior. PDEs are more general than ODEs, but they are more complex and difficult to simulate.

- **Partial Differential-Algebraic Equations (PDAEs)**: they model the input/output behavior and the internal state as a set of implicit PDEs and algebraic equations. They are to PDEs what DAEs are to ODEs. They are complex and difficult to simulate.

There are methods for going from a detailed representation to a less detailed one. These methods can either be manual or automatic. For example, to transform a PDE to an ODE, a designer can use the lumped matter abstraction [38]; to transform a DAE to an ODE, a computer can run Tarjan's or Pantelide's algorithms [37]. The state-space representation offers a good compromise between the level of details and ease of analysis, and we count with a rich set of numerical methods for their simulation.

2.4.2.2 Graphical Representations

From the graphical representation perspective, models can reveal either the topological or the computational structure or both. Some representations are:

- **Electrical Circuit Diagrams**: they model electrical systems in the form of graphs composed of electrical elements (resistors, capacitors, inductors, etc.) that are interconnected by wires that offer zero resistance, zero capacitance, and zero inductance [38]. Figure 2.4 (a) gives an example of a circuit that adds two signals $A \cdot x$ and $B \cdot x$. Each element has an associated voltage (*across quantity*) and current (*through quantity*).

These diagrams model the topological system structure and they can be automatically transformed into a mathematical model by applying Kirchhoff's laws. Equivalent diagrams for mechanical and hydraulic systems exist and they can be analyzed by applying Newton's and fluid dynamic laws.

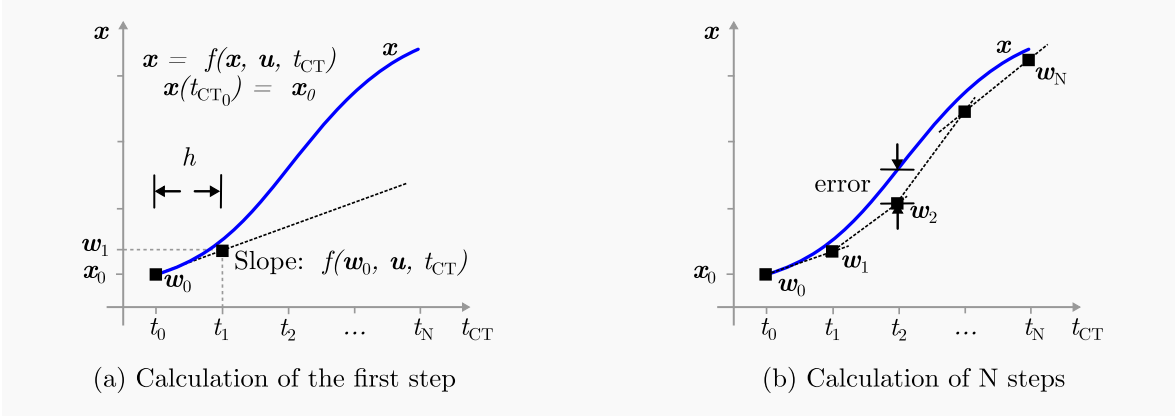
- **Block Diagrams**: they model the system as a set of functional blocks connected by directed signals (system variables). Figure 2.4 (b) gives an example of a block diagram equivalent to equation $\dot{x} = A \cdot x + B \cdot x$. Blocks apply mathematical operations on input signals to produce output signals. Each signal represents only an across, or a through quantity, but not both. Block diagrams model the computational but not the topological system structure. They can be transformed to transfer functions and state-space by using graph algorithms [39].

- **Signal Flow Graphs**: they are for the most part dual to block diagrams [39]: signals in block diagrams turn into nodes, and single input single output blocks turn into signals. Signal flow graphs are more restricted because they have no dual for blocks with multiple inputs and outputs. Nodes in signal flow graphs represent either an across or a through quantity, but not both. As block diagrams, they model the computational but not the topological system structure. Figure 2.4 (c) gives an example for equation $\dot{x} = A \cdot x + B \cdot x$. They are mostly used in control engineering and can be transformed to transfer functions and state-space by using Mason's rule [40].

- **Bond Graphs**: as opposed to the previous three, they model both, the computational and topological system structure. To this end, they represent systems by bonds that connect single port, double port, and multi-port elements. Each bond has one associated effort (across quantity) and flow (through quantity) variable that give the power (energy flow) of the system when multiplied together [39]. Bond graphs can represent physical systems and components from different disciplines based on energy analogies. Figure 2.4 (d) gives an example of a bond graph of a mechanical system. Bond graphs can be converted to state-space using graph algorithms.

These are not the only representations. Each physical discipline may model systems in one or more graphical forms. Going from a graphical to a mathematical model imposes additional computational costs.

Among these mathematical and graphical representations, the state-space representation is the most used because it is general enough to model relevant physical systems without omitting details about their internal structure. There is a rich set of numerical methods to solve state-space models. More detailed representations can be transformed to state-space either manually by the designer or automatically by the simulator. Let us define it more formally.



(a) Calculation of the first step

(b) Calculation of N steps

Figure 2.5: Numerical solution of an ordinary differential equation, adapted from [4]

2.4.3 State-Space Representation of Continuous-Time Models

We are concerned with physical system models defined in the form [41]:

$$\begin{aligned} \dot{\boldsymbol{x}} &= f(\boldsymbol{x}, \boldsymbol{u}, t_{\mathrm{CT}}) \\ \boldsymbol{y} &= g(\boldsymbol{x}, \boldsymbol{u}, t_{\mathrm{CT}}) \\ \boldsymbol{x}(t_{\mathrm{CT_o}}) &= \boldsymbol{x}_{\mathrm{o}} \end{aligned} \tag{2.1}$$

where $\boldsymbol{x} \in \mathbb{R}^n$ is the CT state, with $n \in \mathbb{N}$ and $n > 0$, $\dot{\boldsymbol{x}} \in \mathbb{R}^n$ is the derivative of the CT state, $\boldsymbol{u} \in \mathbb{R}^p$ is the CT input, with $p \in \mathbb{N}$, $\boldsymbol{y} \in \mathbb{R}^q$ is the CT output, with $q \in \mathbb{N}$, $t_{\mathrm{CT}} \in \mathbb{R}$ is the CT time, f and g are either linear or nonlinear functions, $t_{\mathrm{CT_o}}$ is the initial CT time, $\boldsymbol{x}_{\mathrm{o}}$ are the system initial conditions.

From Equation (2.1), we can compute the dynamics of the CT state and the CT outputs given the system inputs and initial conditions by using numerical integration methods.

2.4.4 Simulation of Continuous-Time Models

CT simulators monitor and maintain the CT simulation time (t_{CT}) and state ($\boldsymbol{x}$). Given Equation (2.1), the task of a CT simulator executed on a digital computer is to compute a finite set of points $\boldsymbol{w}_0, \boldsymbol{w}_1, \boldsymbol{w}_2, ...\boldsymbol{w}_{\mathrm{N}}$ at the time instants $t_0, t_1, t_2, ..., t_{\mathrm{N}}$ that approximates the exact state evolution. The system output $\boldsymbol{y}$ can be derived from it [42].

The numerical methods differ on how to approximate these points. The simplest of them is Euler's method, which defines the approximation sequence as:

$$\begin{aligned} \boldsymbol{w_0} &= x_0 \\ \boldsymbol{w_{i+1}} &= \boldsymbol{w_i} + h \cdot f(\boldsymbol{w_i}, \boldsymbol{u}, t_i) \end{aligned} \tag{2.2}$$

for $i = 0, 1, ..., N-1$. Variable h is known as the *integration step*. Figure 2.5 (a) shows an example of the computation of the first step ($\boldsymbol{w_1} = \boldsymbol{w_0} + h \cdot f(\boldsymbol{w_0}, \boldsymbol{u}, t_0)$) and Figure 2.5 (b) shows the approximated solution after computing N steps. In general, the smaller the

integration step, the more points in the interval $[t_0, t_N]$ and the more precise the approximation but the longer it takes to compute.

While in DE simulation nothing happens between consecutive events, in CT simulation anything can happen in any non-zero length interval, so the CT simulators must divide these intervals into steps of size h to ensure accurate enough solutions.

Euler's method should not be used in practice because it introduces big errors in the approximation. Multiple other methods reduce the error. They differ on how to compute each step $\boldsymbol{w_{i+1}}$ [42]:

- **Higher-order methods**: based on Taylor series expansions, they use higher-order derivatives ($\dot{f}, \ddot{f}, etc.$) instead of just f. Runge-Kutta methods are an example.
- **Varying step methods**: they modify the step size h along with simulation depending on the error at each step. The Runge-Kutta Dormand-Prince method is an example.
- **Multistep methods**: they base the computation on multiple preceding steps ($\boldsymbol{w_{i-1}}$, $\boldsymbol{w_{i-2}}$, etc.) instead of just $\boldsymbol{w_i}$. The Adams-Moulton and Adams-Bashfort methods are two examples.

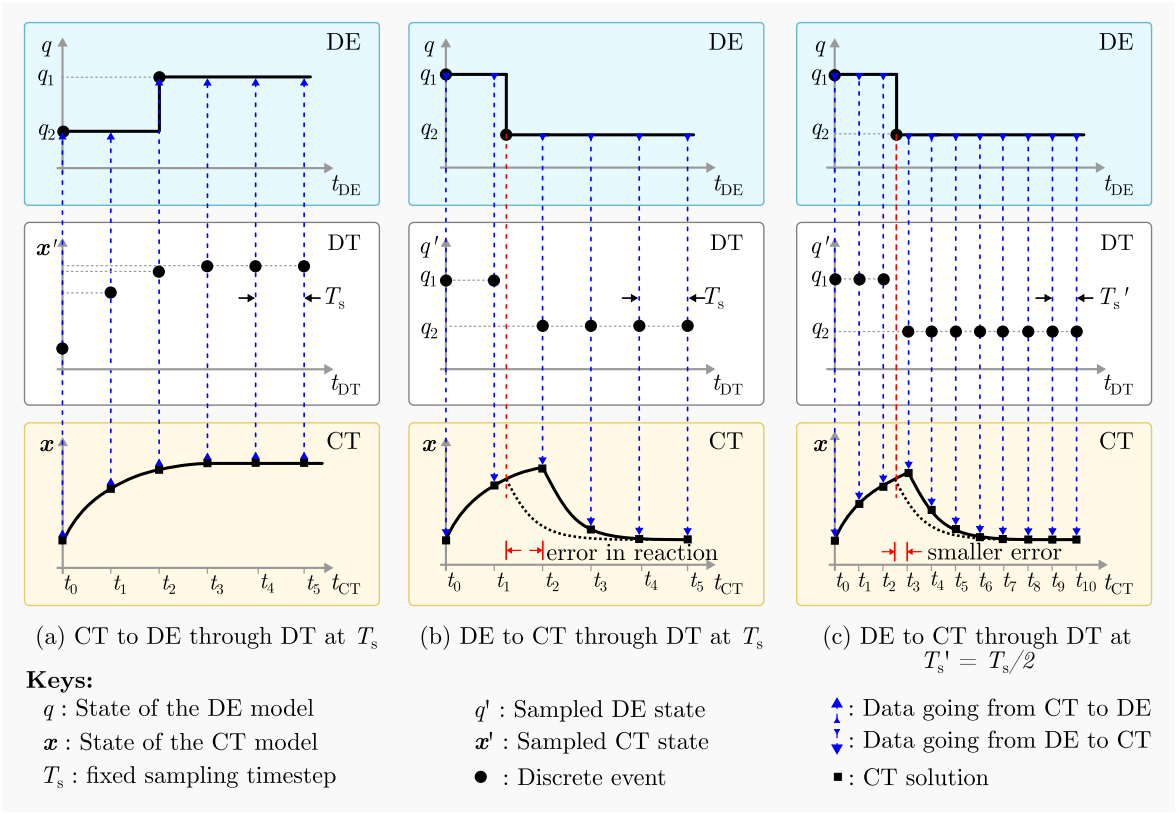
There is a rich set of numerical methods to solve ODEs and their selection depends on the application accuracy and speed needs.

2.5 The Problem of Synchronization in Continuous-Time and Discrete-Event Simulation

In cyber-physical systems and their simulations, the CT and DE parts interact to exchange data. But, as explained in Section 2.3 and Section 2.4, CT and DE signals are different. We need a way to convert one type of signal to the other. Sampling CT signals at fixed timesteps is common for signal conversion. Sampling results in DT signals. Figure 2.6 (a) shows the process. From bottom to top, the CT signal x is sampled at timepoints t_1, t_2, t_3, etc. that are separated by a timestep of size T_s to produce the DT signal x'. This signal is interpreted by the DE system, which can react to the sampled values by changing its state q, as it does at time t_2.

Similarly, a DE signal can be transformed to a DT signal from which a CT signal can be reconstructed. Figure 2.6 (b) shows the process. From top to bottom, signal q is sampled at timepoints t_1, t_2, t_3, etc. that are separated by a timestep of size T_s to produce the DT signal q'. This signal is interpreted by the CT system, which can react to the sampled values by changing the evolution of its state x, as it does at time t_2.

During simulation, these points allow the CT and DE parts to synchronize their local times ($t_{CT} = t_{DE}$). Time synchronization allows a consistent *global simulation time* advance even if the CT and DE time bases are different. The global simulation time can be taken to be either t_{DE} or t_{CT} or other time depending on the simulation engine.



(a) CT to DE through DT at T_s (b) DE to CT through DT at T_s (c) DE to CT through DT at $T_s' = T_s/2$

Figure 2.6: Continuous-time and discrete-event synchronization through discrete-time signals

However useful they might be, DT signals are a form of time abstraction and they can introduce accuracy problems. The fixed timesteps in DT signals constraint their capacity to encode important information in the original signals. Figure 2.6 (b) shows an example: an event in q occurs right after t_1, but it is not reflected in q' until t_2, and consequently, its intended effect on the evolution of x is delayed. Other examples of information that cannot be encoded are corners and discontinuities in piecewise continuous signals. In general, all that happens between timepoints is either omitted or delayed.

One way to reduce errors is to decrease the timestep. Figure 2.6 (c) shows that half the timestep creates a smaller delay in x's reaction. But this accuracy gain has a speed cost.

Small timesteps introduce too many synchronization points, as seen in Figure 2.6 (c). Most of these points are not useful, the only relevant information being the event between t_2 and t_3. But each one of them has a computational cost that accumulates. DT signals bring in an accuracy/speed trade-off: either the timestep is small, the simulation accurate but slow, or the timestep is large, the simulation possibly inaccurate, but fast.

While the level of abstraction in models depends on the use case, synchronization depends on the simulation engine. The CT/DE synchronization plays a major role in simulation accuracy and speed.

2.6 Parallel Simulation

The complexity of the simulated systems and the ubiquity of networked processors and multi-threaded environments justify the study of parallel simulation algorithms for combined CT/DE simulation acceleration.

2.6.1 Parallel Discrete-Event Simulation

The need to execute discrete events in time order to preserve system causality brings about sequential DE simulators. They are fast enough when processing few events. However, in complex models, the number of events can be very high. We can count by tens of thousands the instructions executed by a microprocessor model that runs a bare-metal control algorithm for a few seconds, for example [43]. Let alone simulations of real-time operating systems: executing millions of events sequentially is slow.

In parallel DE simulation, the goal is to process the events in parallel irrespective of their timestamps while preserving causality [44]. There are two major approaches to this aim. *Conservative* ones avoid causality errors by executing in parallel two events only if they do not affect each other; they require a way to identify the causality relations between events, which is difficult in general. *Optimistic* ones allow the execution of events in the future, detect the causality errors, and recover the simulation by rolling back the system state and remaking computations in the right order; they require to save the system state, which can be huge. Which strategy produces faster simulations depends on the application.

2.6.2 Parallel Continuous-Time Simulation

Similar to DE simulation, the motivation of parallel CT simulation derives from system complexity: their models can be too big or take too long to execute on a single processor. These complex models may implement ODEs whose evaluation is expensive over long integration intervals and that may involve many state variables.

The problem of parallel CT simulation can be treated at multiple resolutions [45]: *parallelism across the method*, which parallelizes the computations required to perform a single integration step of a given numerical method; *parallelism across the steps*, which parallelizes the computation of several integration steps of a given numerical method; and *parallelism across the system*, which parallelizes the solution of the different parts of a CT model. The selected level depends on the application.

In general, parallel simulation works best when models show a degree of data parallelism or when the simulation algorithms consist of instructions that are independent of each other. But this is rarely the case. In Chapter 3, we study other more specific problems and the state-of-the-art parallel simulation strategies to use them for CT/DE simulation acceleration.

2.7 Conclusions

In this thesis, our goal is to accelerate the modeling and simulation of cyber-physical systems to support system design use cases that demand high simulation accuracy and speed. These two aspects depend on many factors: the level of behavioral abstraction, the level of structural abstraction, the time abstraction, and the simulation and synchronization algorithms.

To this aim, we are interested in constraining the models of the different system components to be at high level of abstraction. In this context, DE models can be either at the system or algorithmic levels. CT models can be linear and nonlinear, they can abstract away fast physical phenomena by discrete events, and they are specified in state-space representation as it offers good structural details and ease of simulation. However, it is also important to consider the possibility of specifying CT graphical representations, as they ease modeling, are used in multiple physical disciplines, and can be automatically transformed to state-space.

Having defined the levels of behavioral and structural abstraction of our interest, we will focus on the exploration of the time abstraction, the simulation and synchronization algorithms, and the parallelization as possible spaces for the acceleration of high-level CT/DE simulation. In particular, we deal with the following questions:

1. What CT/DE synchronization strategies other than the intermediary discrete-time representations apply to high-level simulation and how can they be exploited to accelerate simulation?

2. How to support the simulation of models from particular physical disciplines in the form of electrical circuits and other graphical representations while still using a generic CT/DE simulation engine?

3. How to apply parallel simulation algorithms to accelerate CT/DE simulation?

Chapter 3: State of the Art

Contents

3.1 Introduction

We seek to accelerate the simulation of CT/DE models to support system design use cases that demand high simulation accuracy and speed. To this aim, we begin the chapter in Section 3.2 by studying direct CT/DE interactions as an alternative to discrete-time signals. Then, in Section 3.3, we introduce the superdense time model that makes direct interactions possible. These concepts find their applications in a set of languages and tools that we present in Section 3.4 and for which we highlight their modeling approach, supported interactions, model of time, and simulation and synchronization algorithms. We dedicate Section 3.5 to the SystemC simulator that is amply used in electronic system-level design. Then, in Section 3.6, we give an overview of the state of the art on CT and DE parallel simulation. Finally, in Section 3.7, we present our conclusions and our space of research.

3.2 Direct Continuous-Time and Discrete-Event Interactions

Although discrete-time signals are useful for modeling sampling applications they omit or delay important features in the original CT and DE signals. Direct interactions are also possible and useful. We study them in this section.

Direct interactions imply that the CT and DE models communicate via CT or DE signals. The first option is to make the DE models consume and produce CT signals, but as these signals change in a continuum, their interpretation in the DE time domain requires, in theory, an infinite amount of events in any non-zero time interval; simulation would be impractical. The second option is to make CT models consume and produce DE signals; this is feasible because CT models can interpret events to modify their state and equations, and because they can generate events when the state meets given conditions, as we explain later in this section. Direct interactions based on events require an event-aware CT simulator.

The literature describes different direct CT/DE interactions [46–49]. The work in [50] groups them into four main types:

1. **Detection and location of state events (CT to DE interaction)**: *state events* are events that occur when the CT state meets a given condition. The *state condition* takes the form of a boolean-valued function or logical predicate; for example, the predicate $\boldsymbol{x} > \boldsymbol{v}_1$ evaluates to either TRUE or FALSE depending on the value of $\boldsymbol{x}$, as seen in Figure 3.1 (a). State events are also known as *threshold crossings*. Mosterman [49] distinguishes between *detecting* a state event and *locating* its exact occurrence time.

   Temperature regulators give one example. To maintain the temperature of a certain environment around a desired set-point, when it crosses a down or up threshold (state event), a heater device turns either ON or OFF. Other examples include, in electronic circuits, diodes that open the circuit when their current is negative; in mechanical systems, frictions that prevent touching bodies to slide until a breakaway force is crossed; and in hydraulic systems, check-valves that restrain negative flows.

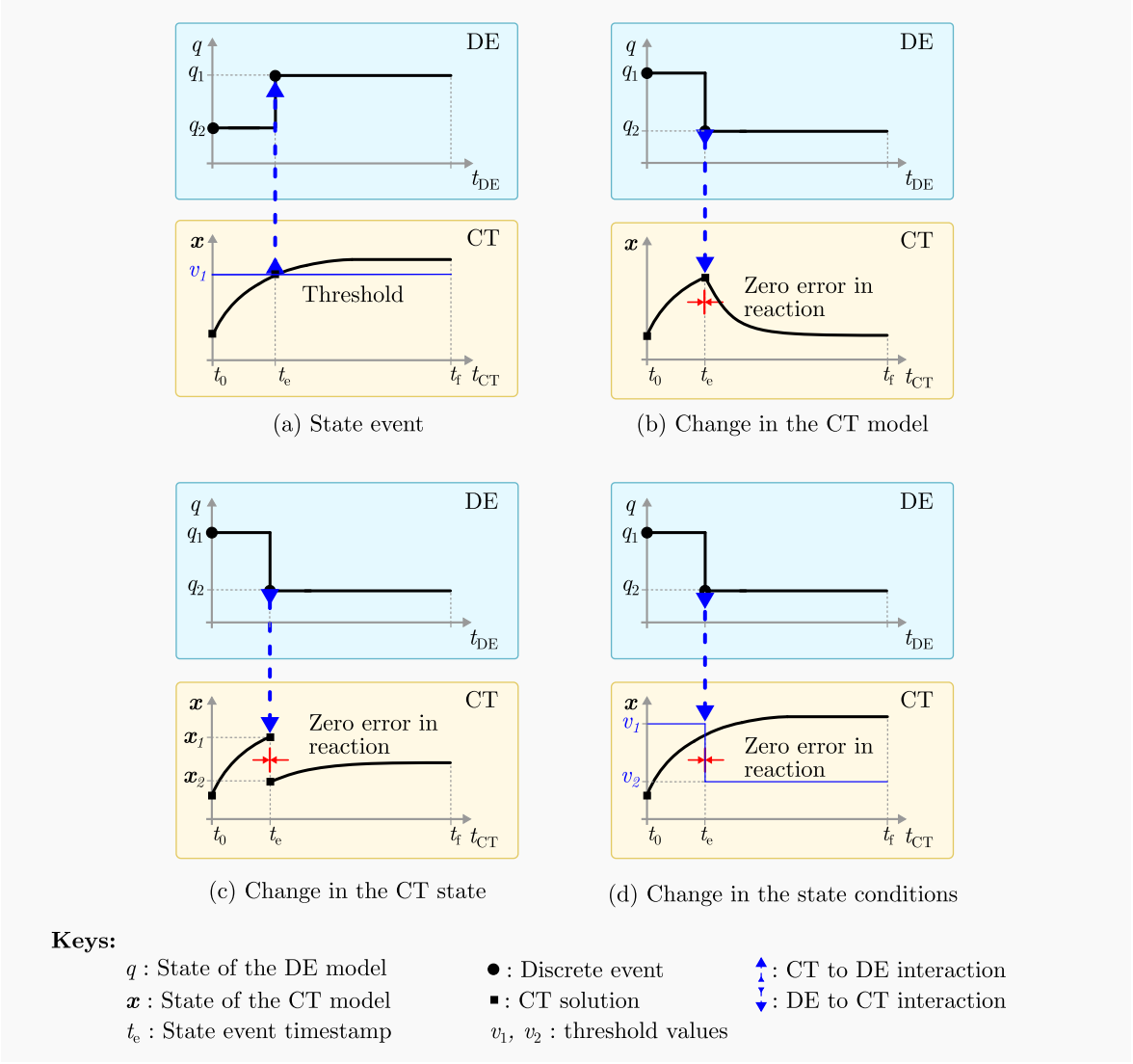


Figure 3.1: Direct continuous-time and discrete-event interactions

The concept of state events has its origin in the GASP-IV CT/DE simulator [46], in which they are distinguished from *time events* whose occurrence time is known and can be scheduled in advance. Time events are useful when the CT modules need to generate events at specific times, e.g., sampling. State and time events, as any other discrete event, can affect the DE and CT models.

2. **Instantaneous changes in the CT model (DE to CT interaction)**: discrete events can instantaneously change the CT models, causing an addition or removal of equations [49], or a modification in their parameters. As Figure 3.1 (b) shows, the equations that govern the CT state evolution from the simulation start time t_0 to the time of the event t_e are different from those after the event, from t_e to the simulation end time t_f. The event changes the CT equations and causes a corner in $\boldsymbol{x}$ at t_e. These changes are also known as *mode transitions* [8, 49].

Temperature regulators give again one example. The differential equations modeling the ON and OFF heater dynamics instantaneously change at the temperature crossing events. Other examples include, in electrical circuits, opening or closing a switch; in mechanical systems, turning ON and OFF a motor; and in hydraulic systems, opening or closing a tank filling valve [51].

3. **Instantaneous changes in the CT state (DE to CT interaction)**: discrete events can instantaneously change the CT state, causing discontinuities, as Figure 3.1 (c) shows. These changes are also known as *state resets*. The new value of the state can be explicitly given by an equation or derived by an algorithm that follows some physical constraint (e.g., a conservation law) [49].

   Zeigler [50] presents the example of leaky integrate-and-fire neurons where the event of an input synapse instantaneously adds a constant value to the neuron's membrane potential. Other examples include, in switched electrical circuits, sudden changes in inductor voltages and capacitor currents at the switching moments; and in mechanical systems, elastic collisions where the kinetic energy of one body is transferred instantaneously to another body.

   Instead of forcing the numerical algorithms to integrate over discontinuities, Cellier [47] pleads for this interaction as an error-free way of handling discontinuities. Discontinuous changes in the continuous variables imply impulses in their derivatives. Although the impulses often arise in CT modeling, they are difficult to represent numerically in the CT domain due to the presence of an infinite quantity and because of its instantaneous effect. Lee [8] states that the value of a discrete event can model the weight of the impulse, which enables a correct numerical representation. This interaction corresponds to the discrete-event abstraction of fast physical phenomena discussed in Section 2.4.1.

4. **Instantaneous changes in the state conditions (DE to CT interaction)**: discrete events can instantaneously change the state conditions. Figure 3.1 (d) shows an example where the threshold value goes from $\boldsymbol{v}_1$ to $\boldsymbol{v}_2$ at t_e.

   Temperature regulators give one example. At any moment, the desired temperature set-point can be increased or decreased, thereby changing the thresholds at which the heater is turned ON and OFF. Other examples include, in electromechanical systems, the changes in the thresholds at which a vehicle gear-box control unit upshifts and downshifts gears; and in hydraulic systems, changes in a tank level set point.

These interactions can be combined to create complex behavior. For example, Mosterman [49] discusses *event iteration* where, after a discontinuity in the state, the new state triggers further events, introducing intermediary system configurations and states. Event iteration exists in real systems such as Newton's cradle and switched circuits. Another type of complex behavior that can be present in pure CT models but that is more likely to occur in combined CT/DE models is *Zeno behavior* [8], which can be described as infinite instantaneous updates to the CT state that prevent the advance of simulation time. Zeno behavior

can be present, for example, in feedback models with zero-delay loops. A formal definition can be found in [52].

The instantaneous evolution of the CT model and state complicate simulation when using $\mathbb{R}$ as a time base. For example, in Figure 3.1 (c), the simulator needs a way to distinguish the instants at t_e at which $\boldsymbol{x} = \boldsymbol{x}_1$ and $\boldsymbol{x} = \boldsymbol{x}_2$. A different model of time is required.

3.3 Model of Time

To solve the problem of representing instantaneous changes in the CT signals, Maler et al. [48] introduces hybrid traces that allow the description of combined CT/DE behavior and that came to be later known as superdense time [53, 54].

3.3.1 Superdense Time

Cremona et al. [55] give the following definition: "A superdense time value can be represented as a pair (t, n), called a *timestamp*, where t is the *model time* and n is the *microstep*. The model time represents the time at which some event occurs, and the microstep represents the sequencing of events that occur at the same model time." We can consider $(t, n) \in \mathbb{R}_+ \times \mathbb{N}$, where $\mathbb{R}_+$ is the set of non-negative real numbers [54].

A superdense time model has the following properties:

1. **Simultaneity**: two timestamps (t_1, n_1) and (t_2, n_2) are simultaneous in a weak sense if $t_1 = t_2$ and in a strong sense if $t_1 = t_2$ and $n_1 = n_2$.
2. **Ordering**: superdense time is ordered lexicographically: $(t_1, n_1) < (t_2, n_2)$ if and only if either $t_1 < t_2$, or $t_1 = t_2$ and $n_1 < n_2$.
3. **Causality**: if an event A at (t_a, n_a) causes another event B at (t_b, n_b), then $(t_a, n_a) < (t_b, n_b)$.

3.3.2 Implementation of a Superdense Time Model in a Digital Computer

Although the definition of superdense time is useful for formal analysis, it presents some subtleties for simulation in digital computers. In particular, it requires model time t to belong to $\mathbb{R}_+$, which is represented digitally only by approximation, e.g., as a floating-point number, making it vulnerable to rounding errors. As the notion of simultaneity requires comparison of model times, rounding errors can make simulation miss simultaneous events [56].

Cremona et al. [55] show an example on how addition of floating-point time variables is not associative because of rounding errors, and proposes a digital implementation of superdense time based on integer numbers: the pair (t, n) belongs to $\mathbb{N} \times \mathbb{N}$, and a *time resolution* Δt helps to interpret the model time as a real quantity given by $t \cdot \Delta t$. For example, for $\Delta t = 0.001\,\mathrm{s}$, the timestamp $(3,\ 0)$ corresponds to $(3 \cdot 0.001\,\mathrm{s},\ 0) = (0.003\,\mathrm{s},\ 0)$. Some of its properties are:

1. **Immunity to rounding errors**: integer addition, subtraction, and multiplication by an integer factor are not prone to rounding errors and simultaneity is not at risk.

2. **Overflow handling**: although integers can overflow, a good choice of time resolution can solve the problem. For example, a 64-bit unsigned integer representation of t with a resolution of 1 femtosecond (1×10^{-15} s) supports $2^{64} \cdot 1 \times 10^{-15}\,\mathrm{s} \cdot \frac{1\,\mathrm{h}}{3600\,\mathrm{s}} \approx 5.12\,\mathrm{h}$ of simulated time.

3. **Support for different precision and simulation length requirements**: designers can choose the time resolution according to the precision and length requirements of their specific applications: simulations of fast circuit dynamics require very small resolutions and not very long simulated times, and astronomical simulations do not require very small resolutions but long simulated times. It only takes to choose 1 picosecond (1×10^{-12} s) in the preceding example to extend the simulation length to around 5120 h.

4. **Pertinent choice of the time origin**: a wise choice of the *time origin* (what time is zero time) can prevent overflow: the simulation start time is less likely to overflow than other alternatives (0h January 1, 1900, for example).

5. **Conversion to legacy representations**: superdense time can be converted to legacy floating-point representations easily (although with loss).

According to Cremona et al. [55], it is not necessary to explicitly represent the microstep, which, for a given model time, can implicitly begin at zero and increment by one at each instantaneous signal change. Superdense time has been accepted as a useful notion and it is present in theoretical as well as applicative simulation languages and tools, such as DEVS [51], hybrid automata [24], VHDL [13], and Ptolemy II [57]. Nutaro [58] studies its further formalization. We present several practical implementations in the following sections.

3.3.3 Difference between Discrete and Superdense Time

Similar to a discrete-time model, the digital implementation of superdense time also uses the set of integer numbers. However, there are major differences between both models of time. The first one is related to the discretization step that does not exist in superdense time. Whereas in discrete-time signals the time base $\mathbb{N}$ serves as the timestep and sampling must occur at each step (possibly requiring synchronization between the different parts of the simulation), in superdense time the first component of $\mathbb{N} \times \mathbb{N}$ multiplies a very small temporal resolution (nano or picoseconds, for example) and synchronization is not required to occur at every multiple of this resolution, but only at multiples in which an interaction actually occurs. Another difference is related to the second component of superdense time. It allows for instantaneous changes in signals at any given time. Such discontinuities are typically present in models combining CT and DE behavior and difficult to simulate in discrete time where the signals must have a unique value at every instant.

```
1 entity ball is
2   generic (v0, phi, air_res: REAL);
3   port (quantity x, z: out REAL);
4 end entity ball;
5
6 architecture behavioral of ball is
7   constant g: REAL := 9.81;
8   quantity vx, vz: REAL;
9   signal stopped: boolean := false;
10  -- Initial conditions
11  break x => 0.0, z => 0.0,
12  vx => v0 * cos(phi), vz = v0 * sin(phi);
13  -- Horizontal movement equations
14  x'dot == vx;
15  vx'dot == -air_res * vx * vx;
16  -- Vertical movement equations
17  z'dot == vz;
18  -- Discontinuity in the z's acceleration
19  break on stopped;
20  if stopped use
21    vz'dot == 0.0;
22  elsif vz > 0.0 use -- Ball rising
23    vz'dot == -g - air_res * vz * vz;
24  else -- Ball falling
25    vz'dot == -g + air_res * vz * vz;
26  end use;
27  -- DE process sensitive to state event on 'z'
28  ctrl: process
29    begin
30      wait until not z'above(0.0);
31      stopped <= true; -- Announce discontinuity
32      break vx => 0.0, vz => 0.0; -- Reset state
33    end process;
34 end architecture behavioral;
```

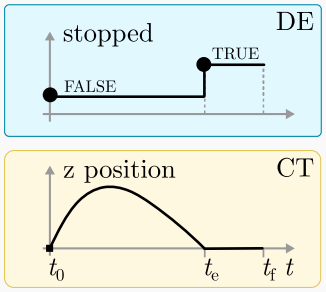


Figure 3.2: VHDL-AMS model of a flying ball, adapted from [5]

3.4 Continuous-Time and Discrete-Event Modeling and Simulation Approaches

Once we have defined the direct CT and DE interactions in Section 3.2 and established a proper model of time in Section 3.3, we explore some general CT/DE modeling and simulation languages and tools. Although the list is not exhaustive, our selection makes them differ on their provenance (industrial or academic research), abstraction levels, modeling support to physical disciplines, implementation of model of time, and simulation and CT/DE synchronization strategies. The goal is to shed light on different ways to accelerate simulation. We complement this list in Section 3.5, where we focus on SystemC-based simulators.

3.4.1 VHDL-AMS

VHDL-AMS Standard IEEE 1076™ is an extension to the VHDL Standard IEEE 1076™ hardware description language for the modeling and simulation of analog, digital, and mixed-signal systems. It supports register-transfer digital models and multiple levels of abstraction for the analog components [20, 59].

3.4.1.1 Modeling

We discuss modeling on the basis of the example in Figure 3.2. It represents of a ball that is launched at t_0 from the earth's surface at an angle `phi` to the horizontal with an initial speed `v0`. The ball moves horizontally (position `x`, speed `vx`), and vertically (position `z`, speed `vz`)

before falling back to earth at t_e. VHDL-AMS models consist of an entity and one or more architectures [59]. The *entity* (Figure 3.2, lines 1–4) describes the interface of the model in terms of its *parameters* (`v0`, `phi`, and `air_res`) and *ports* (`x` and `z`). The *architecture* (Figure 3.2, lines 6–34) specifies the implementation and it can be *structural*, as a netlist of other interconnected models, *behavioral*, as a set of statements, or both. The *statements* can be *concurrent*, for DE models, or *simultaneous*, for CT models. Examples of concurrent statements are regular VHDL signal assignment (Figure 3.2, line 31) and `process` (Figure 3.2, lines 28–33) statements . Simultaneous statements allow specifying the CT equations in the form of ODEs and DAEs (Figure 3.2, lines 14–26).

The unknown CT variables are called *quantities* (Figure 3.2, lines 3 and 8) and the CT simulator solves for their values. Related to a quantity `q` are other implicit quantities such as its time derivative `q'dot` and time integral `q'int`.

Regarding CT modeling, besides supporting equations directly specified by the designer and allowing connecting CT modules in block diagram form, VHDL-AMS also provides primitives for describing systems as circuit diagrams that follow energy conservation laws in terms of across and through quantities from which the equations are automatically derived [59].

3.4.1.2 CT/DE interactions

VHDL-AMS supports the four types of interactions described in Section 3.2:

- **State events**: the event of the quantity `q` crossing threshold `E` is informed by the boolean signal `q'above(E)`, whose value is TRUE if `q` > `E` and FALSE otherwise. Line 30 in Figure 3.2 gives an example.

- **DE changes in the CT model**: the CT statements can directly use any DE signal `s`. For the CT simulator to be sensitive to the events in `s`, the designer should declare them using the `break on s` statement, as in line 19 of Figure 3.2. In this example, a *simultaneous if statement* (lines 20-26) models the change in the equations.

- **DE changes in the state**: at the occurrence of an event, a `break` statement can set the new quantity values (line 32).

- **DE changes in the state conditions**: DE signals can be used to define the value of threshold `E` in the `q'above(E)` statements.

3.4.1.3 Model of Time

VHDL uses a $\mathbb{N} \times \mathbb{N}$ superdense time model with a variable time resolution. Microsteps are implicitly represented by the DE simulation *delta cycles* in which the simulator updates signals and executes their sensitive processes without model time advancing [60]. VHDL-AMS defines the CT time as a floating-point type called `universal_time` and a set of conversion functions to and from superdense time. Conversion ensures agreement at the CT/DE interaction times.

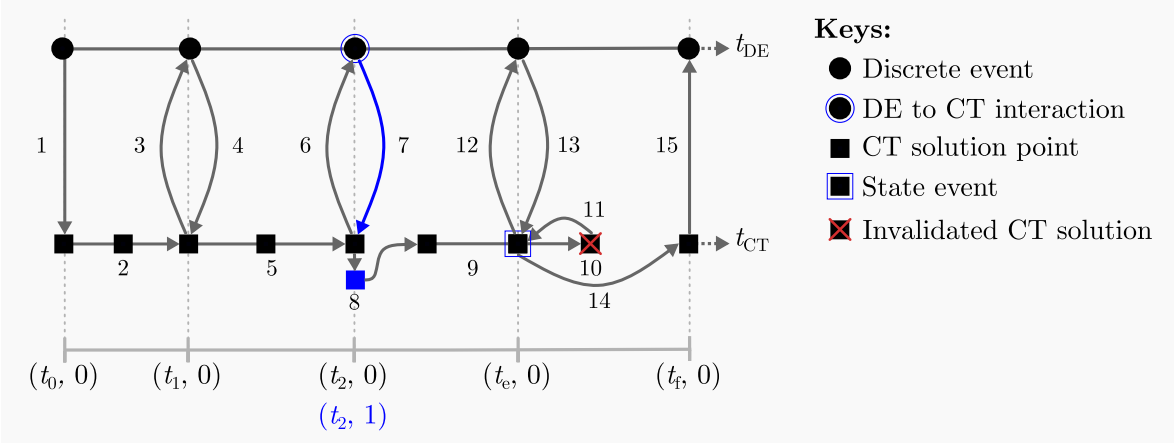


Figure 3.3: VHDL-AMS continuous-time and discrete-event synchronization

3.4.1.4 Simulation and Synchronization

Simulation consists of several phases [59]. First, the *compilation* of a model into an executable. Second, the *elaboration* of the data structures to be used during simulation. Third, *initialization*, where the simulator assigns initial signal values, evaluates all DE processes, and executes the CT solver during several delta cycles until all CT signals stabilize. And last, *time domain simulation* with alternating CT and DE model execution.

To understand how CT and DE simulation evolve and synchronize, consider the example in Figure 3.3. The DE and CT timelines are at the top and bottom, respectively. Let us assume that simulation starts at $(t_0, 0)$, that both domains are synchronized ($t_{DE} = t_{CT} = (t_0, 0)$), and that the next discrete-event timestamp is $(t_1, 0)$. We explain the figure in three parts, arrow numbers are indicated in parenthesis in the text:

Execution without interactions: first, the DE simulator lets the CT simulator resume (1) to compute solutions in the interval $[(t_0, 0), (t_1, 0)]$ (2) before giving control back (3). The DE simulator executes the events at $(t_1, 0)$. At this point, both domains are synchronized and the next event timestamp is $(t_2, 0)$.

DE to CT interaction: the CT simulator resumes (4), computes solutions from $(t_1, 0)$ to $(t_2, 0)$ (5), and gives control back (6). The DE simulator executes all events at $(t_2, 0)$. If one of these events implies an update to the CT state (DE to CT interaction), the CT simulator resumes (7) and reacts to the event by computing a new solution at $(t_2, 1)$ (8).

State Event: from the preceding step, the CT simulator tries to compute solutions from $(t_2, 1)$ to the next known event timestamp $(t_f, 0)$ (9) but, in the process, it detects a state event (10). It locates its occurrence time at $(t_e, 0) < (t_f, 0)$ (11), schedules it in the DE simulation, and gives control back (12). The DE simulator evaluates all processes sensitive to the event and lets the CT simulator resume (13) to compute solutions until the next timestamp $(t_f, 0)$ (14), the moment at which it gives control back (15). The DE simulator executes all events at $(t_f, 0)$. At this point, both domains are again synchronized.

VHDL-AMS synchronization algorithm handles all direct CT/DE interactions. Let us highlight the following points:

1. **Synchronization overhead**: the CT simulation is constrained to find solutions in the interval $[(t_c, 0), (t_n, 0)]$, where $(t_c, 0)$ is the current global simulation time and $(t_n, 0)$ is the time of the next discrete event. This is a conservative strategy known as *lock-step* [61]: synchronization always occurs at the time of the next event. What if most of the DE events do not imply CT/DE interactions? (e.g., when the DE part is complex) CT/DE simulation would be slowed down by useless synchronization.

2. **Constraints on the CT integration steps**: the next event time $(t_n, 0)$ limits the CT integration step size h used to divide the interval $[(t_c, 0), (t_n, 0)]$. This also limits the performance gains of adaptive step algorithms (Section 2.4.4). Larger intervals allow for larger step sizes and faster simulation.

3. **Portability**: this algorithm requires a modification to the original VHDL DE simulator to execute the CT simulator at the beginning of each microstep. A more portable algorithm avoids modifications to ensure it can be used on any standard implementation.

4. **Generality**: more general synchronization would not constraint the CT simulation execution to the beginning of each microstep but would assume no control in the order of execution of the processes.

This algorithm is referred to as the *canonical* algorithm [5] and there have been some proposed modifications for simulation acceleration.

Xiao et al. [6] propose to relax the constraints on the size of the CT simulation interval, which can now go from $(t_c,\ 0)$ to a time $(t_{n+\delta},\ 0)$ beyond the next event time $(t_n,\ 0)$, as seen in Figure 3.4 (a). The CT simulator provides an interpolation of the solutions at $(t_n,\ 0)$ based on the solutions at $(t_{n-\delta},\ 0)$ and $(t_{n+\delta},\ 0)$. For the next execution, the CT simulation resumes at $(t_{n+\delta},\ 0)$ until a time beyond the next discrete event or until finding a state event. This is an *optimistic* approach that assumes that the events at $(t_n,\ 0)$ do not imply any DE to CT interaction. If they do, the CT simulator discards the values at $(t_{n+\delta},\ 0)$, rolls back to $(t_{n-\delta},\ 0)$, and recomputes the solution precisely at $(t_n,\ 1)$ (Figure 3.4 (b)). The authors argue the algorithm to be efficient because it allows larger integration step sizes. They omit the discussion on error control in the interpolated solutions, which may affect accuracy.

The authors of [7, 62] propose to reduce the synchronization overhead by distinguishing the discrete events that imply CT/DE interactions from those that do not. They change the canonical algorithm in two ways: first, the CT simulation now executes from $(t_c,\ 0)$ to the time of the next DE to CT interaction $(t_{n'},\ 0)$ instead of the time of the next event $(t_{n_1},\ 0)$, as seen in Figure 3.4 (c). Second, the CT simulation stops at state events only if they can generate back, directly or indirectly, a DE to CT interaction, otherwise, CT simulation continues until $(t_{n'},\ 0)$ and schedules the located state events in DE simulation only before returning control; DE simulation resumes at the time of the earliest scheduled event, as Figure 3.4 (d) shows. These changes prevent unnecessary synchronization at the events between $(t_c,\ 0)$ and $(t_{n'},\ 0)$. The authors state that the speed-up depends on the density of discrete events implying

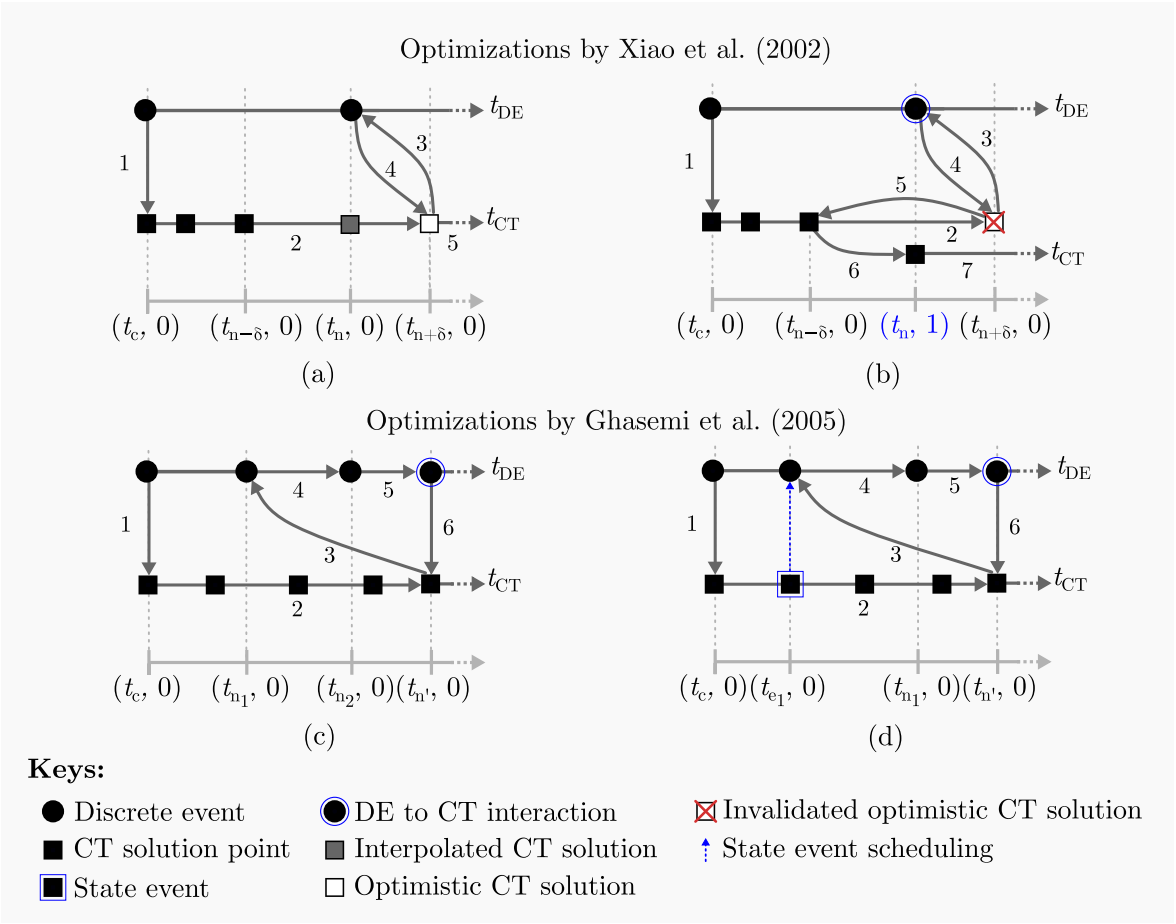


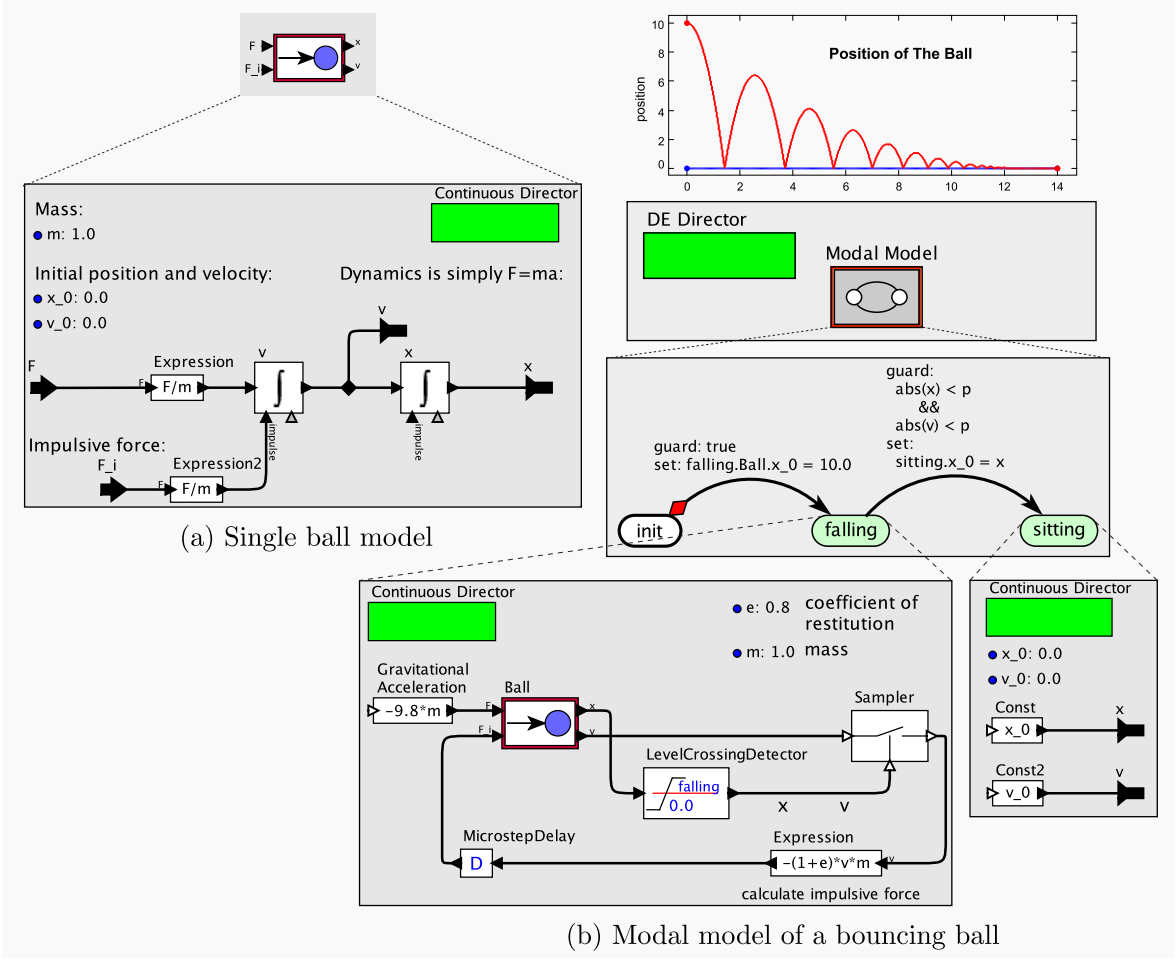
Figure 3.4: Optimizations to VHDL-AMS synchronization, adapted from [6, 7]

DE/CT interactions—the higher the density, the lower the performance. This solution avoids useless synchronization by identifying the events linking both domains before simulation.

These algorithms exploit the integration step sizes and the nature of the events to accelerate simulation.

3.4.2 Verilog AMS

Verilog AMS [21] is an extension to Verilog HDL for high-level behavioral and structural descriptions of combined DE and CT systems. Verilog-AMS and VHDL-AMS can be regarded as similar languages because they allow modeling the same type of systems. The work in [22] gives a detailed comparison. Here, we highlight only Verilog-AMS's major characteristics. Verilog AMS preserves the standard Verilog DE **modeling** and extends it to support CT block and circuit diagram representations. Verilog AMS supports all **CT/DE interactions**: the CT part can read DE variables, react to discrete events, and generate state events; and the DE part can read CT variables, react to state events, and generate events intended to modify the CT part. Verilog AMS supports a superdense **model of time** similar to that of VHDL-AMS. Finally, regarding **Simulation and synchronization**, Verilog AMS implements a synchronization algorithm that admits an *optimistic* CT execution beyond



(a) Single ball model

(b) Modal model of a bouncing ball

Figure 3.5: Model of a bouncing ball in Ptolemy II, adapted from [8]

the DE simulation time and backtrack in the case that the skipped events affect the CT simulation. In the following section, we illustrate more in detail the operation of an optimistic CT execution by taking Ptolemy II's synchronization as an example.

3.4.3 Ptolemy II

Different from VHDL-AMS and Verilog AMS that are industrial tools, Ptolemy II is an academic research tool. It enables modeling and simulation for the design of systems that integrate multiple domains [57].

3.4.3.1 Modeling

Ptolemy II supports different modeling domains. A *modeling domain* is the concrete implementation of a *model of computation* (MoC), a collection of rules that specify what is a component, the execution mechanisms of a set of components, and how they communicate. The notion of modeling domain is different from that of a time domain and includes continuous-time and discrete-event models, but also models with more restricted syntax and semantics such as dataflow, state machines, etc.

Ptolemy II's components are known as *actors*, execute concurrently, and communicate by sending messages through signals via *ports*. Actors can be *atomic* or composed of other actors (*composite*). A *director* drives the execution of a set of actors and makes their MoC explicit. Different MoCs can interoperate. We limit our discussion to combined CT/DE models.

Figure 3.5 (a) gives the example model of a ball that falls from an initial height, collides with a surface, and bounces up, losing kinetic energy at each collision and eventually stopping. The single ball model is a composite actor. It has two input ports, force `F` and impulsive force `F_i`, and two output ports, position `x` and speed `v`. Its director is CT. Figure 3.5 (b) is an example of a CT/DE model that contains a modal model at the top level (DE). Modal models represent finite sets of behaviors (*modes*) and their transitions. Transitions activate when a *guard* predicate evaluates to TRUE. Each mode can be refined to a different MoC model. For example, the `falling` and `sitting` modes in the figure are CT.

Regarding CT modeling, designers specify ODE system models by using `Integrator`, `Expression`, and other actors that relate CT signals in a block diagram form.

3.4.3.2 CT/DE interactions

Ptolemy II supports the four types of interactions described in Section 3.2:

- **State Events**: the `CTEventGenerator` class of actors converts CT input to DE output signals. They generate events when a CT input crosses a threshold. The `LevelCrossingDetector` actor in Figure 3.5 (b) illustrates the detection of the ball's collision event.

- **DE changes in the CT model**: transitions between modes in modal models are DE-driven and different modes can contain different sets of equations. Figure 3.5 gives an example: the `falling` and `sitting` modes moidel the ball before and after it has lost all of its kinetic energy.

- **DE changes in the state**: `Integrator` actors have an optional DE impulsive input whose event values instantaneously set the integrator's CT state. Figure 3.5 gives an example: the ball's speed is set by the impulsive force `F_i`.

- **DE changes in the state conditions**: `Expressions` involving DE signals can modify the inputs and parameters of the `CTEventGenerator` actors.

3.4.3.3 Model of Time

Ptolemy II uses a $\mathbb{N} \times \mathbb{N}$ superdense time model. It implements the model time by an arbitrarily large integer and represents the microstep by a 32-bit integer. The time resolution is a double-precision floating-point number set by the designer and common to all components.

3.4.3.4 Simulation and Synchronization

Ptolemy II simulation consists of three phases: *setup*, which initializes actors; a sequence of *iterations*, where actors read data, update their state, and output data; and *wrapup*, which ends the execution of an actor [63]. During the iterations phase, actors test a set of conditions for execution (*prefire*); if they are met, the actors execute to read inputs and produce outputs (`fire`), and update their state (`posfire`). These operations are part of Ptolemy II's *abstract semantics*: a set of execution and communication phases common to all actors and directors independently of their MoC. Actors and directors decide on how to implement each phase.

We are concerned with the `CTMixedSignalDirector`, which handles the CT actors that interact with the DE domain the following way [64]:

1. `setup`: ask for the first activation to ensure CT execution ahead of the global time.
2. `prefire`: handle rollbacks produced by discrete events if needed.
3. `fire`: execute in two phases separated by a microstep. First, in the *event phase*, consume input events, generate output events, and request a reactivation at the next microstep. And second, in the *execution phase*, handle the DE changes in the CT state, save the CT state (*checkpoint*), and integrate the equations until the end execution time or until finding a state event.
4. `posfire`: undefined. The director saves the CT state during `fire`.

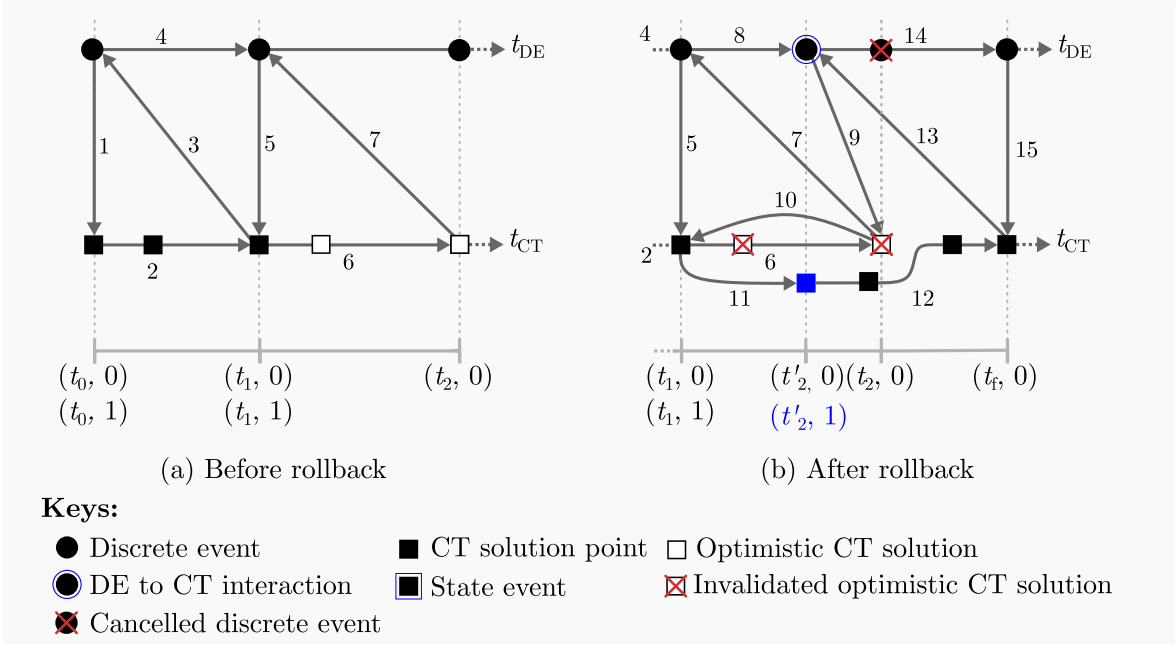


Figure 3.6: Ptolemy II continuous-time and discrete-event synchronization

Figure 3.6 illustrates synchronization when a CT model is embedded in a larger DE model. Let us assume that the DE model is at the top level—t_{DE} is regarded as global—,

that simulation starts at $(t_0,\ 0)$, that both domains have been initialized (`setup` has been already invoked), that they are synchronized ($t_{\mathrm{DE}} = t_{\mathrm{CT}} = (t_0,\ 0)$), and that the next discrete-event timestamp is $(t_1,\ 0)$. In what follows, we refer to the DE director as the DE simulator and the CT director as the CT simulator. For clarity reasons the double arrows corresponding to the `prefire` and `fire` activations at each time are represented by the same arrow. Let us explain the figure in three parts—arrow numbers are indicated in parenthesis in the text, e.g., (1) designates arrow 1.

Execution without interactions (Figure 3.6, (a)): the DE simulator lets the CT simulator resume (1), which identifies no rollback need at `prefire`, executes the event phase of `fire`, and requests a reactivation at the next microstep. The DE simulator continues execution of the events at $(t_0,\ 0)$, advances global time to $(t_0,\ 1)$, and lets the CT simulator resume once again (same arrow, 1). The CT simulator continues with the execution phase of `fire` until the time of the next event $(t_1,\ 0)$ (2) and gives control back to the DE simulator (3), which continues the execution of events at $(t_0,\ 1)$ and then advances time to $(t_1,\ 0)$ (4).

Computation of optimistic solutions (Figure 3.6, (a)): the DE simulator lets the CT simulator resume (5), execute `prefire`, the event phase of `fire`, and give control back. The DE simulator advances global time to $(t_1,\ 1)$ and lets the CT simulator continue the execution phase of `fire` (same arrow, 5) to create a checkpoint and compute new solutions optimistically until the next known event time $(t_2,\ 0)$ (6). It then gives control back (7).

DE to CT interaction (Figure 3.6, (b)): the DE simulator resumes at $(t_1,\ 1)$, and executes the rest of events at this time. One of these events cancels the event at $(t_2,\ 0)$ and schedules a new event implying a DE to CT interaction at $(t_2^{'},\ 0) < (t_2,\ 0)$; the DE simulator advances time to $(t_2^{'},\ 0)$ (8) and lets the CT simulator resume (9). At `prefire`, the CT simulator rolls back (10) because the event invalidates the optimistically calculated solutions. It recomputes solutions from the last checkpoint at $(t_1,\ 1)$ to the current global time $(t_2^{'},\ 0)$ (11), moment at which it executes the event phase of `fire`, requests a reactivation at the next microstep, and gives control back to the DE simulator, which advances time to $(t_2^{'},\ 1)$ and lets the CT simulator resume (we have omitted the corresponding arrows for space and clarity reasons). The CT simulator continues execution to $(t_{\mathrm{f}}, 0)$ (12), and gives control back to the DE simulator (13), which advances time (14), and resumes the CT simulator (15). At this point, both simulators are again synchronized.

State event detection is similar to the one in VHDL-AMS. Some other characteristics that are important to highlight are:

1. **Optimistic execution:** the CT simulator advances its local time t_{CT} beyond t_{DE} until either it reaches the next event time or it detects a state event or the time difference $t_{\mathrm{CT}} - t_{\mathrm{DE}}$ reaches a maximum given by the `runAheadLength` simulation parameter.

2. **Need to rollback**: events before t_{CT} can invalidate the optimistic solutions, so the CT simulator needs to create checkpoints and to be able to roll back.

3. **Causality**: the CT simulator detects and locates state events in a way similar to

VHDL-AMS, but they are output only when t_{DE} advances to their timestamps so that there is no rollback risk that may compromise their validity.

4. **Portability**: this algorithm is portable because it allows the top-level DE simulator to see the CT simulator as a consumer and generator of discrete events, the internal CT simulation aspects being transparent and requiring no change in the DE simulator.

5. **Generality**: it is more general than the VHDL-AMS algorithm because it assumes no order of execution of the CT simulator w.r.t. to the discrete events at any given time.

However, choosing the next synchronization time to be at most the next event time makes it encounter the same synchronization overhead problems and introduces constraints in the CT integration steps similar to those of the VHDL-AMS algorithm.

3.4.4 Discrete-Event System Specification Simulators

Also from the world of academic research, the work presented in [51] describes three formalisms for the specification of models in different time domains: the Discrete-Event System Specification (DEVS) formalism for DE models, the Discrete-Time System Specification (DTSS) formalism for DT models, and the Differential Equations System Specification (DESS) formalism for CT models. Regarding **modeling**, DEVS provides a mathematical framework for describing and simulating hierarchical DE models in terms of their inputs, states, outputs, and functions for state transitions (which can be triggered by internal states and external events), output generation (which maps the state to the outputs), and time advance. Different general-purpose simulators have been implemented on top of DEVS, such as PowerDEVS and ADEVS [65–67]. The DEVS framework can be used as a base to model and simulate hybrid models. In this direction, one important work is the one described in [68], which introduces four new modeling formalisms for combined CT/DE systems. First, it specifies the semantics of coupling two components from different formalisms out of the three DEVS, DTSS, and DESS to enable combining DE, DT and CT components in a modular and hierarchical way. Second, it formalizes atomic DEVS systems with some continuous inputs in which only a finite set of values is able to produce input events, thus avoiding an infinite number of events caused by its the continuous evolution. Third, it formalizes atomic DESS&DEVS systems that are CT systems with state and time events that cause discontinuities in the CT state and outputs. And fourth, it formalizes atomic DEVS&DESS systems in which the main dynamics is DE but that can also contain a CT state. DEVS&DESS is the precursor of more recent work embedding CT representations inside DEVS models [50]. These embedded models contain the system **equations** and can be interconnected via DE signals. DEVS and its embedded CT models support the four types of **CT/DE interactions** described in Section 3.2. The **model of time** in DEVS is superdense. For **simulation and synchronization**, CT models are simulated as regular DE models. Their equations can be integrated by numerical algorithms.

An alternative and novel approach for integrating CT models in DEVS are Quantized State Systems (QSS): instead of computing the CT state at the next integration step, QSS quantizes the state and computes the time at which it passes from one constant value to the next (advancing a quantum). Advancing a quantum can be considered as an event. A QSS integrator can calculate in advance the time of occurrence of these events while avoiding rollbacks [69]. The authors report speed-ups of around $20\times$ w.r.t. numerical integration algorithms. QSS simulators have been tested mostly in the context of academic research. Industrial tools mostly rely on numerical methods for which research is more extensive.

3.4.5 Synchronous Languages Based CT/DE Simulators

Also from research, the work in [70] tackles the problem of enabling the specification of discrete models including CT components on top of synchronous languages. Models in these languages are discrete time and can be statically analyzed to avoid unsafe behavior. The goal is to use static analysis to guarantee important safety properties in models containing CT components modeled as **differential equations** and supporting **direct interactions**: that the CT signals are never used in the place of DE signals and the absence of algebraic loops. Another goal is to generate statically scheduled code for simulation. Their approach is based in an alternative to superdense time known as nonstandard analysis [71], in which the CT signals are treated as if they were discrete and each step is considered as having infinitesimal length. The consequence is that of transforming CT differential equations into difference equations (their discrete counterpart) to enable symbolic reasoning and static analysis. They use a type system that expresses causality relations between signals and that enables to reject, during compilation, wrongly typed models (models combining signals in incomparable time domains with unspecified semantics) and models containing causality loops. Although they use nonstandard semantics during compilation, they argue that it is a theoretical construct difficult to implement in practice and adopt **superdense time** for simulation. Regarding **simulation and synchronization**, their solution is able to calculate a static schedule for the sequence of actions occurring in the CT/DE interactions—but not for the interactions, as their time cannot be known in advance and is determined only during simulation. As it is not the main focus of their study, their synchronization loop is not leveraged for simulation acceleration. On this same line of research, other works deal with modeling and simulation of CT components given as DAEs on top of synchronous languages [72].

3.4.6 MATLAB/Simulink

Finally, we give a brief description of MATLAB/Simulink, a language for mathematical modeling and simulation [16, 73, 74]. For **modeling**, MATLAB provides functions for numerical integration of ODEs and PDEs. On top of it, Simulink enables CT/DE modeling and simulation at high level of abstraction. Regarding CT modeling, it supports the specification of equations, block diagram, and circuit diagram representations. MATLAB/Simulink supports the four types of **CT/DE interactions** described in Section 3.2, for example, by threshold

crossing detection. No explicit information about the **model of time** is available in the documentation. Finally, **simulation and synchronization** consists of alternating CT and DE simulation. The documentation omits details about the synchronization mechanisms. The strong aspect of MATLAB/Simulink lies in its support to specialized domains via libraries of components. This makes it one of the preferred choices during early design stages as a complementary language to other more specialized system-level simulators such as SystemC [75]. We use MATLAB/Simulink as a reference to compare the precision of our simulations.

3.4.7 Partial Conclusions

Several industrial and academic tools provide combined CT/DE modeling and simulation support. They typically extend their DE modeling approach to the CT domain in the form of equations, block diagrams, and circuit diagrams. They support direct CT interactions and a superdense time model. Additionally, combined CT/DE simulation can be conservative, preventing rollbacks but with likely reduced performance due to increased synchronization and smaller integration steps, and optimistic, with less synchronization and bigger integration steps but having to handle eventual rollbacks. These algorithms vary in terms of their portability and generality and may require changes or not in the DE simulators.

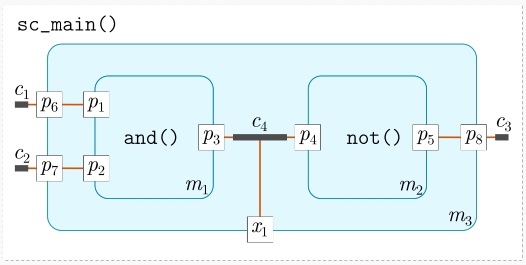
3.5 SystemC-Based Modeling and Simulation Approaches

The SystemC 1666-2011 IEEE standard is a C++ class library for hardware and software system design at the system and RTL levels [15]. SystemC is the reference tool in the Electronic System Level (ESL) design community for modeling and simulation at high level of abstraction. We devote an entire section to SystemC because of its standardized support to virtual prototyping via the Transaction Level Modeling (TLM) extensions and to CT modeling via a multitude of extensions of which the Analog and Mixed Signal (AMS) 1666.1-2016 IEEE standard is the most widely used.

In this section, we describe SystemC modeling, model of time, simulation, and the details of some of its CT/DE extensions.

3.5.1 Modeling

SystemC models consist of interconnected modules that communicate data through channels via ports. Figure 3.7 is an example that shows some of the SystemC modeling elements. *Modules* (class `sc_module`) are units that have certain functionality; components m_1 and m_2 are modules, m_2 is an instance of the `NotMod` class. *Processes* are functions executed on the *notification* of events to which they are sensitive; m_2 contains a process that computes the logical `not` of a boolean. *Ports* (class `sc_port`) are inputs and outputs of a module; p_1, p_2, ..., p_8 are ports. *Channels* (class `sc_prim_channel`) are interconnections between modules; c_1, c_2, c_3, and c_4 are channels. A channel implements one or more interfaces (`sc_interface`), which are abstract classes that declare the available channel methods (`write`, `read`, etc.).



Model of the boolean equation $c_3 = \texttt{not}(c_1 \text{ and } c_2)$

Keys:

m_1, m_2, m_3 : modules	—: port and export bindings
p_1, p_2, ..., p_8 : ports	`not()`, `and()`: processes
c_1, c_2, ..., c_4 : channels	`sc_main()`: main function
x_1 : export	

```
1  SC_MODULE(NotMod){
2      // Ports
3      sc_in<bool> p4_in;
4      sc_out<bool> p8_out;
5
6      SC_CTOR(NotMod) { // Constructor
7          SC_METHOD(not_process);
8          sensitive << p4_in.default_event();
9      }
10
11     void not_process() {
12         bool result = !p4_in.read();
13         p8_out.write(result);
14     }
15
16 };
```

Figure 3.7: Model of a boolean equation in SystemC and sample code of a module implementing a logical `not`

Ports are typically connected or *bound* to channels. And *Exports* (class `sc_export`), which expose a channel that is internal to a module to other external modules; x_1 is an export.

Models have a related *module hierarchy* composed of the set of objects of class `sc_module`, `sc_port`, `sc_export`, and `sc_prim_channel`. They are all, in turn, derived from the `sc_object` class. For any given `sc_object`, designers can obtain the child objects it contains (`get_child_objects`) and the parent object containing it (`get_parent_object`) [15]. For example, in Figure 3.7, p_1, p_2, and p_3 are children of m_1, which in turn is a child of m_3.

The SystemC TLM extensions support virtual prototyping [15]. TLM decouples computation from the component communication interfaces with a two-fold aim: first, to ensure interoperability independently of the internal implementations, and second, to speed up simulation by loose communication (*transactions*). Generally, TLM components can be initiators, which initiate transactions, targets, which respond to transactions; interconnect, which carry transactions, or combined initiator/targets. In the search for speed-up, TLM introduces temporal decoupling to save synchronization overhead. TLM defines two coding styles that can be chosen according to the application: *loosely-timed*, implemented by *blocking transport interfaces* that model only the transaction start and end times; and *approximately-timed*, implemented by *non-blocking transport interfaces* that model transaction start and end times, and multiple times in between. Apart from blocking and no blocking transport, TLM provides the Direct Memory Interface (DMI) that enables initiators to directly access memory addresses in the target modules, saving interconnect processing time.

3.5.2 Model of Time

SystemC's time is global to all components and can be interpreted as an $\mathbb{N} \times \mathbb{N}$ superdense time model. The model time is realized as an unsigned integer of at least 64 bits. We can consider that the microsteps are implicitly represented by the DE simulation *delta cycles*,

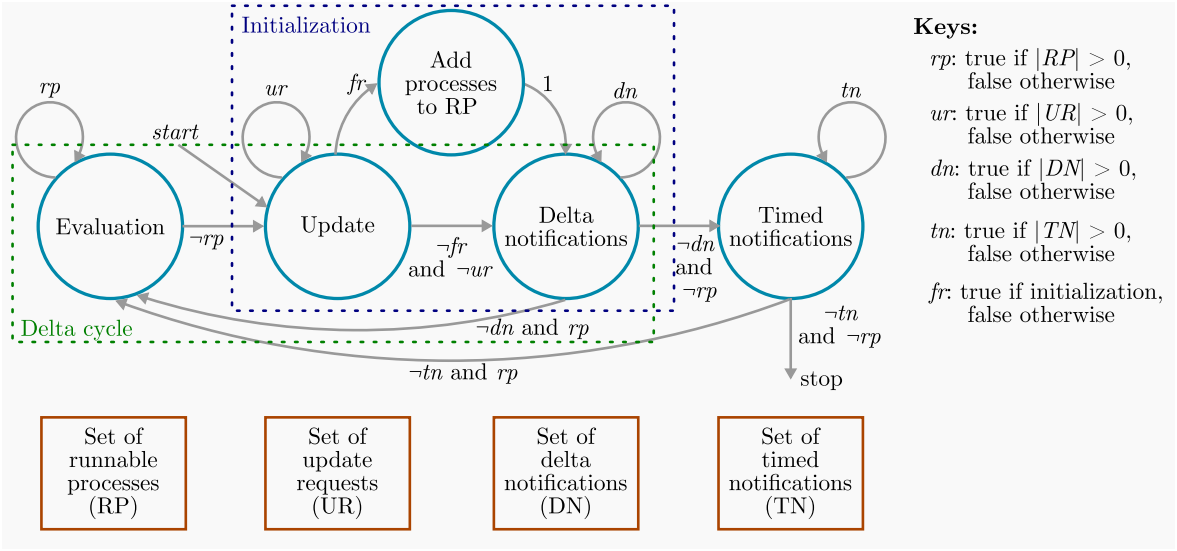


Figure 3.8: SystemC's scheduler, adapted from [9]

similar to VHDL-AMS. The time resolution is a global value set by the designer.

3.5.3 Simulation

SystemC simulation has two main phases: elaboration and simulation. During *elaboration*, the simulator sets the time resolution, instantiates modules and channels, registers processes and their sensitivities, and makes port and export binding; this phase results in the construction of the module hierarchy and a set of data structures needed for simulation. During *simulation*, a *scheduler* executes the model according to the following phases (Figure 3.8):

1. **Initialization**: when the simulation starts, the scheduler updates channels to their initial values (Update), schedules all processes for their first execution (Add processes to set of runnable processes), and processes the delta notifications that result from channel update (Delta notifications), as explained in the items below.

2. **Evaluation**: the scheduler selects and removes processes from the *set of runnable processes*, which contains all processes sensitive to the events at the current simulation time. It evaluates processes one by one and in no particular order. Each process decides when to yield simulation control back (*cooperative multitasking* a.k.a. *non-preemptive execution*). Processes can read from and write to channels. They can also schedule new events immediately, at the next delta cycle, or at a future time. When there are no more processes left in the set, the scheduler goes to the update phase.

3. **Update**: the scheduler keeps track of a *set of update requests* on the channels that have been written during process evaluation. It treats requests to update channel values and schedules events at the next microstep.

4. **Delta notifications**: the scheduler keeps track of the *set of delta notifications* that contains the events scheduled at the next delta cycle. The scheduler handles delta

notifications by adding the sensitive processes to the set of runnable processes. At the end of this phase, if there is at least one process in the set, the scheduler executes a delta cycle (Evaluation followed by Update followed by Delta notifications), otherwise it goes to the timed notifications phase.

5. **Timed notifications**: the scheduler also keeps track of a *set of timed notifications* that contains the events scheduled in the future. It advances the global simulation time to the next event time, removes all events at this time, and adds the sensitive processes to the set of runnable processes. If there is at least one process in the set, it goes back to evaluation, otherwise, it stops the simulation.

Designers can interfere with elaboration and simulation by invoking simulation callbacks. For example, the `end_of_elaboration` callback allows performing actions that depend on the module hierarchy construction and port binding to have finished, such as verifying the module hierarchy [15]. This is profitable for the extension of SystemC to CT simulation.

3.5.4 Continuous-Time Modeling an Simulation Extensions

Given SystemC's support for modeling and simulation of hardware and software systems, and its adoption in the ESL design community, there have been multiple tentatives to extend it to other domains [76–83]. The following frameworks target the CT time domain.

3.5.4.1 SystemC A

Extension for modeling and simulation of linear and nonlinear CT systems [76, 84, 85]. It follows the SystemC **modeling** approach. Designers specify CT models in the form of equations or electrical circuit diagrams. The simulator extracts DAEs from the circuits. It can also simulate distributed systems (PDEs). Regarding the **CT/DE interactions**, SystemC A supports state events from threshold crossings via its `interfaceAD` module. Howevern, for DE to CT interactions, it only provides the `interfaceDA` module that converts a DE boolean input to a CT double output by smoothing the sudden discrete changes, which implies no support for the instantaneous interactions. Although SystemC time can be considered superdense, the extension does not profit from it as a **model of time**. Regarding **simulation and synchronization**, the SystemC scheduler runs the CT simulator as a SystemC process, synchronization occurs in locked steps determined by the minimum time between the next event and the next CT solver integration step. Although SystemC-A detects state events, it does not implement any algorithm to exactly locate their occurrence time, which may harm precision. It modifies the standard SystemC scheduler.

3.5.4.2 SystemC AMS

The SystemC AMS 1666.1-2016 IEEE standard is a C++ class library [23] that extends SystemC to combined DE, DT, and CT modeling and simulation for analog and mixed-signal system design. It is used in industrial and academic research and development [25].

SystemC AMS follows the SystemC **modeling** approach. SystemC AMS defines three models of computation, each of which offers a set of primitive modules for computation and channels and ports for communication that restrict models to particular domains. *Timed Data Flow (TDF)* describes models in the DT time domain, each module defines a sample *processing* function and activates when a predefined number of samples are present at their input ports, processes them, and produces a predefined number of samples at their output ports. *Linear Signal Flow (LSF)* describes models in the CT time domain in the form of block diagrams from which the SystemC AMS simulator obtains equivalent linear DAEs. *Electrical Linear Networks (ELN)* describes models in the CT time domain in the form of circuit diagrams, from which the SystemC AMS simulator obtains equivalent linear DAEs.

Regarding the **CT/DE interactions**, TDF interacts with DE via the `sca_tdf::sca_de::sc_in` and `sca_tdf::sca_de::sc_out` conversion ports. Input ports interpret events as samples and output ports generate events from samples at fixed timesteps. TDF interaction with DE is discrete-time-based. LSF and ELN communicate with DE modules via SystemC signal ports whose values are constant between reactivations. Reactivations occur at timesteps fixed by the designer or derived from interconnections to the TDF modules. LSF and ELN interaction with DE is also discrete-time-based.

The TDF, LSF, and ELN signals are defined in only one temporal dimension, so they do not profit from a superdense **model of time**.

Finally, regarding **simulation and synchronization**, simulation of TDF modules occurs according to a static activation schedule that makes their execution fast. Simulation of the LSF and ELN modules occurs by a linear DAE solver. Synchronization of all MoCs with the DE simulation is DT and occurs via a TDF synchronization layer at fixed steps. To overcome the disadvantages of fixed-step synchronization, SystemC AMS introduces the Dynamic Timed Data Flow (DTDF) paradigm that defines the following functions [35]:

- `change_attributes`, called during the simulation and used by designers to specify the rules for deciding on module activation (`request_next_activation`) and timing attribute changes (`set_max_timestep`).
- `request_next_activation`, used to request the next activation at a future time point or event.
- `set_max_timestep`, used to force reactivation at a maximum step.

Although DTDF has proved helpful in many use cases, it provides no mechanism to detect state events. In addition, it relies on the user to explicitly specify the rules to calculate the successive synchronization steps. Finally, these functions are only defined for TDF modules. LSF and ELN only synchronize via a static TDF layer.

There exists an increasing interest in supporting virtual prototyping and efficient multi-domain simulation strategies on top of SystemC. However, the AMS extensions do not offer mechanisms to support TLM communication. For this reason, Damm et al. [86] present an approach to connect TDF and loosely timed TLM 2.0 modules. They identify two major

challenges. The first one refers to converting TLM transactions with variable rates to and from fixed-rate TDF signals. The second challenge is to ensure processing transactions in time order: they can be initiated from different processes, each of which maintains its own local simulation time that differs from the global simulation time by a quantum; these transactions can target the same TDF module and arrive with unordered timestamps. Their solution is based on TLM to TDF and TDF to TLM converter modules. To solve the first challenge, the TLM to TDF converters include a buffer to store transactions when they are frequent. Converters process buffered transactions at a regular rate to write data to the TDF signal. When there are not enough transactions to maintain the data rate, they write a default value to this signal. Similarly, the TDF to TLM converters store the TDF tokens in a buffer to be able to use them at the TLM read transaction instants. To solve the second problem, the converters include a payload event queue between the transaction input and the converter buffer. Transactions are read from this queue in time order. These converters act as a TLM interface for TDF and have been shown to be useful for specifying TLM/TDF virtual platforms. However, this work deals with interconnecting sampling system models to TLM models, but not CT models, which are our focus. Consequently, specific CT/DE interactions such as event generation and discrete event change to the CT models and state are neither discussed nor used for the converter module specification.

3.5.4.3 SystemC MDVP

It is an extension that follows the SystemC AMS standard [23]. It allows the detection of causality issues between DE and TDF MoCs [77, 87] and simplifies the hierarchical composition of different MoCs using a master-slave simulator structure [88]. This structure introduces the possibility of directly synchronizing the DE SystemC kernel (master) with CT MoCs (slaves). However, the supporting case studies illustrate only indirect synchronization through an intermediary TDF MoC.

3.5.4.4 SystemC/Matlab&Simulink Co-simulation

Bouchima et al. [89] explore SystemC (DE) and Matlab & Simulink (CT) co-simulation. We refer to this work because it explores three synchronization modes based on VHDL-AMS's canonical algorithm. The first mode, *Full Synchronization*, implements the canonical algorithm without modifications. The second mode, *Predictable Events*, assumes that all direct interactions take place at known sampling times, it modifies the canonical algorithm so that the CT simulator can advance time to the next sampling time rather than to the next discrete event time—which results in DT synchronization. The third mode, *Events-Driven*, allows the DE simulator to execute optimistically beyond the CT simulator and to synchronize only when a discrete event implies a DE to CT interaction. CT state events imply DE simulation rollbacks. This operation is expensive because the size of the DE state for models of software and hardware systems may be big. SystemC does not support rollback. However, if a CT model does not generate state events, then no DE rollback is necessary.

3.5.5 Partial Conclusions

SystemC is the reference tool for high-level DE simulation. Many extensions serve CT/DE modeling and simulation. However, they do not completely support direct interactions the way tools such as VHDL-AMS and Ptolemy II do. Given SystemC's modeling and simulation advantages, we can develop an extension for high-level CT/DE simulation. The extension would support direct interactions and implement direct synchronization for fast and accurate simulation. In this sense, we can also explore parallel simulation to achieve even higher simulation speed, for which we describe the state of the art in Section 3.6.

3.6 Parallel Simulation

Time order between events makes DE simulation sequential by definition, but current multi-core and networked processors could be used to execute different DE model parts on different processors. One major challenge is to ensure causality even if the events are processed possibly out of order. The work in [44] exposes several parallel discrete-event simulation approaches. They generally belong to two categories: conservative, which avoids causality errors at the cost of synchronization overhead, and optimistic, which allows causality errors to reduce overhead at the cost of having to detect and correct errors when they occur. Given the richness of research in this area, we limit our discussion to parallel simulation in SystemC.

The standard SystemC simulator demands sequential execution and poses several parallelization challenges [90]. These challenges are mostly related to race conditions on shared data that arise from SystemC processes that simultaneously:

- Access global variables or module attributes: the work in [91] highlights that the non-preemptive and sequential SystemC evaluation of processes guarantees mutual exclusive access to shared variables, but parallel simulation is a different story.
- Communicate over unprotected channels: the SystemC channel primitives do not implement any protection mechanism (e.g., mutexes and locks).
- Communicate by direct access to a memory location: the Direct Memory Interface (DMI) protocol in TLM models promotes this possibility.
- Access the simulator state: scheduling functions, for example, are also unprotected.

Any parallelization attempt must define race condition prevention policies and mechanisms [91]. The existing approaches differ not only on this point but also on what and how they parallelize.

3.6.1 Register-Transfer Level Approaches that Preserve Global Time

The SystemC standard does not impose any process evaluation order. Some solutions rely on this to parallelize the scheduler's evaluation phase. This is also known as parallelization at

the delta cycle [92]. Chopard et al. [93] modify the scheduler to maintain a pool of parallel threads and requires designers to manually partition the model by grouping modules into nodes. Each node groups together the SystemC processes for execution in one thread.

Ezudheen et al. [94] follow the same idea but explore three process-thread assignment policies: work-sharing, with a fixed number of processes per thread; work-stealing, with processes being stolen by idle threads from busy threads; and manual grouping, with manual assignment by the designer. They conclude that work-stealing and manual grouping policies perform best because they profit from increased resource usage and more data locality. Protection mechanisms prevent channel and simulator state race conditions.

Schumacher et al. [95] also evaluate processes in parallel but handle race conditions differently: first, the designer is held responsible for writing thread-safe code inside SystemC processes, and second, access to the simulator state is sequential. Dömer et al [91] also enforce manual protection of process shared data by sponsoring a preemptive execution semantics, as the one found in the SpecC ESL simulator.

Ainey et al. [96] explore parallel process evaluation by restricting modeling. Each component is a finite state machine that implements three types of functions: a *transition function* that maps inputs to new states, a *Moore function* that maps states to outputs, and a *Mealy function* that maps inputs and state to outputs. At each simulation delta cycle, the modified scheduler executes in parallel all functions of each type.

In general, the more modules in the model, the greater the speed-up, the more the computations per module, the greater the speed-up, and the more runnable processes at each delta cycle, the greater the speed-up. These results are valid for RTL models.

3.6.2 Register-Transfer Level Approaches that Distribute Time

The preceding approaches introduce an implicit synchronization barrier at the end of the evaluation phase. To achieve greater speed-up, Chen and Dömer [97] propose out-of-order process execution. To this aim, the compiler divides models into segments of sequential execution. The simulator assigns each segment to a particular thread. Each thread has an independent local simulation time. To prevent causality errors and race conditions, the simulator maintains a set of data structures with information on shared data and causality relations between segments. Before executing a segment, the simulator consults these structures to predict the absence of conflicts. This prediction is valid beyond one simulation cycle and is used to increase the number of segments ready to run at any given time. Cheng et al. [98] extend this approach by predicting not only conflicts but also event notifications. They require a SystemC-aware compiler and a modified simulator.

Roth et al. [99] present another approach to circumvent this barrier while preserving causality. They map SystemC processes to logical processes (execution threads) and communication channels to FIFO links that carry messages in time order. The resulting network is simulated by the Chandy-Misra-Bryant algorithm [44]: each logical process reads the earliest input message, sets its local time to the message timestamp plus a time delta, processes the

message if it is not null, and writes messages to its output links followed by a null message containing the new local time. As time is local, no global coordination is needed. They argue that their approach is faster than parallel global-time preserving approaches.

These works have been applied to RTL simulations.

3.6.3 General Transaction Level Modeling Approaches

TLM models may be more apt for time distribution given their approximately timed and loosely timed abstractions. Viaud et al. [100] make one of the first attempts in this direction. They implement the TLM with Time approach—an intermediate between cycle-accurate and regular TLM—to achieve cycle-accurate precision with TLM speed. They provide TLM initiator, target, combined initiator/target, and interconnect primitives, and map them to a logical process network model. Execution follows the conservative Chandy-Misra-Bryant approach. Similar to this work, Mello et al. [101] and Pessoa et al. [102] introduce the TLM with Distributed Time paradigm similar and the parallel simulator SystemC-SMP. This simulator divides processes into groups of sequential execution that the designer can manually assign to particular threads. These works target shared memory multi-core models and do not support the DMI protocol.

Schumacher et al. [103] tackles the DMI problem and intends to support legacy TLM models. Designers are required to partition the simulation state into "*zones*" and to group the SystemC processes that operate on the same zone. Processes in each group execute sequentially in the same thread. If a process issues a DMI transaction that modifies the state of another zone, the simulator migrates it to the corresponding group to prevent race conditions. This solution parallelizes only at the delta cycle level.

Weinstock et al. [104, 105] present the parallel SystemC simulator SCope that provides causality error prevention in distributed time TLM simulations. Different threads simulate different parts of the model with local times that differ by a time quantum. During simulation, all transactions are manually or automatically delayed so that their targets always receive them in their local time future, thus preventing causality errors. They automatically handle channel race conditions and let the designer manually protect process shared variables. SCope only supports the TLM blocking transport interface.

Weinstock et al. [106] also introduce SystemC-Link, a solution that supports non-blocking transport. It requires designers to partition SystemC models into interconnected segments. The simulator assigns each segment to a thread running the standard SystemC simulator with an independent local time. Channels connecting segments have a latency that is used to determine how long threads on the receiving end can go ahead without risking skipping messages. This approach differs from Chandy-Misra-Bryant simulation in that a controller centralizes synchronization. At each simulation step, the controller computes a limit time for each segment based on the local time of its peers and on the input channel latencies. If a segment has not yet reached its limit time, it is considered to be ready to simulate, otherwise, it waits for its peers to further advance time. Execution is non-preemptive. Segments yield

control following two policies: as-soon-as-possible, when they reach the next communication point; and as-late-as-possible, when they reach their limit time. The former policy is more accurate, the latter is faster. Sequential execution within segments and in DMI transactions prevents race conditions.

Although these approaches speed-up simulation, in industrial models, speed-up gains have been observed to be limited by the number of processes that execute at each time [92].

3.6.4 Transaction Level Modeling Approaches that Increase the Number of Runnable Processes

To further speed up simulations, some works implement strategies to increase the number of TLM runnable processes. Moy [107] applies the notion of "tasks with duration", first exposed in [108]: instead of assuming that computational tasks (SystemC processes) execute instantaneously, designers can specify a non-zero duration by using the new primitive `sc_during`. The scheduler identifies the tasks whose duration overlaps and executes them in parallel.

Based on the work in [101, 102], Peeters et al. [109] proposes to periodically synchronize TLM processes. They partition processes into clusters that execute in different threads. Each thread executes a standard SystemC simulator with its own local time. Processes may communicate via regular TLM transactions. To avoid DMI race conditions, writing is exclusive and is always done after all reads are finished. Threads synchronize their local times periodically, which helps to prevent causality issues. The same authors propose in [110] a parallel SystemC scheduler and a specialized hardware accelerator for this kernel.

Ventroux and Sassolas [111] present the SCale simulator that sets the same global quantum to all TLM processes to increment their number at each delta cycle. A dependency graph maintains information on read and write memory accesses from each thread by using designer manual annotations. To detect race conditions, the simulator checks the graph for interleaved access. Additionally, simulation repeatability is ensured by using the dependency graphs to store the process evaluation order that can be repeated in subsequent simulations.

Based on this work, Busnot et al. [112, 113] introduce SCale 2.0. The major change is that it automatically annotates memory accesses. To this aim, it associates to each address a finite state machine with four states: no access, owned, read-exclusive by one process, and read shared by two or more processes. The state is used to prevent race conditions by sequentially executing processes that try to read from or write to an address already owned by another process. To avoid unnecessary sequential execution, they apply a reset policy so that the finite states are valid over some simulation cycles and not over the whole simulation. However, conflicts can still occur. They are detected via graph dependency analysis. The simulation can recover via a rollback.

These approaches report significant speed-ups.

3.6.5 Other Transaction Level Modeling Approaches

The preceding solutions generally parallelize on one host with multiple cores. Sauer et al. [114] propose the Concurrent Model Interface framework that distributes the TLM simulation into cores from different networked hosts. The designer partitions the model and uses the framework instances to handle each part. Each instance networks, synchronizes, and communicates with other instances. Networking is based on direct TCP/IP connections. Synchronization follows temporal decoupling, with each instance advancing local time up to a quantum. Communication mechanisms allow processes running in different instances to exchange messages via any SystemC communication primitives except for DMI between instances in different hosts. The authors state that their solution is used in virtual prototypes of complex network processing systems with hundreds of processor models. Other works intend to use graphical processing units to parallelize the computation within individual processes, but they have limited applicability because little data parallelism exists at the TLM level [111].

3.6.6 Continuous-Time Approaches

The preceding works are not extended to SystemC AMS for CT systems. Two works mention parallelization in this context. First, for model verification, Vörtler et al. [115] make several instances of the same SystemC AMS model run in parallel with different parameter and test inputs to speed up test coverage; the execution of each model continues to be sequential. Second, in ForSyDe SystemC [81], the authors argue that their models are apt for parallel execution thanks to their Kahn Process Network based modeling; however, they do not give any detail on the parallel simulation algorithms.

As we state in Section 2.6.2, the simulation of CT systems can be parallelized in three levels: parallelism across the method, parallelism across the steps, and parallelism across the system. Numerical analysis studies the first two levels and the resulting parallel numerical methods, in general, can be integrated into larger simulation frameworks. We consider that the study of these algorithms is beyond our purposes for this thesis. However, parallelism across the system implies the manual or automatic partition of one system into multiple CT constituents and it might be useful to speed up the simulation of complex cyber-physical system models. Research in this direction in the context of high-level cyber-physical system modeling and simulation might be worthy.

3.6.7 Partial Conclusions

Research on parallel DE simulation in SystemC is rich. The major challenges are to preserve causality, to prevent race conditions in various parts of the simulator and model execution, and to ensure efficient parallel resource usage. Some approaches preserve causality by parallelizing at the delta cycles while others prefer to profit from TLM time decoupling at the cost of having to detect and correct causality errors. Some approaches prevent race conditions by restricting modeling while others analyze dependencies during compilation to sequentially execute the code prone to race conditions. Some approaches increment parallel resource usage

by relaxing time constraints while others introduce modeling constructs with implicit time flexibility. We are interested in adapting some of these ideas to combined CT/DE simulation.

3.7 Conclusions

In Section 2.7, we posed three questions related to the CT/DE synchronization, its simulation support for CT models from different physical disciplines, and the parallel simulation of CT/DE models. The state of the art answers them in different ways. First, languages and tools such as VHDL-AMS, Verilog AMS, Ptolemy II, etc., implement *direct CT/DE synchronization* as a feasible alternative to fixed-step approaches. Direct synchronization is based on direct interactions that neither omit nor delay important features in the original CT and DE signals. But VHDL-AMS and Verilog AMS are typically used for modeling at the Register-Transfer Level of abstraction, and Ptolemy II is used mostly for academic research. Their synchronization algorithms are limited by the time of the next event, which can cause overhead. We need an efficient and high-level alternative received by the design community.

SystemC is the reference electronic system-level design language. It provides support for virtual prototyping. For this reason, multiple extensions target CT/DE modeling and simulation, of which SystemC AMS is the most widely used. SystemC AMS has many advantages: portability, rapid execution of TDF models via a pre-simulation schedule, etc. But it implements only fixed-step synchronization. A direct approach on top of SystemC may be useful to reach high simulation accuracy and speed in the virtual prototyping of CT/DE models and other high-level system design use cases. Such an approach should:

- Provide modeling support to all direct interactions.
- Implement a superdense time model that ensures the simultaneity, ordering, and causality of events.
- Be optimistic to avoid the overhead of lock-step approaches and free the integration steps of size constraints.
- Preserve causality by a roll-back mechanism that does not add significant computational costs in time or space.
- Be portable and general, i.e., avoid modifications to the DE simulator.

Second, SystemC AMS and Ptolemy II implement the concept of model of computation, which is helpful for the *simulation of CT models from different physical disciplines*. For example, SystemC AMS's CT models of computation allow specifying general CT systems in the form of signal flow graphs and electrical systems in the form of electrical linear networks. But both of them synchronize with the DE time domain by fixed steps. The direct CT/DE synchronization approach should be generic enough so that different CT models of computation can be specified and simulated on top of it.

And third, research on DE *parallel simulation* is rich, but it has not yet been explored in the context of CT/DE high-level models. In the case of SystemC, parallel simulation faces several challenges such as the preservation of causality, the prevention of race conditions, and the correct use of parallel resources. These challenges may be present in parallel CT/DE simulation and some of the state-of-the-art solutions may still be applicable.

In the following chapters, we present our solution to these three questions.

Chapter 4: Direct Continuous-Time and Discrete-Event Synchronization

Contents

Pantoporos aporos : passant partout quand il n'y a pas de passages, à travers rien il va vers ce qui arrive
— Barbara Cassin, Le bonheur, sa dent douce à la mort

4.1 Introduction

In Chapter 3, we have described several simulation languages and tools that implement direct CT/DE synchronization as an alternative to fixed-step approaches. We have also seen that SystemC and its TLM extensions provide software and digital hardware modeling and simulation capabilities that make it the reference tool for Electronic System Level design. In addition, its AMS extensions provide support for modeling and simulation of CT systems via its Linear Signal Flow and Electrical Linear Networks models of computation. However, they synchronize with the SystemC DE simulator only by fixed steps. As direct CT/DE synchronization may help to attain high simulation accuracy and speed, we consider of interest, then, to design and implement a direct synchronization algorithm on top of SystemC.

We begin this chapter with the definition of a set of CT modeling requirements to support direct interactions in Section 4.2. Then, in Section 4.3, we propose a direct CT/DE synchronization algorithm that relies on the compliance of these requirements to synchronize both simulations; in this section, we also study the effect of the selection of the CT integration interval size on simulation speed and demonstrate some important algorithm properties such as causality (it does not generate events in the past), completeness (its behavior is determined for all possible cases of execution), and liveness (it does not stagnate the simulation). We test our algorithm with three different models in Section 4.4 to compare its accuracy and speed to fixed-step approaches. Finally, we present our conclusions in Section 4.5.

4.2 Continuous-Time Modeling Abstraction Suited for Direct Continuous-Time and Discrete-Event Synchronization

To define a new direct CT/DE synchronization strategy on top of SystemC, the CT models must follow the SystemC modeling approach, where modules communicate through DE channels via ports. These CT modules must use superdense time, appear to the DE SystemC simulator as regular DE modules, and support the direct interactions. For this reason, before defining the synchronization algorithm, we establish three essential aspects for continuous-time modeling: the time model implementation, the level of abstraction to internally represent the continuous behaviors, and the necessary interface to support direct CT/DE interactions.

4.2.1 Implementation of the Superdense Time Model

We need superdense time to correctly simulate the direct interactions. As discussed in Section 3.5.2, SystemC's time can be interpreted as a $\mathbb{N} \times \mathbb{N}$ superdense time model. We represent the first component—the model time—by the SystemC global simulation time, and we get it by invoking SystemC's `sc_core::sc_timestamp` function. We represent the second component—the microstep—implicitly by the simulation delta cycles, and we use SystemC's `sc_core::sc_delta_count` function [15] to get the total number of delta cycles, from which we keep track of the microstep at the current model time.

4.2.2 Level of Abstraction Selected for Continuous-Time Models

We have presented in Section 2.4.1 that the behavior of CT models can be either linear or nonlinear, that linear models are simpler, and that although there are techniques to linearize a nonlinear model, this is not always possible. For this reason, we intend to provide support to both linear and nonlinear models. In addition, in Section 2.4.2, we present different mathematical representations, such as transfer functions, state-space, differential algebraic equations, etc.; they give different levels of detail on the system structure. We also present different graphical representations, such as electrical circuit diagrams, block diagrams, etc.; they also give different levels of detail on the system structure and can be transformed into mathematical ones. Given its support to linear and nonlinear modeling, capacity to represent high-level details about the system structure in the form of state variables, possibility to be obtained from graphical representations, and extensive numerical simulation support, we consider the state-space representation as an appropriate level of abstraction for our purposes.

Equation (2.1) gives a general form of this representation, we first defined it in Section 2.4.3 and we repeat it here for convenience. Variables $\boldsymbol{x}$, $\boldsymbol{u}$, t_{CT}, and $\boldsymbol{x}_{\mathrm{o}}$, denote the CT state, CT inputs, CT time, and CT initial conditions, respectively. Functions f and g give the derivative and output values, respectively. They are more precisely defined in Section 2.4.3.

$$\begin{aligned} \dot{\boldsymbol{x}} &= f(\boldsymbol{x}, \boldsymbol{u}, t_{\mathrm{CT}}) \\ \boldsymbol{y} &= g(\boldsymbol{x}, \boldsymbol{u}, t_{\mathrm{CT}}) \\ \boldsymbol{x}(t_{\mathrm{CT_o}}) &= \boldsymbol{x}_{\mathrm{o}} \end{aligned} \tag{2.1}$$

In this and the following chapters, variables $\boldsymbol{x}$, $\boldsymbol{u}$, t_{CT} and $\boldsymbol{x}_{\mathrm{o}}$, refer to the CT state, CT inputs, CT time, and CT initial conditions of a particular CT module, respectively.

4.2.3 Interface for Describing Continuous-Time and Discrete-Event Interactions

The state-space representation accounts for the internal CT module behavior, but for combined CT/DE simulation to be possible, these modules must interact by events and appear to the DE simulator as regular DE modules. To this aim, they must generate and consume DE signals (Section 3.2). In this and the following chapters, variables $\boldsymbol{i}$ and $\boldsymbol{o}$ denote the values of the signals connected to the DE input and output ports of a CT module, respectively.

To ensure modeling support to the direct CT/DE interactions, we describe a set of requirements with which the CT modules must comply:

1. **Support to instantaneous DE changes in the CT model**: the first requirement is to define a method that can describe the CT state behavior as given by the derivative function (f) in Equation (2.1) and its dependence on the DE input events. This method must take in the CT state, the DE inputs, and the CT simulation time, calculate the derivative values that represent the CT state behavior at the given time according to

the given DE inputs, and return this value. In this and the following chapters, we refer to this method as GET-DERIVATIVES.

Consider the model of the flying ball that we present in Figure 3.2 and describe in Section 3.4.1.4. GET-DERIVATIVES would calculate and return the ball's position and speed derivatives.

2. **Support the detection of state events and the instantaneous DE changes in the state conditions**: the second requirement is to define a method that can represent the state conditions and their dependence on the DE input events. As discussed in Section 3.2, state conditions allow detecting state events. This method must take in the CT state, the DE inputs, and the CT simulation time, evaluate the state conditions at the given time, and return a boolean indicating whether or not they are met. In this and the following chapters, we refer to this method as IS-EVENT. IS-EVENT must not report the same event over infinitely consecutive microsteps, otherwise, it can prevent the advance of the simulation time, as we discuss in Section 4.3.4.2.

   Consider again the model of the flying ball in Figure 3.2, IS-EVENT would monitor the the ball's vertical position (`z`) and return FALSE when it is greater than zero, and TRUE when it is equal to 0—indicating a collision state event.

   In addition, the CT module may define a method that can calculate and return a positive integer representing the time to the next time event (an event generated by the CT module at a time known in advance, as described in Section 3.2, item 1). This method is optional: if left undefined, it is assumed to return a default positive infinite value indicating that there is no such event. In this and the following chapters, we refer to this method as GET-TIME-TO-NEXT-TIME-EVENT.

3. **Support to instantaneous DE changes in the CT model and state**: the third requirement is to define a method that can represent the conditions and rules to instantaneously update the CT model and state as a reaction to an input event. This method must take in the state, the DE inputs, and the CT simulation time, evaluate the model and state update conditions, make the updates if necessary, and return a boolean indicating whether or not an update has been made. In this and the following chapters, we refer to this method as EXECUTE-UPDATES.

   For our example of the flying ball model in Figure 3.2, EXECUTE-UPDATES would set the ball's vertical speed to zero (`vz`$\leftarrow 0$) at the collision event.

4. **Support to the generation of DE outputs**: finally, the fourth requirement is to define a method that can generate output events. This method must take in the state, the DE inputs, and the CT simulation time, compute the CT outputs ($\boldsymbol{y}$) as given by the output function (g) in Equation (2.1) and map them to events in the DE output ports ($\boldsymbol{o}$). Output events can be both, state and time events. In this and the following chapters, we refer to this method as GENERATE-OUTPUTS.

For our example of the flying ball model in Figure 3.2, GENERATE-OUTPUTS would toggle the value of a DE boolean output port to indicate the collision event.

These methods constitute our proposed CT modeling interface to provide support to the direct interactions. It can be defined manually by the designer during modeling or automatically by a CT model of computation during elaboration. The model of computation transforms an interconnected set of modeling primitives representing the CT system, such as electrical circuit elements, into the CT equations, conditions, and rules needed to define these methods. We provide examples of manual CT modeling in Section 4.4 and of the definition of CT models of computation in Chapter 5. In the current chapter, we assume that the CT module complies with our required interface, either manually or automatically, so that it can be simulated and synchronized by the algorithm that we propose in Section 4.3.

4.3 Direct Continuous-Time and Discrete-Event Synchronization Algorithm

We propose an algorithm for the direct synchronization of CT/DE simulation on top of SystemC. Our algorithm is optimistic, it advances the CT local time (t_{CTf}) in the future with respect to the DE global simulation time (t_{DE}). This helps prevent the synchronization overhead of conservative approaches and experiment on different optimistic strategies to achieve our simulation speed-up goal. As we want to be as non-intrusive as possible, we build our algorithm on top of the standard SystemC simulator.

To this end, we implement it as a SystemC process that is launched by the CT module at the SystemC `end_of_elaboration` callback. By invoking the methods in our CT modeling interface (Section 4.2.3), the process can react to input events, evolve the CT state, detect state events, and generate output events on behalf of the CT module. The CT modules are seen, then, as regular DE modules and their CT simulation and synchronization are transparent to the SystemC simulator. This process is executed for the first time during the SystemC initialization phase, and later at different simulation times during the SystemC evaluation phase.

4.3.1 Synchronization Algorithm Phases

Figure 4.1 gives an overview of the synchronization process. It is initially suspended, and executes in three phases:

1. **Initialization and reactivation handling**: reactivation can occur during SystemC's initialization phase, at the time of a reactivation scheduled by itself ($t_{\mathrm{DE}} = t_{\mathrm{CTf}}$), and at the notification of input events. During initialization and at a self-reactivation (black path), the process generates output events and creates checkpoints on the CT solutions. At the notification of input events (red path), it rolls the optimistic solutions back—they are invalidated by the input event—, re-calculates new solutions to catch-up its local time with t_{DE}, and creates a checkpoint on the new solutions.

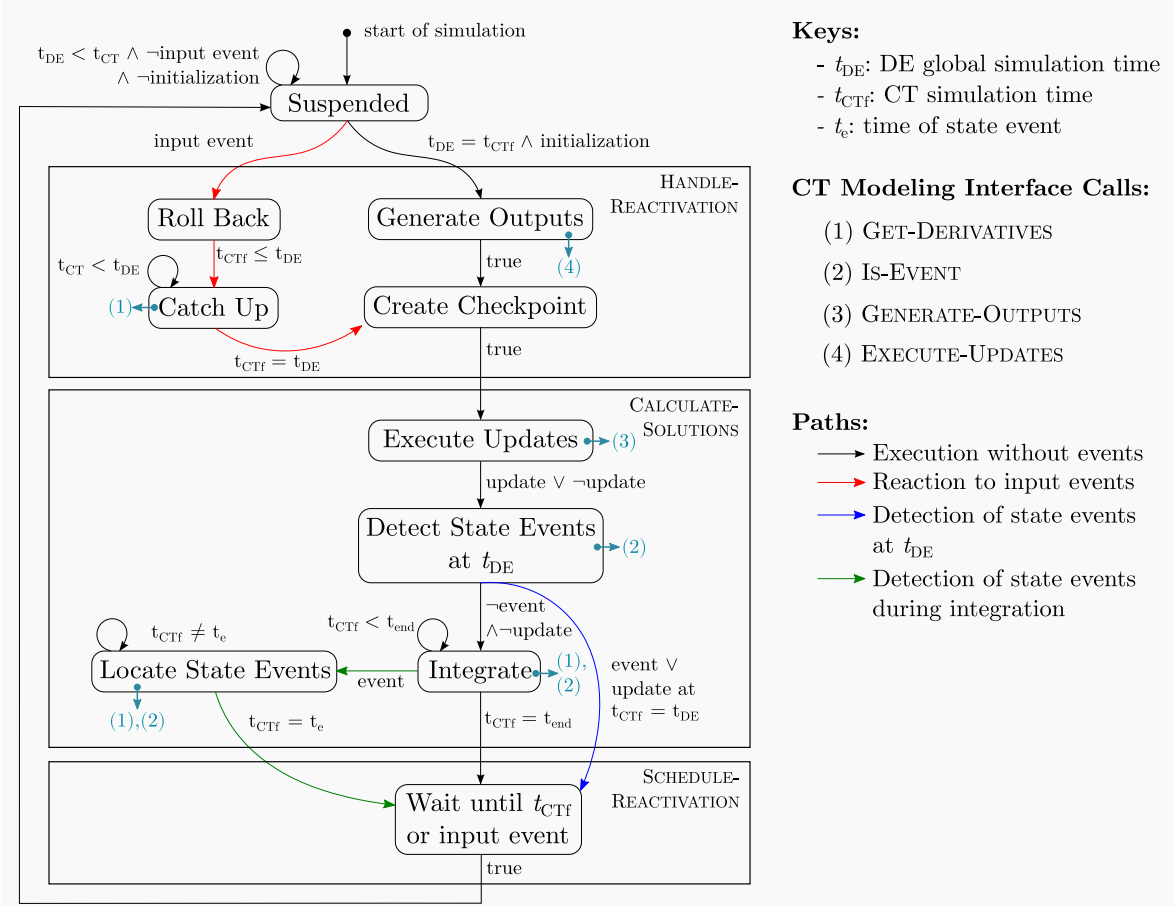


Figure 4.1: Direct synchronization process overview

2. **Calculation of solutions**: the process first checks for updates to the CT model or state as a reaction to input events at the current simulation time, then it tests for state events at this time, if there is an update or event, it skips integration of the CT equations and goes to the scheduling reactivation phase to output the event (blue path); otherwise, it integrates the equations to evolve the CT state optimistically over the interval I_v that goes from the current global time ($t_{CT_o} = t_{DE}$) to some time in the future ($t_{CT_f} > t_{DE}$, black path) or until it detects and locates a state event (t_e, green path).

3. **Reactivation scheduling**: it schedules its next reactivation at t_{CT_f}, which can be at the next delta cycle if the process detected a state event or state update at the current time, or in the future if it computed solutions over the integration interval; then it suspends.

Notice that the process calls our CT modeling interface functions at different points of its execution.

Algorithm 1 gives the top-level algorithm. Let us describe these phases in detail.

Algorithm 1 SYNCHRONIZATION-ALGORITHM

```
1: while true do
2:     HANDLE-REACTIVATION()
3:     CALCULATE-SOLUTIONS()
4:     SCHEDULE-REACTIVATION()
5: end while
```

4.3.1.1 Initialization and Reactivation Handling

This phase handles the process initialization and reactivation, creates checkpoints of solutions, and generates output events based on the solutions computed during the previous execution. A SystemC process can activate during simulation initialization, at the time of a reactivation scheduled by itself, and at the notification of input events. Figure 4.2 provides a graphical representation of these different cases; we find the DE simulation timeline at the top and the CT simulation timeline at the bottom, circles represent events and squares represent CT solution points. HANDLE-REACTIVATION deals with these cases as follows:

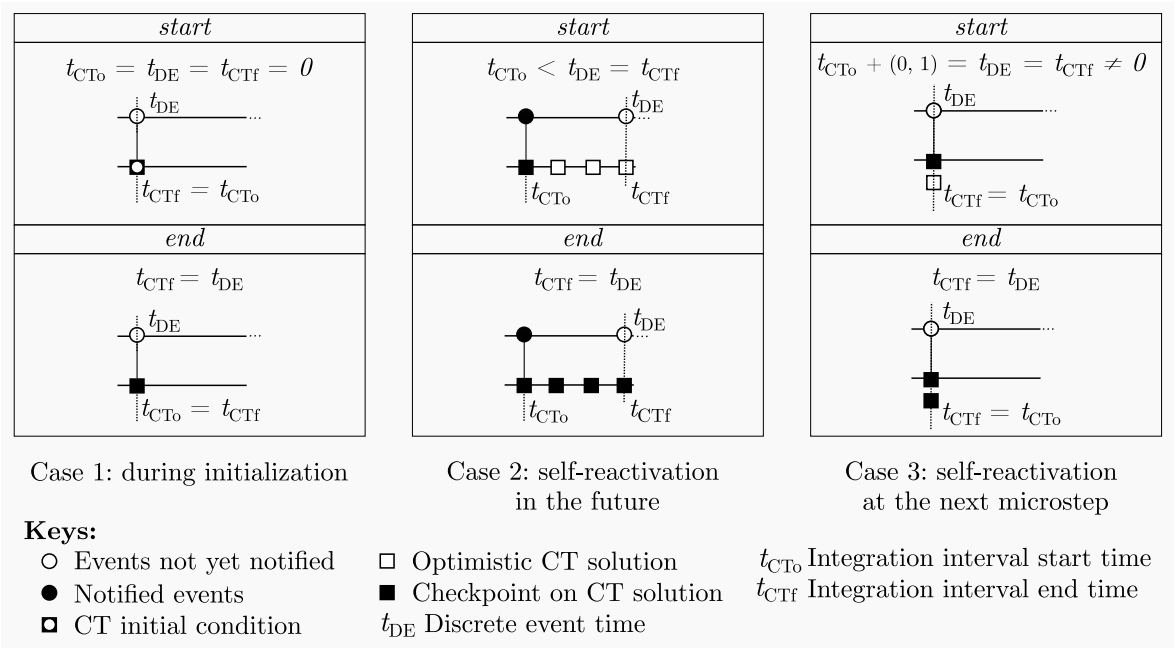


Figure 4.2: Graphical representation of the first three cases of HANDLE-REACTIVATION

1. **Activation during initialization**: let us assume that this is the first activation of the process ($t_{\mathrm{CT_o}} = t_{\mathrm{DE}} = t_{\mathrm{CT_f}} = (0, 0)$). It has not computed any solution yet. It sets the state from the CT initial conditions, generates outputs, and creates a checkpoint on the CT state and the DE inputs at this time.

2. **Self-reactivation in the future**: let us assume that, during the previous execution, the process computed optimistic solutions over an interval I_{v}[1]. It reactivates and the

[1] I.e., assume that $t_{\mathrm{DE}} = t_{\mathrm{CT_o}}$, that the process computed solutions over the non-zero interval $I_{\mathrm{v}} = (t_{\mathrm{CT_o}}, t_{\mathrm{CT_f}}]$ with $t_{\mathrm{CT_f}} > t_{\mathrm{CT_o}}$, and that it scheduled a reactivation at $t_{\mathrm{CT_f}}$.

DE simulator has advanced time such that $t_{\mathrm{DE}} = t_{\mathrm{CT_f}}$. The process did not receive any input events over I_{v}. Solutions are valid from start to end. The process generates outputs and creates a checkpoint on the CT state and DE inputs at this time.

3. **Self-reactivation at the next microstep**: let us assume that during the previous execution $t_{\mathrm{DE}} = t_{\mathrm{CT_o}}$, that an input event produced a discontinuity in the state at this time, and that the process scheduled a reactivation at the next microstep (see Section 4.3.1.2). It reactivates, the solutions interval $I_{\mathrm{v}} = (t_{\mathrm{CT_o}}, t_{\mathrm{CT_f}}]$ has zero length ($t_{\mathrm{CT_f}} = t_{\mathrm{CT_o}} + (0, 1)$), and the DE simulator has advanced time to the next microstep ($t_{\mathrm{DE}} = t_{\mathrm{CT_f}}$). The process generates outputs, creates a checkpoint on the state based on the new discontinuous solution, and on the DE inputs at this time.

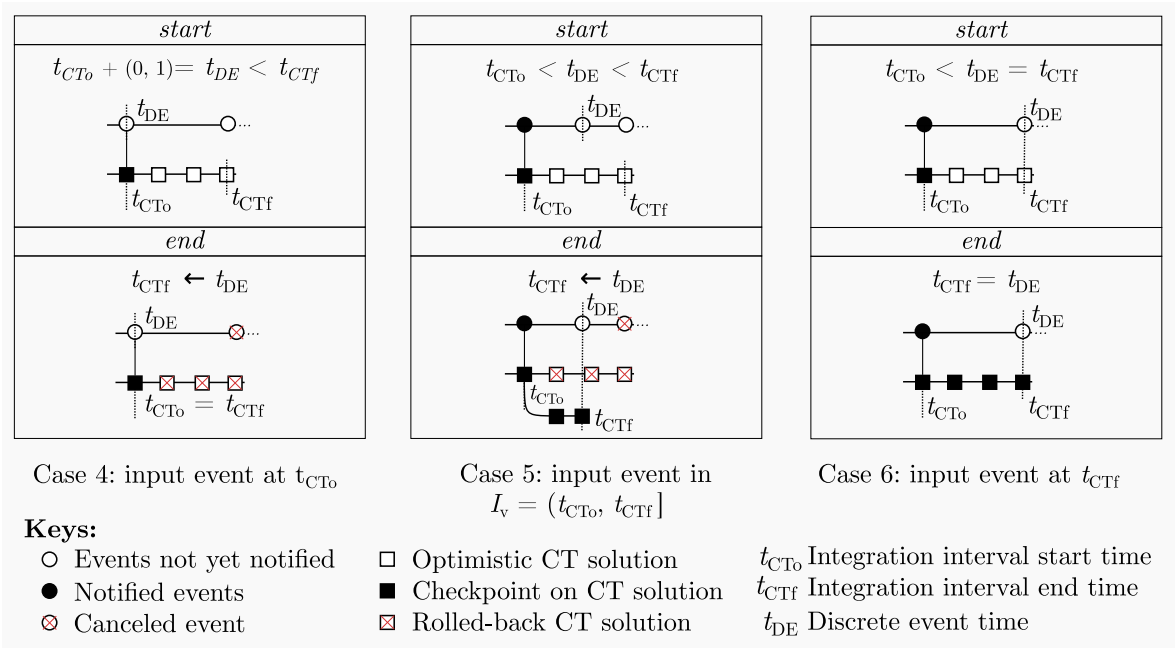


Figure 4.3: Graphical representation of the last three cases of HANDLE-REACTIVATION

Additional cases arise from CT simulation: input events can occur at the start ($t_{\mathrm{CT_o}}$), in the middle, or at the end ($t_{\mathrm{CT_f}}$) of the last interval of optimistic solutions I_{v}, possibly modifying the CT model and invalidating these solutions. HANDLE-REACTIVATION deals with these cases as follows (Figure 4.3):

4. **Reactivation by an input at the start of** I_{v}: let us assume that, during the previous execution, the process computed optimistic solutions over the interval I_{v} (Footnote 1). However, an input event activates it again and the DE simulator has advanced time only by one microstep ($t_{\mathrm{DE}} = t_{\mathrm{CT_o}} + (0, 1) < t_{\mathrm{CT_f}}$). The event invalidates the optimistic solutions. The process rolls back to the previous checkpoint, restores the interval end time, and skips outputs at this time because they have already been generated during the previous activation.

5. **Reactivation by an input in the middle of I_v**: let us assume that, during the previous execution, the process computed optimistic solutions over the interval I_v (Footnote 1). However, an input event activates it and the DE simulator has advanced time such that $t_{CT_o} < t_{DE} < t_{CT_f}$. The event invalidates the optimistic solutions from t_{DE} forward. The process rolls back to the previous checkpoint at t_{CT_o} and invokes the CATCH-UP function that sets the interval end time to the current time and recomputes solutions from t_{CT_o} to t_{DE} based on the previous inputs. Then, the process generates outputs and creates a new checkpoint on the CT state based on the catched-up solutions and the DE inputs at t_{DE}. CATCH-UP prevents the process from being left behind t_{DE}, which would violate causality. The new inputs will be used to compute solutions from t_{DE} forward.

6. **Reactivation by an input at the end of I_v**: let us assume that, during the previous execution, the process computed optimistic solutions over the interval I_v (Footnote 1). Reactivation occurs at $t_{DE} = t_{CT_f}$. The new inputs only affect the integration interval that begins. Solutions are valid from start to end. The process handles the situation exactly as in case 2: it generates outputs and creates checkpoints at $t_{CT_f} = t_{DE}$.

Algorithm 2 HANDLE-REACTIVATION

```
1:  activated_by_input ← i ≠ i_cp                          // Flag used in Algorithm 7
2:  if t_DE = t_CTo + (0,1) < t_CTf then                   // Case 4
3:      x ← x_cp                                           // Restore state
4:      t_CTf ← t_CTo + (0,1) = t_DE                       // Restore time
5:  else                                                   // Cases 1, 2, 3, 5 and 6
6:      if t_CTo < t_DE < t_CTf then                       // Case 5
7:          x ← x_cp                                       // Restore state
8:          CATCH-UP()
9:      end if
10:     if t_DE = (0,0) then                               // Case 1
11:         x ← x_o
12:     end if
13:     GENERATE-OUTPUTS(x, i_cp, t_CTo)                   // Cases 1, 2, 3, 5 and 6:
14:     x_cp ← x                                           // Create state checkpoint
15:     i_cp ← i                                           // Create input checkpoint
16: end if
```

Algorithm 3 CATCH-UP

```
1:  t_CTf ← t_DE                                           // Restore time
2:  INTEGRATE(t_CTo, t_CTf, x, i_cp)                       // Evolve the state until current time
```

Algorithm 2 and Algorithm 3 summarize the previous discussion. Notice that we pass $\boldsymbol{i}_{cp}$ to GENERATE-OUTPUTS because outputs depend on the solutions and DE inputs from the previous execution and $\boldsymbol{i}$ may have changed since. Notice also that at the end of Algorithm 2, $t_{CT_f} = t_{DE}$ for all cases.

4.3.1.2 Calculation of Solutions

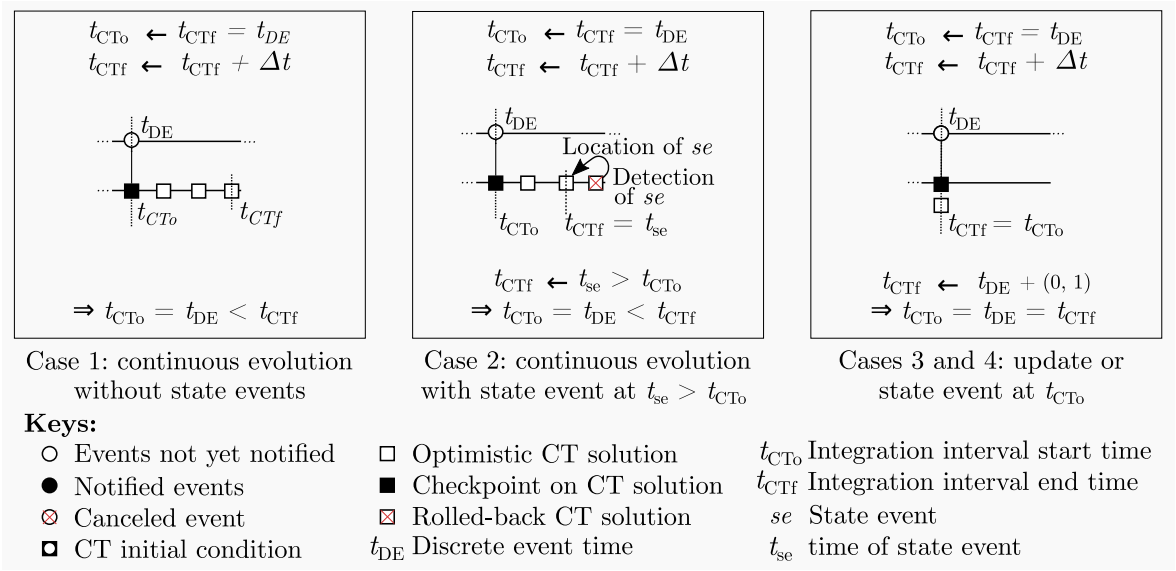


Figure 4.4: Graphical representation of CALCULATE-SOLUTIONS

The process has just executed HANDLE-REACTIVATION and, as a result, it has checkpointed CT solutions at the present DE time ($t_{\mathrm{CT_f}} = t_{\mathrm{DE}}$). CALCULATE-SOLUTIONS evolves the CT state continuously over the non-zero length interval $I_{\mathrm{v}} = (t_{\mathrm{CT_o}} = t_{\mathrm{DE}}, t_{\mathrm{CT_f}}]$ or discontinuously over one microstep $I_{\mathrm{v}} = (t_{\mathrm{CT_o}} = t_{\mathrm{DE}}, t_{\mathrm{CT_f}} = t_{\mathrm{DE}} + (0, 1)]$. During this phase, it detects state events triggered by the evolution of the state and by input events. The following discussion is represented in Figure 4.4. Assume that the process selects the interval end time optimistically in the future (some $t_{\mathrm{CT_f}}$ such that $t_{\mathrm{CT_f}} > t_{\mathrm{CT_o}} = t_{\mathrm{DE}}$); we give more precisions on different strategies to select $t_{\mathrm{CT_f}}$ in Section 4.3.3.

1. **Continuous evolution without state events**: the process invokes an integration function INTEGRATE that implements a numerical integration algorithm (e.g., Runge Kutta Dormand Prince) to compute solutions over the interval $I_{\mathrm{v}} = (t_{\mathrm{CT_o}}, t_{\mathrm{CT_f}}]$ by dividing it into integration steps of possibly different lengths (Section 2.4.4) and by invoking GET-DERIVATIVES to compute each solution point. Notice that GET-DERIVATIVES has access to the DE inputs, any change in the equations produced by these inputs is considered during integration. After each integration step, INTEGRATE tests the state conditions by a call to IS-EVENT, which, in this case, returns FALSE at each time to indicate the absence of state events.

2. **Continuous evolution with state events**: the process invokes INTEGRATE over the same interval and tests the state conditions as described in the preceding item. But in this case, IS-EVENT returns TRUE after some step in the middle of the interval. The process invokes LOCATE, which implements a root-finding method (e.g., bisection) to precisely locate the time of occurrence of the event (t_{se}) and the solution at this point

[2]. The integration interval ends at the time of the event.

3. **Discontinuous evolution**: assume that there has been an input event at the end of the previous integration interval $t_{\mathrm{CT_f}}$ (case 6 of HANDLE-REACTIVATION). Before invoking INTEGRATE, the process tests if the new input events create a discontinuity in the CT state or change in the CT model at the current DE time by a call to EXECUTE-UPDATES. If it returns TRUE, indicating an update to the model or state, the process skips the call to INTEGRATE and sets the integration interval end time to the next microstep ($t_{\mathrm{CT_f}} = t_{\mathrm{DE}} + (0, 1)$) to generate outputs and to create a new checkpoint at the next microstep. There can be multiple successive discontinuous updates at any particular model time, they are reflected in the microstep.

4. **State events triggered by input events**: the instantaneous updates to the state from the previous case or the new DE inputs at the current time can instantaneously make the state conditions hold TRUE, producing a state event. The process tests IS-EVENT immediately after invoking EXECUTE-UPDATES, if it returns TRUE, the process sets the integration interval end time to the next microstep and skips integration.

Algorithm 4 CALCULATE-SOLUTIONS

```
1:  t_CTo ← t_CTf = t_DE                                   // Integration interval start time
2:  Δt ← GET-INTERVAL-SIZE()
3:  t_CTf ← t_CTf + Δt                                     // Tentative end time
4:  is_update ← EXECUTE-UPDATES(x, i, t_CTo)               // Case 3
5:  is_event_by_input ← IS-EVENT(x, i, t_CTo)              // Case 4
6:  if is_update or is_event_by_input then                 // Cases 3 and 4
7:      t_CTf ← t_DE + (0, 1)
8:  else                                                   // Cases 1 and 2
9:      (event_detected, t_min, t_max) ← INTEGRATE(t_CTo, t_CTf, x, i)
10:     if event_detected then                             // Case 2
11:         t_se ← LOCATE(t_min, t_max, x, i)
12:         t_CTf ← t_se                                   // Interval ends at event time
13:     end if
14: end if
```

Algorithm 4 summarizes this discussion. GET-INTERVAL-SIZE defined in Algorithm 6 and called from Algorithm 4, line 2 helps to compute the optimistic interval end time $t_{\mathrm{CT_f}}$: it returns a positive real value representing the interval's length; we discuss its implementation in section 4.3.3.

4.3.1.3 Reactivation Scheduling

Once CALCULATE-SOLUTIONS has advanced time to the future or the next microstep, this phase schedules the reactivation of the process at that time, as shown in Figure 4.5. It

[2]Textbook [42] discusses the theory on a set of numerical integration and root-finding methods. The *odeint Boost library* [116] provides some implementations in C++. We do not impose any constraint on these methods so that they can vary according to the application requirements.

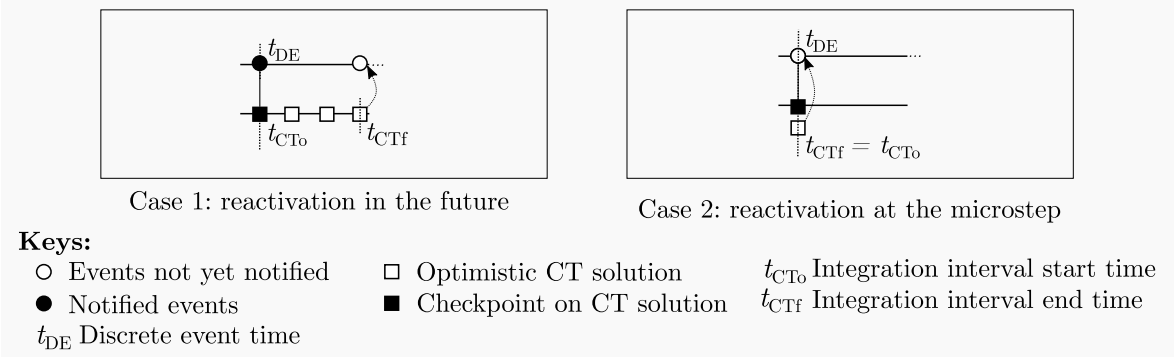


Figure 4.5: Graphical representation of SCHEDULE-REACTIVATION

Algorithm 5 SCHEDULE-REACTIVATION

1: WAIT(RE($t_{CT_f} - t_{DE}$) **or** *input_events_list*)

uses the SystemC `wait` function, which suspends a process until a given time has elapsed or one or more events of a list occur (Algorithm 5). In our case, the elapsed time is the difference between t_{CT_f} and the global simulation time t_{DE}; when RE($t_{CT_f} - t_{DE}$) > 0, it schedules a reactivation in the future (case 1), and when RE($t_{CT_f} - t_{DE}$) $= 0$, it schedules a reactivation at the next microstep (case 2). Function RE returns the model time (first component) of a superdense time instance. The list of events (*input_events_list*) can be created before launching the process by traversing the model's input ports while invoking the `value_changed_event` method of their signal interfaces to get a reference to the event [15].

4.3.2 The Big Picture

The SYNCHRONIZATION-PROCESS executes HANDLE-REACTIVATION, CALCULATE-SOLUTIONS and SCHEDULE-REACTIVATION in a loop. The DE simulator activates this process during the initialization and evaluation phases. The process computes solutions and schedules the next reactivation either in the future or at the next microstep (SCHEDULE-REACTIVATION). A reactivation in the future leads to cases 2, 4, 5 and 6 of HANDLE-REACTIVATION. A reactivation at the next microstep leads to case 3. In all cases, HANDLE-REACTIVATION ensures that t_{CT_f} equals t_{DE} just before invoking CALCULATE-SOLUTIONS. CALCULATE-SOLUTIONS advances t_{CT_f} further in the future or to the next microstep, which determines the two cases of SCHEDULE-REACTIVATION. The loop continues this way until DE simulation ends.

This algorithm handles all direct interactions. It detects state events during integration, locates their occurrence time (CALCULATE-SOLUTIONS, cases 2 and 4), and schedules a reactivation at that time to generate the corresponding output. It also handles the input events by rolling back the CT state when needed (HANDLE-REACTIVATION, cases 4 and 5) and by executing their instantaneous effect on the model (GET-DERIVATIVES called from INTEGRATE), on the model and state (EXECUTE-UPDATES called from CALCULATE-SOLUTIONS), and on the state conditions (IS-EVENT called from CALCULATE-SOLUTIONS).

In Section 4.3.3 we will see that this algorithm also handles time events from the CT model during the call to GET-INTERVAL-SIZE in Algorithm 4, line 2. This call determines the integration interval end time $t_{\mathrm{CT_f}}$ and controls how often CT/DE synchronization takes place, affecting simulation speed. We devote Section 4.3.3 to its study.

4.3.3 Automatic Selection of an Integration Interval Size

As we presented in Section 4.3.1.2, the synchronization process computes CT solutions over an interval and yields control to the DE simulator. The selection of the integration interval size can affect the simulation speed (Section 3.4.1.4). With small intervals, the process synchronizes frequently and has greater overhead. Similarly, small intervals impose more constraints on the integration step size, which can also severe the simulation speed. The Δt variable in Algorithm 4, line 2, denotes the interval size. In this section, we expose different strategies to select Δt. To this aim, we show in Figure 4.6 the effect of different fixed Δt—horizontal axis—on the wall-clock simulation time—vertical axis—for different case studies whose descriptions we delay to Sections 4.4.1, 4.4.2 and 4.4.3, as the our discussion in this section depends only on the results shown in the figure. The horizontal axis, Δt, is in log-scale to cover a wide range of values. For each case study, the figure shows the *average time between input events* to the CT module (μ_{e}), their *standard deviation* (σ), and the optimal wall-clock simulation time achieved by using a fixed Δt. We can observe that:

- A small Δt slows down simulation. Small intervals increase the number of synchronization points and their accumulated cost, and limit the maximum integration step size of the adaptive numerical integration methods.

- A large Δt also slows down simulation. Large intervals introduce less synchronization points and profit from a bigger integration step size, but more solutions are removed at rollbacks and their computational cost still accumulates.

- A smaller than optimum Δt can severe simulation speed more rapidly than a larger than optimum Δt. The cost of computing more CT solutions that are later rolled back when using large intervals is less than the cost of introducing several synchronization points and limiting the integration step size when using small intervals.

- Simulation efficiency approaches an optimum when Δt is around the average time between input events μ_{e} (i.e., $\Delta t \approx \mu_{\mathrm{e}}$) for models where input events are approximately uniformly distributed (small standard deviation σ). Two examples of such models are (1) and (2) in Figure 4.6, for which σ is less than $0.3 \cdot \mu_{\mathrm{e}}$. For these models, $\Delta t = \mu_{\mathrm{e}} + \sigma \approx \mu_{\mathrm{e}} + 0.3 \cdot \mu_{\mathrm{e}} = 1.3 \cdot \mu_{\mathrm{e}}$ is close to the optimum. More generally, Equation (4.1) with $K \approx 1$ makes simulation efficiency approach to an optimum.

$$\Delta t = K \cdot \mu_{\mathrm{e}} \tag{4.1}$$

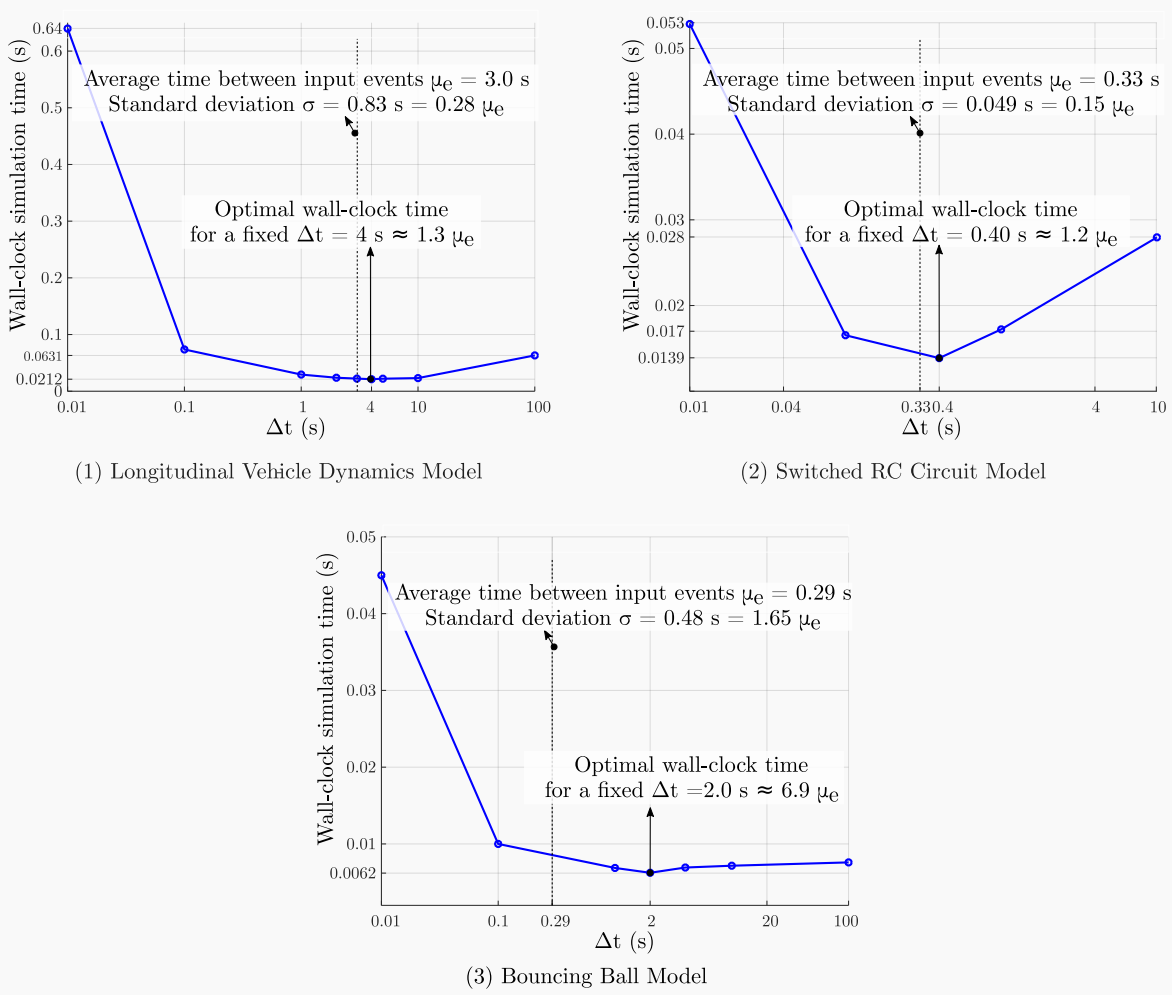


Figure 4.6: Wall-clock simulation time vs. Δt for different case studies (log horizontal scale)

- The size of a fixed Δt does not affect the number of CT simulation rollbacks. As events occur instantly and $\Delta t > 0$, each input event is contained by exactly one integration interval, which will be rolled back, no matter its size.

4.3.3.1 Proposed Strategies

The preceding discussion leads us to describe three strategies for selecting Δt:

1. **Fixed Interval Size as a Consequence of an Uniform Input Event Distribution**: interval sizes around the average time between input events, i.e., $\Delta t = K \cdot \mu_e$, provide a near to optimum wall-clock time. But the input events have to be approximately uniformly distributed. If there are simulation regions where they are sparse and others where they are frequent, Δt will get an intermediate value too small for the former regions, increasing overhead, and too big for the latter, increasing the rollback costs. Figure 4.6 (3) shows an example for which $\sigma = 1.65 \cdot \mu_e > 0.30 \cdot \mu_e$. In order to deal with variations on μ_e, we need an adaptive Δt.

2. **Adaptive Interval Size as a Consequence of Non-Uniform Input Event Distribution**: if the input events are not uniformly distributed or if the designer does not have *a priori* knowledge on μ_e, making it difficult to fix Δt, an alternative is to use an algorithm to estimate μ_e and adapt Δt accordingly as the simulation advances. Such an algorithm should have the following properties:

   (a) **Convergence**: Δt converges to a sequence of values that depends on the estimations of μ_e no matter the initial guess of Δt.

   (b) **Adaptability**: Δt increases in the regions where input events are sparse to cut the cost of unnecessary synchronization, and decreases in the regions where they are frequent to cut the cost of computing solutions over long intervals that are later rolled back.

   It is reasonable to use local μ_e estimates rather than just one fixed μ_e; in this way, $\Delta t = K \cdot \mu_e$ is always a good local choice even if the input events are not uniformly distributed over the whole simulation.

   The problem of calculating an adaptive Δt becomes, then, a problem of estimating μ_e to faithfully reflect the average time between input events as the simulation advances. To this end, let t_{le} denote the time of the last input event and $t_{sle} = t_{DE} - t_{le}$ be the elapsed time since the last input event, we propose Equation (4.2) as an estimator for μ_e and Equation (4.1) to calculate Δt. The value is updated at each reactivation of the synchronization process. This estimator is designed to consider both, the historical value of the time between input events (μ_e, the first term in the right-hand side of Equation (4.2)), and the last estimate of the time between input events (t_{sle}, the second term). This estimator can be implemented easily and does not add significant computational costs in time or space.

$$\mu_e \leftarrow \frac{\mu_e + t_{sle}}{2} \tag{4.2}$$

   Updating Δt according to our proposed adaptive strategy given by Equation (4.2) and Equation (4.1) meets the convergence and adaptability properties:

   - Averaging ensures convergence—property (a). When there are no input events at reactivation, t_{sle} gives the time that, up until the current simulated time, is known to be the maximum to separate the last input event and the next one. When there are events at reactivation, t_{sle} gives the exact time between the last two events. By a reasoning similar to the law of large numbers in probability theory, continually averaging the last estimate of the time between input events makes μ_e converge to the expected value of the time between events.
   - The μ_e estimator of Equation (4.2) allows $\Delta t = K \cdot \mu_e$ to adapt to local variations in the density of input events—property (b). When input events are sparse,

$t_{\text{sle}} = t_{\text{DE}} - t_{\text{le}}$ becomes large because t_{DE} progressively increases while t_{le} remains unchanged; a large t_{sle} makes μ_{e} also large, and consequently, Δt increases when input events are sparse. When input events are frequent, $t_{\text{sle}} = t_{\text{DE}} - t_{\text{le}}$ becomes small because t_{DE} increments only sufficiently to notify each one of the events; a small t_{sle} makes μ_{e} also small, and consequently, Δt decreases when input events are frequent. In models where input events are approximately uniformly distributed, $t_{\text{sle}} = t_{\text{DE}} - t_{\text{le}} \approx C$, for a real constant $C > 0$, μ_{e} converges to this value and Δt also adapts to an approximately constant value.

In addition to this theoretical analysis, we provide experimental evidence that the adaptive algorithm presents these properties in Section 4.4.5.

3. **Interval Size Given by the Next Input Event Time**: if we knew the time to the next input event, we could adapt Δt to avoid rolling back. Although the SystemC `time_to_pending_activity` function provides the time to the next event [15], this event is not always the next input event to the CT module; a high proportion of events unrelated to this module would unnecessarily reactivate it for small intervals. Moreover, as discussed in [41], this strategy avoids rollbacks only if there is just one CT module in the design: a second module could be evaluated immediately after the first one and generate output events that, if consumed by the first module, would cause it to roll back. For models where most of the events are related to the CT module, an additional gain in simulation speed can be attained by using Equation (4.3) to set Δt, where Δt_{ne} is the time to the next discrete event returned by the `time_to_pending_activity` function, and MIN is a function that returns the minimum of its arguments; this equation allows using the adaptive value unless there is an event nearer in time.

$$\Delta t \leftarrow \text{MIN}(K \cdot \mu_{\text{e}}, \Delta t_{\text{ne}}) \tag{4.3}$$

We can group these different strategies into one algorithm to select Δt.

4.3.3.2 Algorithm to Get the Integration Interval Size

GET-INTERVAL-SIZE (Algorithm 6) calculates and returns Δt based on the previous strategies. If parameter IS_ADAPTIVE evaluates to TRUE, the adaptive strategy is used; otherwise, Δt is set to FIXED_STEP_VALUE, a positive real number set by the designer. Boolean parameter USE_TIME_TO_NEXT_DE_EVENT indicates whether or not to use the time to the next discrete event. This function invokes the CT module method GET-TIME-TO-NEXT-TIME-EVENT (Section 4.2.3) to set the interval length to the time of the next time event if it is nearer. These three parameters allow the designer to select a strategy before starting the simulation.

As the adaptive strategy depends on μ_{e}, GET-INTERVAL-SIZE invokes ESTIMATE-TIME-BETWEEN-INPUTS to update its value according to Equation (4.2) (Algorithm 7, line

Algorithm 6 GET-INTERVAL-SIZE

```
1: ESTIMATE-TIME-BETWEEN-INPUTS()                                  // Update μe
2: if IS_ADAPTIVE and μe > 0 then
3:     Δt ← K · μe
4: else
5:     Δt ← FIXED_STEP_VALUE
6: end if
7: if USE_TIME_TO_NEXT_DE_EVENT then
8:     Δtne ← TIME-TO-PENDING-ACTIVITY()
9:     Δt ← min(Δt, Δtne)
10: end if
    // If there is a nearer CT time event, take its time as the final interval time
11: Δt ← min(Δt, GET-TIME-TO-NEXT-TIME-EVENT())
12: return Δt
```

6). The value of μ_e is not updated between microsteps to prevent it tending to zero ($t_{sle} = t_{DE} - t_{le} = (0, 0)$) (Algorithm 7, line 5). In addition, ESTIMATE-TIME-BETWEEN-INPUTS handles the special case when $t_{DE} > t_{le}$ and $\mu_e = 0$ (Algorithm 7, line 2), which occurs the second time the function is invoked; it sets μ_e directly to t_{DE}, which is a better estimate than the average result (Algorithm 7, line 6). Variables t_{DE}, t_{le}, and t_{sle}, should all be initialized to zero. The *activated_by_input* variable is set in line 1 of Algorithm 2 each time the synchronization process reactivates.

We discuss the speed-up provided by each strategy in Section 4.4. Before, we develop on other synchronization algorithm properties in Section 4.3.4.

Algorithm 7 ESTIMATE-TIME-BETWEEN-INPUTS

```
1: if tDE > tle then
2:     if μe = 0 then                                  // No estimate made yet
3:         μe ← tDE                                    // Use current time as an estimate
4:     else
5:         tsle ← tDE − tle                            // Update time since last event
6:         μe ← (μe + tsle) / 2
7:     end if
8:     if activated_by_input then
9:         tle ← tDE                                   // Set last input event time
10:    end if
11: end if
```

4.3.4 Synchronization Algorithm Properties

In this section, we discuss the causality, completeness, and liveness of our algorithm.

4.3.4.1 Discussion on Causality and Completeness

Lemma 1. *At the end of the execution of HANDLE-REACTIVATION (Algorithm 2),* $t_{CT_f} = t_{DE}$.

Proof. In cases 1, 2, 3, and 6 $t_{\mathrm{CT_f}} = t_{\mathrm{DE}}$ and it remains unchanged from start to end of Algorithm 2. In case 4, $t_{\mathrm{CT_f}} > t_{\mathrm{DE}}$ at the start, but it is assigned t_{DE} (Algorithm 2, line 4) and then remains unchanged until the end. In case 5, $t_{\mathrm{CT_f}} > t_{\mathrm{DE}}$ also, but it is set to t_{DE} in the call to the CATCH-UP function (Algorithm 3, line 1), after which it remains unchanged. At the end of all six cases, $t_{\mathrm{CT_f}} = t_{\mathrm{DE}}$. □

Lemma 2. *GET-INTERVAL-SIZE (Algorithm 6) always returns a positive value* $\Delta t > (0, 0)$.

Proof. Δt is assigned in three different places of Algorithm 6. If the adaptive strategy is used, condition $\mu_{\mathrm{e}} > 0$ (Algorithm 6, line 2) ensures that $\Delta t \leftarrow K \cdot \mu_{\mathrm{e}}$ is always positive (Algorithm 6, line 3); otherwise, Δt gets FIXED_STEP_VALUE (Algorithm 6, line 5), which by definition is a positive real number. If the time to the next event is used (Algorithm 6, line 7), Δt either retains the value assigned in the previous cases or gets Δt_{ne} (Algorithm 6, line 9), which is at least at the next microstep (0, 1)—`time_to_pending_activity` returns at least the constant SC_ZERO_TIME that is used by SystemC's WAIT function to schedule a delta notification or notification at the next microstep [15]. Finally, GET-INTERVAL-SIZE gets the minimum between the Δt calculated in the previous steps, which is positive, and the value returned by GET-TIME-TO-NEXT-TIME-EVENT (Algorithm 6, line 11), which is also positive (Section 4.2). Δt is not assigned anywhere else, thus the value returned by Algorithm 6 at line 12 is always positive. □

Lemma 3. *At the end of the execution of CALCULATE-SOLUTIONS (Algorithm 4),* $t_{\mathrm{CT_f}} \geq t_{\mathrm{DE}}$ *provided that* $t_{\mathrm{CT_f}} = t_{\mathrm{DE}}$ *at the start.*

Proof. First, in Algorithm 4, lines 1-3, $t_{\mathrm{CT_o}} \leftarrow t_{\mathrm{CT_f}} = t_{\mathrm{DE}}$ and $t_{\mathrm{CT_f}}$ is incremented by $\Delta t > 0$ (lemma 2). Second, if there is a discontinuous update or if a state event is triggered by an input event at $t_{\mathrm{CT_o}}$, then $t_{\mathrm{CT_f}} \leftarrow t_{\mathrm{DE}} + (0, 1)$ (Algorithm 4, line 7). Third, if a state event at $t_{\mathrm{se}} > t_{\mathrm{CT_o}} = t_{\mathrm{DE}}$ occurs, then $t_{\mathrm{CT_f}} \leftarrow t_{\mathrm{se}}$ (Algorithm 4, line 12). In all situations $t_{\mathrm{CT_f}} > t_{\mathrm{DE}}$. □

Theorem 1. *SYNCHRONIZATION-ALGORITHM (Algorithm 1) is causal, i.e., it does not generate events in the past w.r.t. the global simulation time* t_{DE}.

Proof. We need to show that (a) whenever it is executed $t_{\mathrm{CT_f}} \geq t_{\mathrm{DE}}$ [41] and (b) generated outputs correspond to solutions at t_{DE}. As CALCULATE-SOLUTIONS is executed immediately after HANDLE-REACTIVATION, and these are the only places where $t_{\mathrm{CT_f}}$ is modified, condition (a) follows from lemmas 1 and 3. Condition (b) is verified by inspection at the end of HANDLE-REACTIVATION: in case 1, outputs are based on the initial condition at $t_{\mathrm{DE}} = t_{\mathrm{CT_f}} = (0, 0)$. In cases 2, 3, and 6, outputs are based on the solutions at $t_{\mathrm{CT_f}} = t_{\mathrm{DE}}$. In case 4, no outputs are generated. In case 5, although a checkpoint is restored at $t_{\mathrm{CT_o}}$, solutions are caught up to t_{DE} and outputs are based on these solutions. Since (a) and (b) are true, Algorithm 1 is causal. □

Theorem 2. *HANDLE-REACTIVATION (Algorithm 2) is complete in the sense that it handles all possible cases of reactivation of SYNCHRONIZATION-ALGORITHM (Algorithm 1).*

Proof. SYNCHRONIZATION-ALGORITHM either activates for the first time at $t_{\mathrm{DE}} = 0$ or not. The first situation is covered by case 1. The second situation can correspond to a delta cycle or not. If it is a delta cycle, there are two possible cases: during the previous execution, it calculated solutions in the future, covered by case 4 ($t_{\mathrm{CT_o}} = t_{\mathrm{DE}} < t_{\mathrm{CT_f}}$), or at the next microstep, covered by case 3 ($t_{\mathrm{CT_o}} + (0,1) = t_{\mathrm{DE}} = t_{\mathrm{CT_f}}$). If it is not a delta cycle, the process is activated for the first time at some $t_{\mathrm{DE}} \neq (0,0)$, which implies that there exists an integration interval $I_{\mathrm{v}} = (t_{\mathrm{CT_o}}, t_{\mathrm{CT_f}}]$ with $t_{\mathrm{CT_f}} > t_{\mathrm{CT_o}}$ that bounds the reactivation at time t_{DE}. Reactivation at $t_{\mathrm{DE}} = t_{\mathrm{CT_o}} + (0,1)$ produces a delta cycle, already discussed. Reactivation at t_{DE} such that $t_{\mathrm{CT_o}} < t_{\mathrm{DE}} < t_{\mathrm{CT_f}}$ is covered by case 5. Reactivation at t_{DE} such that $t_{\mathrm{CT_o}} < t_{\mathrm{DE}} = t_{\mathrm{CT_f}}$ is covered by cases 2 and 6. Reactivation at $t_{\mathrm{DE}} < t_{\mathrm{CT_o}}$ is not possible as it would imply that t_{DE} goes backward ($I_{\mathrm{v}} = (t_{\mathrm{CT_o}}, t_{\mathrm{CT_f}}] \implies t_{\mathrm{DE}} \geq t_{\mathrm{CT_o}}$ since all intervals start at $t_{\mathrm{CT_o}} = t_{\mathrm{DE}}$). Reactivation at $t_{\mathrm{DE}} > t_{\mathrm{CT_f}}$ is not possible as it would imply that the DE kernel skipped the reactivation event at $t_{\mathrm{CT_f}}$. There are no more branches to explore, thus HANDLE-REACTIVATION is complete. □

4.3.4.2 Discussion on Liveness

DE simulation in SystemC stagnates when (a) a process fails to give control back to the kernel or when (b) it produces an infinite sequence of delta cycles. Situation (a) does not occur in SYNCHRONIZATION-ALGORITHM because all of its branches reach the WAIT statement. Situation (b) does not occur because we constrain IS-EVENT not to report the same event twice: infinite delta cycles mean that the argument of the WAIT statement is equal to zero an infinite number of times from a given simulation time forward (Zeno system, see [52] for a formal definition). This means that $t_{\mathrm{CT_f}} = t_{\mathrm{DE}} + (0,1)$ at the end of CALCULATE-SOLUTIONS, which is only possible if either EXECUTE-UPDATES or IS-EVENT returns true. Since EXECUTE-UPDATES is triggered by input events (Algorithm 4, line 4), infinite updates can only be caused by infinite such events produced by either a faulty DE component or an unstable loop between the CT and the DE domains; both are modeling errors designers should avoid. IS-EVENT depends on $\boldsymbol{x}$ and $\boldsymbol{i}$ (Algorithm 4, line 5); infinite variations on $\boldsymbol{x}$ without advancing time can only be attributed to EXECUTE-UPDATES, which falls back to the case of infinite variations on $\boldsymbol{i}$, already discussed. When neither $\boldsymbol{x}$ nor $\boldsymbol{i}$ vary, IS-EVENT reports infinitely the same event; to avoid it, we constrain the function not to report the same event over infinitely consecutive microsteps (Section 4.2.3, item 2), for which it should have a memory of the last event or of the arguments that have produced it. A typical example is to record the sign of the derivatives of the state to report a crossing event only when going in one direction.

4.4 Experiments

In this section, we present three CT/DE system case studies, their modeling in our approach, their modeling in SystemC AMS, and their simulation and synchronization challenges (both,

direct and fixed-step). We model and simulate the same systems using MATLAB/Simulink as a reference. We discuss their simulation accuracy and speed and the simulation acceleration provided by the different choices of the integration interval length presented in Section 4.3.3.

For illustration, we provide the code for one of the proposed examples in Appendix A, which we have chosen for its completeness and simplicity of explanation; all other models and algorithms described in this chapter count with a proof-of-concept implementation that we have made available as open source to the research community in [117].

4.4.1 Comparison of Accuracy to Existing Approches

The first case study models the automatic transmission control of a vehicle. We use it to detail the modeling and simulation by our Synchronization-Algorithm and to compare its accuracy and speed to SystemC AMS's TDF synchronization.

4.4.1.1 System Description

This system consists of two parts, the electronic transmission control unit, which can be described in the DE time domain, and the longitudinal dynamics of the vehicle, which can be described in the CT time domain [118]. The transmission control unit selects an appropriate *gear* position according to the vehicle speed: different gears correspond to different speed ranges, an example of how a CT state event (speed change) can affect a DE state (gear). In turn, the selected gear is used to adjust the engine torque and speed to furnish fuel efficiency and driving comfortability, an example of how a discrete event (change in the gear) can affect the CT model (torque and speed). Let us describe the model.

4.4.1.2 Modeling with Our Approach

Figure 4.7 presents the CT Vehicle and its DE Transmission Control Unit models.

1. **CT Vehicle**: this is a CT module that defines the methods required by the modeling interface presented in Section 4.2.3 as follows:

   Get-Derivatives: contains the nonlinear equations that account for the angular acceleration of the engine (Figure 4.7, Equation 1), the vehicle speed (Figure 4.7, Equation 2), and the pressure of the braking system (Figure 4.7, Equation 3) [119]. The K_{tc}, R_{tr}, and C_{tr} parameters are functions of the gear position that is selected in the DE Transmission Control Unit and transmitted via the `gear_sg` signal. The Synchronization-Algorithm is sensitive to the *value changed event* [15] in this signal.

   Is-Event: returns True when the speed crosses a given pair of up and down speed thresholds imposed by the gear. The threshold crossings are output as events (see next item) and provoke gear shifts in the control unit.

   Execute-Updates: returns False, no instantaneous updates are needed in this model.

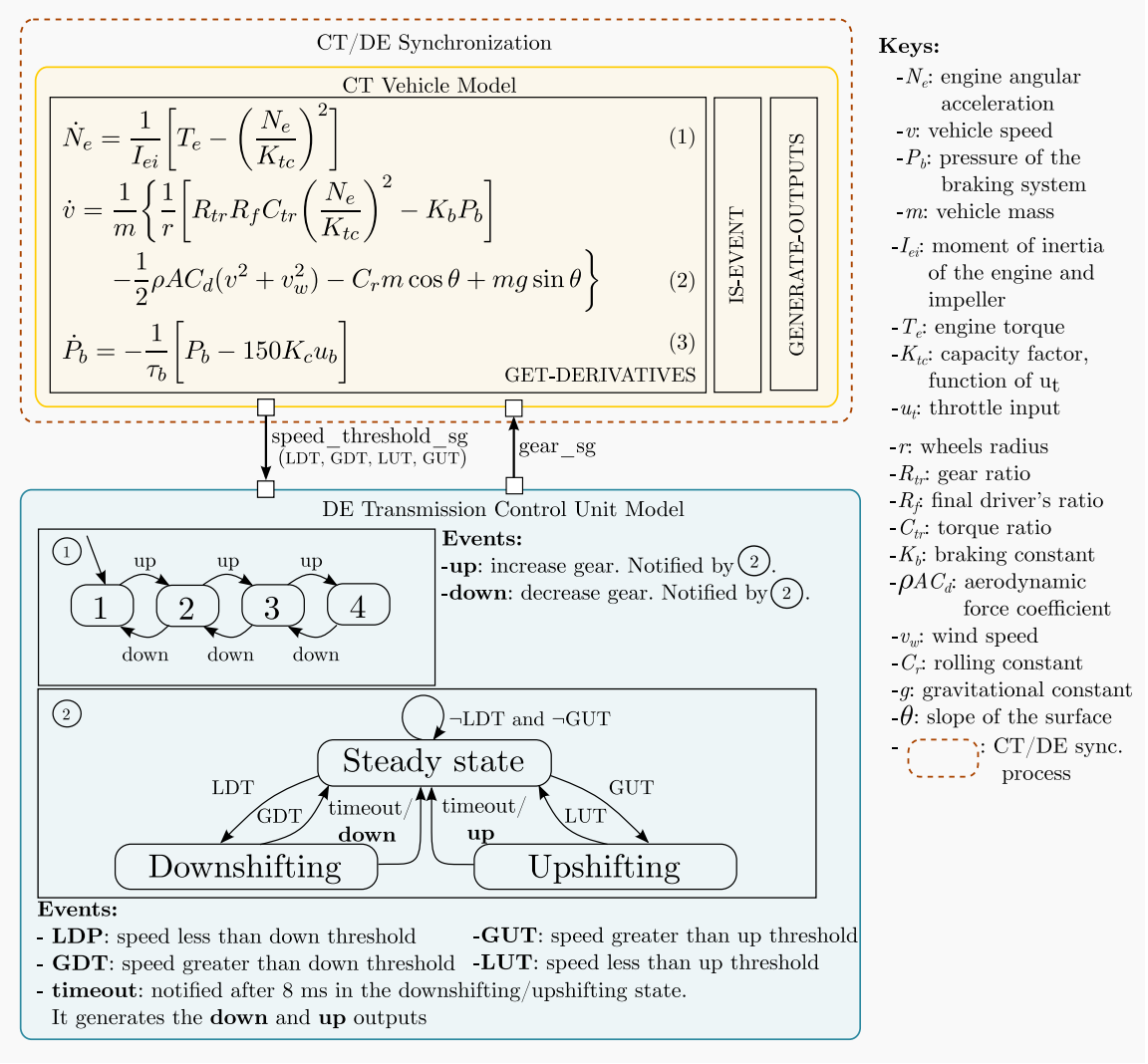
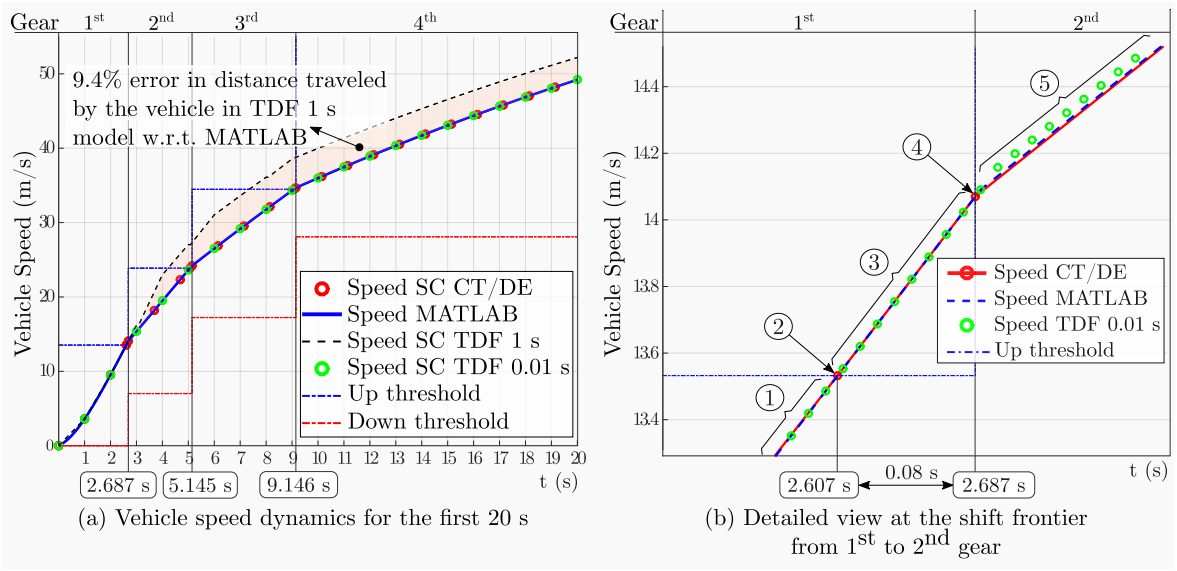


Figure 4.7: Continuous-time and discrete-event longitudinal vehicle dynamics model

GENERATE-OUTPUTS: maps the speed and gear values to an output `speed_threshold_sg` signal indicating the speed position (up or down) w.r.t. the pair of thresholds. Writing in this signal causes the notification of the value changed event.

2. **DE Transmission Control Unit**: this is a standard SystemC module that selects a gear based on the value of `speed_threshold_sg`. The unit is sensitive to the value changed event in this signal. It implements a state machine with four gear positions (State machine ① in Figure 4.7). To filter out false positives, the unit waits for the speed to remain at least 0.08 s above (or below) the threshold before shifting (State machine ② in Figure 4.7) [120]. When the unit writes a new gear position in signal `gear_sg`, the value changed event in the signal activates the SYNCHRONIZATION-ALGORITHM; which reacts by selecting the appropriate parameter values for the equations used in GET-DERIVATIVES and by updating the corresponding speed thresholds used in IS-EVENT.



(a) Vehicle speed dynamics for the first 20 s

(b) Detailed view at the shift frontier from 1st to 2nd gear

Figure 4.8: Vehicle speed dynamics and gear shifts for a constant throttle input

4.4.1.3 Modeling with SystemC AMS TDF

Concerning the CT part, we implement the same nonlinear equations inside a SystemC AMS TDF module to compare the SystemC AMS Proof-of-Concept [121] TDF synchronization to our direct approach. We use the same numerical integration algorithm as in our approach (Runge-Kutta Dormand-Prince 5 from the odeint Boost library [116]) so that we can compare the speed-up from the synchronization strategies and not the integration algorithms. The TDF module contains an object representing the CT equations and another one that instantiates the integration algorithm. At each TDF timestep, the TDF `processing` method (Section 3.5.4.2) calls the algorithm to integrate the equations for an interval of length equal to the TDF timestep. The DE Transmission Control Unit is the same described above.

4.4.1.4 Simulation Challenges

The simulation challenges of this model relate to the accurate detection of state events, change of state event detection conditions, and timely CT/DE direct interactions. We describe the model execution by our SYNCHRONIZATION-ALGORITHM running on top of SystemC.

Figure 4.8 (a) shows the speed response to a constant throttle. At time 0 s, the transmission control unit is in 1st gear and the speed is zero and begins to increase. When it reaches the up threshold, the event is communicated to the control unit to shift to 2nd gear (2.687 s). With a smaller gear ratio, less power goes from the engine to the wheels and the vehicle accelerates more slowly. The 2nd to 3rd upshift occurs when the speed reaches the 2nd gear's up threshold (5.145 s). So on and so forth. To gain better insight, let us describe the interactions in the frontier where the gear shifts from 1st to 2nd in five parts (Figure 4.8 (b)):

1. Before the threshold crossing at time (2.607 s, 0): the control unit is in steady state at 1st

gear. While the synchronization process computes solutions, it detects the up threshold crossing and locates it at time (2.607 s, 0) (CALCULATE-SOLUTIONS, case 2). It schedules a reactivation at this time and suspends (SCHEDULE-REACTIVATION, case 1). The DE simulator advances time to (2.607 s, 0).

2. During the threshold crossing at time (2.607 s, 0): the process reactivates, creates a checkpoint at (2.607 s, 0), and outputs the state event (HANDLE-REACTIVATION, case 2) by writing on signal `speed_threshold_sg`. It computes optimistic solutions over an interval I_v starting at (2.607 s, 0) and ending in the future t_{CT_f} (CALCULATE-SOLUTIONS, case 1)—this time depends on the strategy used to select Δt, e.g., if $\Delta t = 1\,s$, then $t_{CT_f} = (3.607\,s, 0)$. It then schedules a reactivation at t_{CT_f} and suspends (SCHEDULE-REACTIVATION, case 1). The DE simulator informs the value changed event on `speed_threshold_sg` to the control unit, which activates at the next microstep (2.607 s, 1), goes from Steady to Upshifting state, schedules a reactivation within 0.08 s and suspends.

3. Between the threshold crossing and the first gear shift at time (2.687 s, 0): the DE simulator advances time to the next event at (2.607 s + 0.08 s, 0) = (2.687 s, 0).

4. During the gear shift at time (2.687 s, 0): the DE simulator reactivates the control unit, which shifts from 1st to 2nd gear, writes the value to `gear_sg` and suspends. The value changed event in `gear_sg` reactivates the synchronization process at the next microstep (2.687 s, 1): the process identifies an activation in the middle of the last integration interval—$t_{DE} = (2.687\,s, 1) \in I_v = ((2.607\,s, 0), t_{CT_f}]$ —, restores the checkpoint at (2.607 s, 0) and catches up to $t_{DE} = (2.687\,s, 1)$ (HANDLE-REACTIVATION, case 5). Then, it tries to compute optimistic solutions over a new interval, but it cannot because the new up and down thresholds from the gear change make the speed no longer be above the up threshold but below, which is a state event triggered by an input event (CALCULATE-SOLUTIONS, case 4). It schedules a reactivation at the next microstep (2.687 s, 2) and suspends (SCHEDULE-REACTIVATION, case 2). At reactivation, it outputs the event (HANDLE-REACTIVATION, case 3) by writing on `speed_threshold_sg` to inform the control unit that the speed is between the new up and down thresholds. Then, it computes optimistic solutions (CALCULATE-SOLUTIONS, case 1) over a new interval, schedules a reactivation in the future, and suspends (SCHEDULE-REACTIVATION, case 1). The DE simulator updates the `speed_threshold_sg` signal and informs the event to the control unit, which activates at the next microstep $(2.687\,s, 3)$ to go back from Upshifting to Steady state.

5. After the gear shift at time (2.687 s, 3): the synchronization process continues to compute solutions until finding the next threshold crossing similarly to step 1.

Table 4.1: Longitudinal vehicle dynamics simulation accuracy and speed

Model →	MATLAB	TDF 0.01 s	TDF 1 s	SYNC. ALGORITHM (Adaptive Δt)
Wall-clock time (s)	7.7	0.43	0.014	**0.022**
Speed-up factor	1	18	550	**350**
Distance traveled (m)	640	640	700	**640**
Error in the distance w.r.t. MATLAB (%)	0	0	9.4	**0**

4.4.1.5 Simulation Results and Discussion

We take the distance traveled by the vehicle (integral of the speed, area under the curve) as a metric of accuracy and the simulation wall-clock time as a metric of speed. We compare to an equivalent MATLAB/Simulink reference configured to use the ode15s solver with a maximum order of 2, and with $1.0 \cdot 10^{-3}$ relative and $1.0 \cdot 10^{-6}$ absolute tolerances. The TDF model with a timestep of 0.01 s executes 18 × faster than MATLAB/Simulink and provides accurate results; the TDF model with a larger timestep of 1 s executes 550 × faster with 9.4% error, showing the accuracy/speed trade-off of fixed-step synchronization. Direct simulation and synchronization by our Synchronization-Algorithm executes 350 × faster than MATLAB/Simulink and 19 × times faster than SystemC AMS TDF without error.

Fixed-step synchronization in the TDF simulation restricts event notifications to fixed points independently of their exact time of occurrence; since notification times are not accurate, neither are simulation results. Small timesteps increase accuracy but produce more synchronization between the TDF and the DE simulations whose cost accumulates. Our Synchronization-Algorithm avoids inaccuracies by reacting to input events and exactly locating output events, and by adapting the integration interval size according to the frequency of input events to the CT model, which lets the numerical algorithm use larger integration steps. Our algorithm increases execution speed by synchronizing when needed to break the accuracy and efficiency trade-off.

Notice that in this experiment we used the adaptive Δt strategy discussed in Section 4.3.3. In Section 4.4.4 we further discuss the simulation acceleration that other choices provide.

4.4.2 Handling State Discontinuities

The bouncing ball is a classical model from the CT/DE modeling and simulation literature [8, 122]. It presents Zeno behavior and illustrates the need of precisely handling state discontinuities. We discuss how our algorithm faces these challenges. Before, let us describe the system and its model.

4.4.2.1 System Description

A ball experiences a free fall from a given height ($h > 0\,\text{m}$) influenced by the acceleration of gravity ($g = -9.81\,\frac{\text{m}}{\text{s}^2}$) and bounces on a solid surface. Equation 1 in Figure 4.9 describes

these dynamics. At each collision, the speed (v) switches from negative (falling) to positive (rising) (Equation 3 in Figure 4.9). Collisions are non-conservative: energy is lost and the ball rises to a smaller height at each bounce, eventually stopping. Equation 2 in Figure 4.9 models the stopped ball. The system has two DE states: Bouncing and Stopped.

4.4.2.2 Modeling with Our Approach

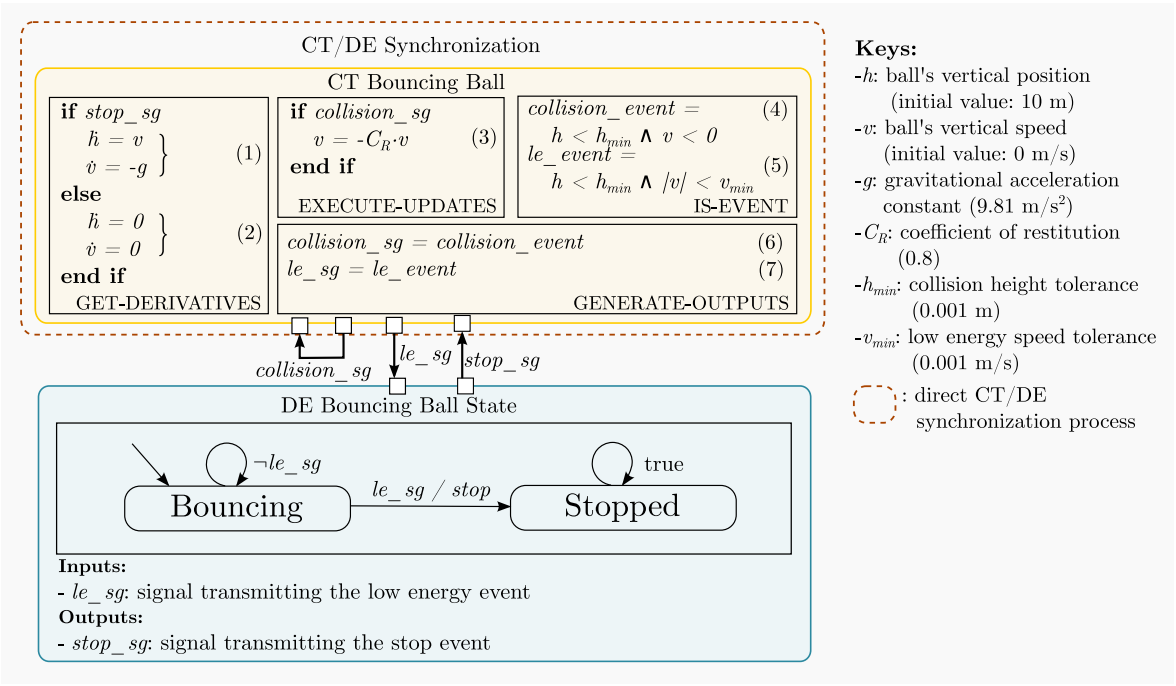


Figure 4.9: Continuous-time and discrete-event bouncing ball model

Figure 4.9 presents the CT Bouncing Ball and the DE Bouncing Ball State models.

1. **CT Bouncing Ball**: this is a CT module that defines the interface functions described in Section 4.2.3 as follows:

   GET-DERIVATIVES: selects between Equation 1 and Equation 2 in Figure 4.9 depending on the DE state that is transmitted by the `stop_sg` boolean signal. The SYNCHRONIZATION-ALGORITHM is sensitive to the value changed event in this signal.

   EXECUTE-UPDATES: instantaneously inverses the speed from falling to rising at each collision reported by the `collision_sg` boolean signal according to Equation 3 in Figure 4.9. The SYNCHRONIZATION-ALGORITHM is also sensitive to the value changed event in this signal.

   IS-EVENT: detects the *collision event* (the ball is falling and its height is near zero) as defined by Equation 4 in Figure 4.9, and the *low energy event* (both, the ball speed and height are near zero, so it should stop bouncing), as defined by Equation 5 in Figure 4.9.

   GENERATE-OUTPUTS: writes the collision and low energy events in the `collision_sg` and `le_sg` signals (Equation 6 and Equation 7 in Figure 4.9).

One peculiarity is that the CT model generates and consumes itself the collision event. This enables EXECUTE-UPDATES to be triggered at each collision. Other DE components could also consume the event to execute DE operations, such as counting the collisions.

2. **DE Bouncing Ball State**: this is a standard SystemC module that implements a state machine that selects between the Bouncing and Stopped states at the notification of the value changed event on the `le_sg` signal. During the transition, it outputs the stop event by writing on the `stop_sg` boolean signal.

4.4.2.3 Modeling with SystemC AMS TDF

Concerning the CT part, we implement the same equations in a SystemC AMS TDF module that contains two objects: one representing the CT equations and the other instantiating the integration algorithm. At each TDF timestep, the **processing** method invokes the algorithm to integrate the equations for an interval of length equal to the TDF timestep. The DE model is the same described above.

4.4.2.4 Simulation Challenges

The authors of [122] argue that this system is Zeno because as $t \to 12.85\,\text{s}$, the time between bounces gets so small that an infinite number occur in finite time. They also claim that the Zeno problem is not easily solved by simulation techniques and they opt for a modeling solution: they use Modal Models in Ptolemy II to define the Bouncing and Stopped DE states and their transition. We have given an overview of this model in Section 3.4.3.1. We rely on standard SystemC models to solve this problem, our DE model is inspired by their solution. If it was not for these DE states, when approaching the Zeno condition, the simulation would slow down and unexpected behaviors would appear, such as the ball falling beneath the surface towards negative infinite values because of small numerical errors [122].

The immediate update to the ball speed at each collision illustrates the need to handle state discontinuities. At the time of a given collision, the ball is falling and the speed is negative. But the collision triggers a state discontinuity to instantaneously make the speed positive (Figure 4.10). Both values are valid at the same time. A one-dimensional model of time ($t \in \mathbb{R}$) would not be enough to represent this phenomenon. Superdense time more appropriate: at $(t_{\text{col}}, 0)$ the speed is negative and at $(t_{\text{col}}, 1)$ it is positive, where t_{col} is the model time of the collision.

4.4.2.5 Simulation Results and Discussion

We take the simulation wall-clock time as a metric of speed and we use a small timestep (0.01 s) in the TDF model to attain high accuracy. We compare to an equivalent MATLAB/Simulink reference configured to use the ode23t solver with $1.0 \cdot 10^{-3}$ relative and $1.0 \cdot 10^{-6}$ absolute tolerances. Table 4.2 shows the results. The TDF model executes $5.72\times$

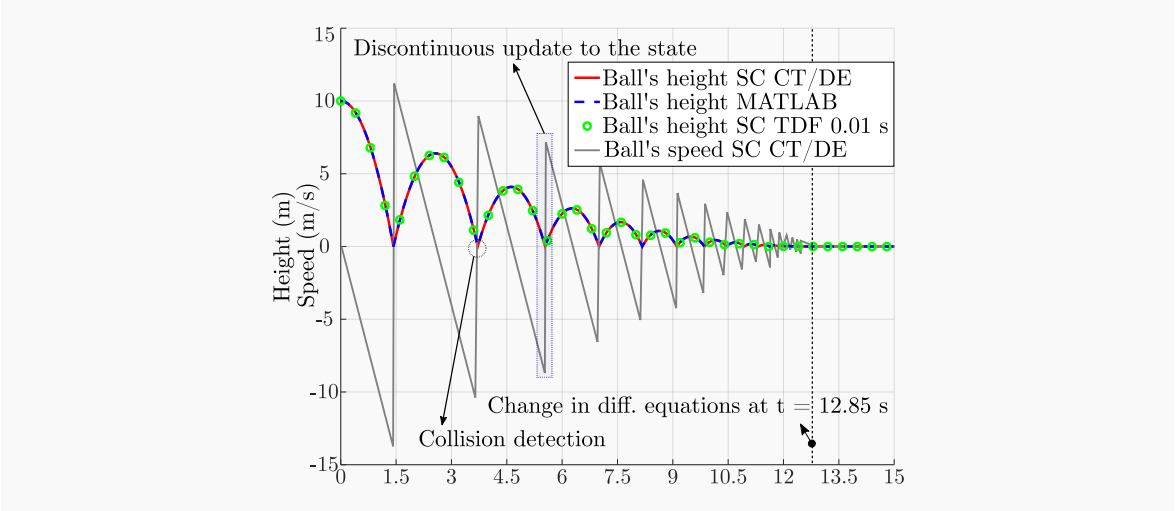


Figure 4.10: Continuous-time and discrete-event bouncing ball dynamics

Table 4.2: Simulation speed of the bouncing ball model

Model	MATLAB	TDF 0.01 s	SYNC. ALGORITHM (Adaptive Δt)
Wall-clock time (s)	0.47	0.0821	**0.00609**
Speed-up factor	1	5.72	**77.2**

faster than MATLAB/Simulink. Direct simulation and synchronization by our approach executes 77.2 × faster than MATLAB/Simulink and 13.5 × faster than SystemC AMS TDF. Our algorithm does not need to constrain the integration interval size to very small values to be able to detect the collision events and to react to them timely.

We used the adaptive Δt strategy. In Section 4.4.4, we discuss other choices.

4.4.3 Handling Error Accumulation

The switched RC circuit case study unveils an intrinsic problem in fixed-step synchronization, the accumulation of error, and motivates the introduction of direct synchronization mechanisms on top of SystemC.

4.4.3.1 System Description

As shown in Figure 4.12, this system is a first-order electrical circuit composed of a capacitor (C), a pair of resistances (R_1 and R_2), a voltage source (V_i), and a DE bang-bang controller. This controller has two DE states, Closed and Open, that serve to maintain the capacitor voltage between a minimum (v_down) and a maximum threshold (v_up). When the controller closes the switch, the capacitor charges; when it opens the switch, the capacitor discharges. Opening and closing occur when the voltage crosses the thresholds.

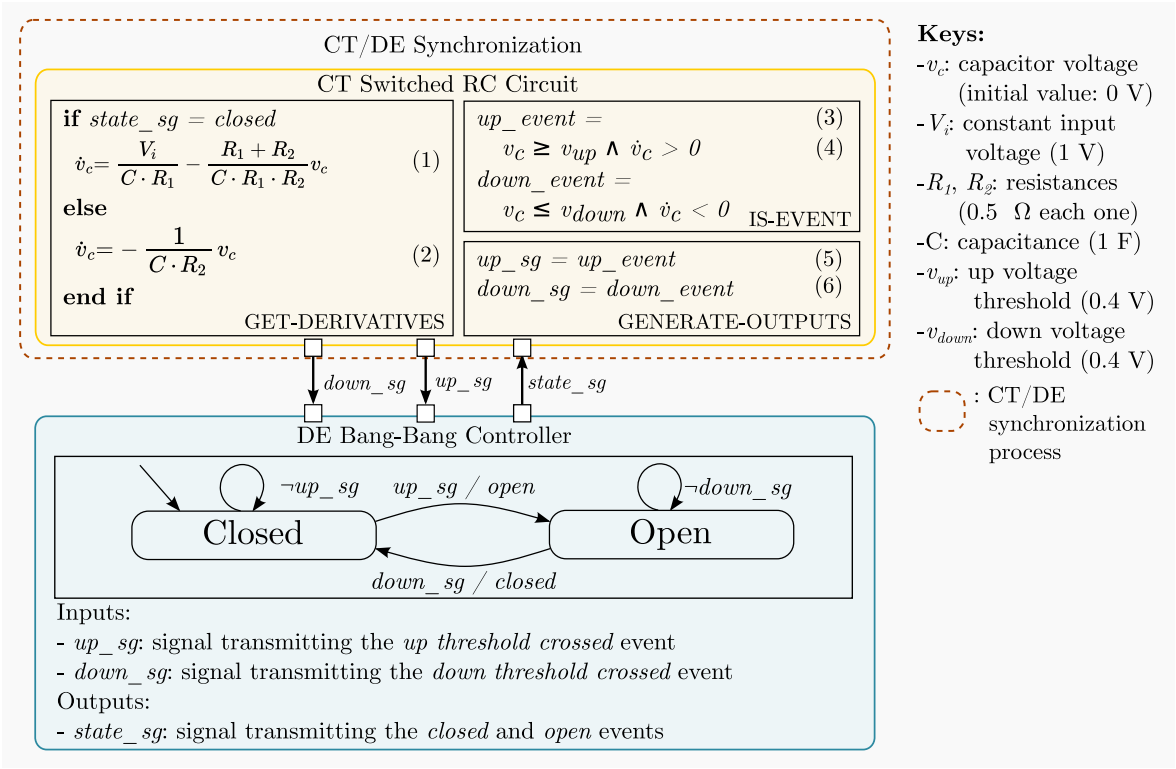


Figure 4.11: Continuous-time and discrete-event switched RC circuit model

4.4.3.2 Modeling with Our Approach

Figure 4.11 presents the CT Switched RC Circuit and its DE Bang-Bang Controller models.

1. **CT Switched RC Circuit**: this is a CT module that defines the interface functions described in Section 4.2.3 as follows:

   GET-DERIVATIVES: selects between Equation 1 and Equation 2 in Figure 4.11 depending on the controller's DE state that is transmitted via the `state_sg` signal. The SYNCHRONIZATION-ALGORITHM is sensitive to the value changed event in this signal.

   EXECUTE-UPDATES: returns FALSE, no instantaneous updates are needed in this model.

   IS-EVENT: detects the capacitor voltage crossing events as defined by Equation 3 and Equation 4 in Figure 4.11.

   GENERATE-OUTPUTS: writes the up and down threshold crossing events in the `up_sg` and `down_sg` as defined by Equation 5 and Equation 6 in Figure 4.11).

2. **DE Bang-Bang Controller**: this is a standard SystemC component that implements a machine with two states, Closed and Open: it goes from Closed to Open when the capacitor voltage crosses the up threshold, and from Open to Closed when the capacitor voltage crosses the down threshold. During state transitions, it outputs events by writing in the `state_sg` to control the switch position.

4.4.3.3 Modeling with SystemC AMS ELN

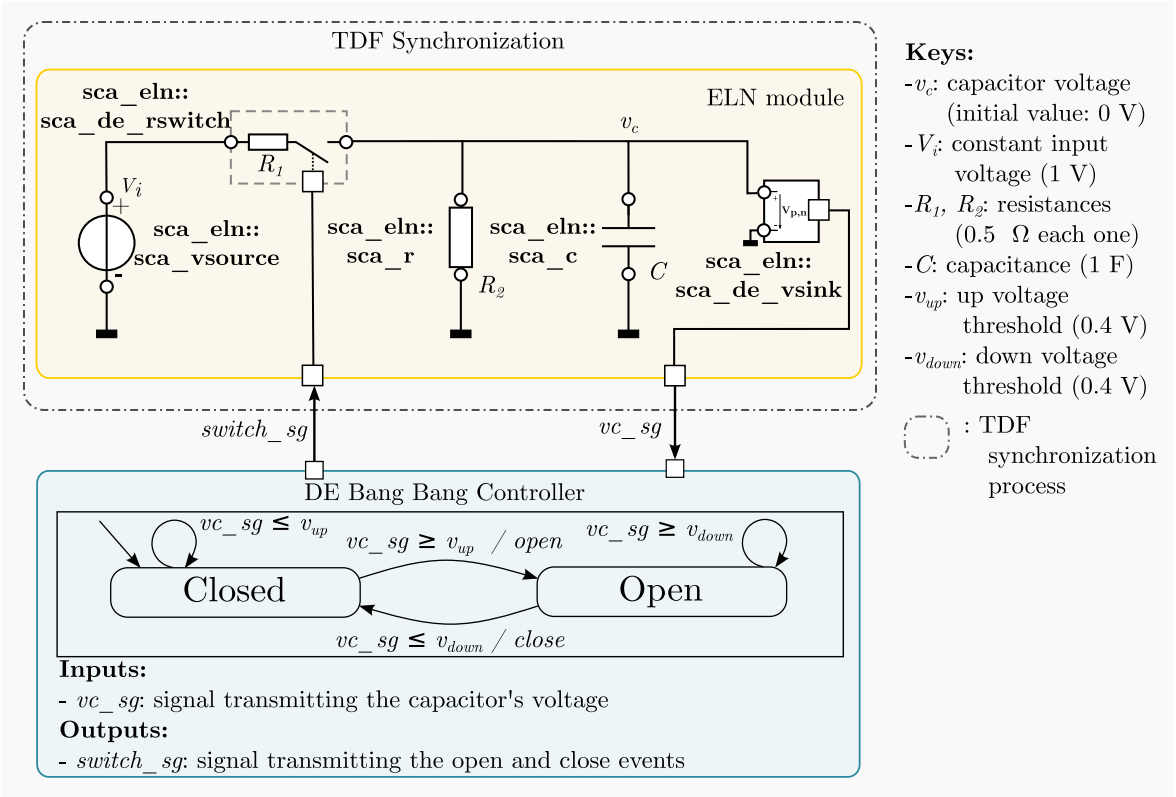


Figure 4.12: SystemC AMS ELN switched RC circuit model

We profit from the SystemC AMS ELN MoC that offers a set of electrical primitives that can be interconnected to build the model, as Figure 4.12 shows. We modify our DE Bang Bang controller to have as input a double signal representing the voltage of the capacitor to be able to detect the crossing events that control the state machine (ELN does not provide threshold crossing detection primitives). Notice that the ELN MoC implements fixed-step synchronization via the SystemC AMS TDF layer (Section 3.5.4.2).

4.4.3.4 Simulation Challenges

The ELN MoC lacks two characteristics to enable direct CT/DE synchronization:

1. State event detection: we are required to use a small timestep (t_{step}) to detect threshold crossings inside the DE module with small *detection error*. In addition, using a root-finding algorithm to reduce this error is impossible because when the threshold is detected in the DE module, the global simulation time has already advanced beyond the exact time of the event.

2. Sensitivity of input events: modules activate only at the TDF fixed steps defined by either the user or the module interconnection to other SystemC AMS modules. Once the DE module generates an open or close event, say at time (t_{e}, δ), the ELN module does

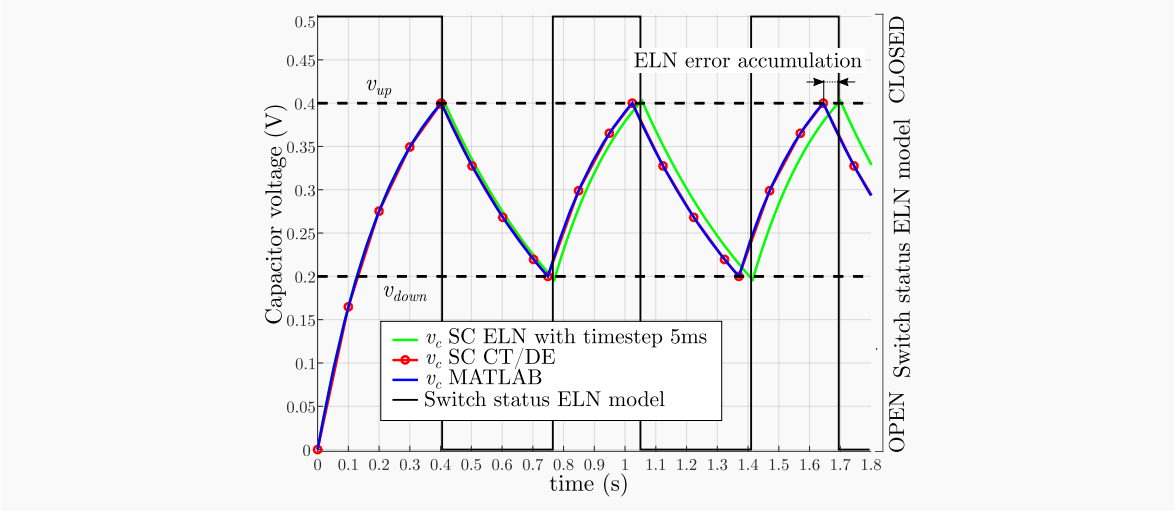


Figure 4.13: Switched RC circuit dynamics

Table 4.3: Simulation speed of the switched RC circuit model

Model	MATLAB	TDF 0.05 s	SYNC. ALGORITHM (Adaptive Δt)
Wall-clock time (s)	0.03	0.00905	**0.0143**
Speed-up factor	1	3.31	**2.09**

not reactivate in the following microstep $(t_e, \delta + 1)$, but advances time to $(t_e + t_{step}, 0)$ and considers the event only from this time forward, which introduces an *activation delay* that is reflected as an error in the simulated capacitor voltage.

If there is a loop between the CT and DE modules, errors from 1) and 2) accumulate. In the ELN circuit model, when t_{step} is small enough, the individual activation delay from 2) is negligible. However, at each crossing, the detection error is still present. The DE module propagates this error back to ELN module when switching. In turn, the ELN module reactivates with an additional delay. This loop adds up errors and rapidly makes results differ from the real system behavior and reference MATLAB/Simulink simulation.

Figure 4.13 shows the simulation results. We handle the accuracy problem by locating state events and reacting to input events timely. As we provide CT modeling mechanisms to evaluate state conditions, our algorithm can locate the exact occurrence time of the state events before the global simulation time advances. However, the ELN MoC in SystemC AMS presents the advantage of modeling the system by interconnecting electrical primitives. We show that it is possible to execute similar MoCs on top of our synchronization algorithm in Chapter 5.

4.4.3.5 Simulation Results and Discussion

We take the simulation wall-clock time as a metric of speed. We compare to an equivalent MATLAB/Simulink reference configured to use the ode15s solver with a maximum order of 2,

Table 4.4: Simulation speed for different Δt selection strategies

Model		**Longitudinal Vehicle Dynamics**	**Switched RC Circuit**	**Bouncing Ball**
Optimal wall-clock time for fixed Δt (s)		0.0212	0.0139	0.00624
Wall-clock time (s)	**Adaptive Δt**	0.0218	0.0143	0.00609
	Adaptive + time to next event	0.0186	0.0138	0.0548
Speed-up w.r.t. optimal fixed Δt	**Adaptive Δt**	**0.972**	**0.972**	**1.02**
	Adaptive + time to next event	**1.14**	**1.01**	**1.14**

and with $1.0 \cdot 10^{-3}$ relative and $1.0 \cdot 10^{-6}$ absolute tolerances. The ELN model with a timestep of 0.05 s executes 3.31 × than MATLAB/Simulink but accumulates error as simulation time advances. Direct simulation and synchronization by our SYNCHRONIZATION-ALGORITHM executes 2.09 × faster than MATLAB/Simulink and is 1.6 × slower than SystemC AMS ELN, but it does not accumulate error. This slow-down is not due to synchronization but to differences in the numerical integration method: ELN uses the Euler method while our solution uses a Runge-Kutta Dorman-Prince 5 method.

4.4.4 Discussion on the Integration Interval Size Effect on Simulation Speed

We have exposed three strategies for the selection of the integration interval size in Section 4.3.3: fixed, adaptive, and given by the next event time. In this section, we compare them in terms of the simulation speed they provide. To this aim, we have used the Longitudinal Vehicle Dynamics, Bouncing Ball, and Switched RC Circuit models of sections 4.4.1, 4.4.2, and 4.4.3.

These models present different event densities. In the Longitudinal Vehicle Dynamics model, events correspond to gear shifts, they are sparse and with progressively increasing time from one event to the next. In the Switched RC Circuit model, events correspond to switchings, they occur at an approximately constant time between events. And in the Bouncing Ball model, events correspond to bouncings, they go from sparse at the beginning of the simulation to frequent just before the ball stops. These models give good coverage for simulation speed study under different event density scenarios.

We use the simulation wall-clock time as a metric of speed. We use the optimal fixed Δt found in the experiments shown in Figure 4.6 and we compare it to the adaptive and next input event strategies. In a real simulation, designers do not have information on the optimal value, and would only be able to set a tentative value that can be far from optimal; hence the interest in an adaptive strategy.

As Table 4.4 shows, the adaptive strategy provides a speed comparable to the one attained by the optimal fixed Δt, with a speed-up factor between 0.972 × and 1.02 × depending on the model. The Bouncing Ball results show that the greater the difference of time between events (presence of sparse and frequent event regions in the same simulation), the more the

chances that our adaptive algorithm will attain higher simulation speed than the optimal fixed Δt: a fixed value risks to be too small for regions with sparse input events, incurring in the penalty of excessive synchronization, and too big for the regions where these events are frequent, incurring in the penalty of computing many solutions that are later rolled back. In all three cases, the adaptive strategy used together with the time to the next event according to Equation (4.3) provides the best speed-up, with a value between 1.01 × and 1.14 ×; this is a valid conclusion only for models where most of the discrete events affect the CT model; in other cases, such as in complex DE models, it is better to use the adaptive strategy alone to avoid CT reactivations in lockstep with the DE simulation for smaller than needed intervals.

4.4.5 Discussion on the Integration Interval Size Convergence and Adaptability

Given the different event densities of the Longitudinal Vehicle Dynamics, Bouncing Ball, and Switched RC Circuit models of sections 4.4.1, 4.4.2, and 4.4.3, respectively, they are also useful to study the convergence of the adaptive strategy to a sequence of interval sizes that depends only on the time between events and not on the initial size guess, and its adaptability to the regions with different event densities along with simulation—properties (a) and (b) in item 2. We discuss the results in what follows.

Figure 4.14 shows the evolution of Δt, calculated by the adaptive strategy, vs. the simulated time for the different models. We have varied the initial guess of Δt for each model. The strategy corrects this guess as simulation time advances and makes it converge to a sequence that only depends on the average time between input events to the CT model—property (a) in item 2. In the Longitudinal Vehicle Dynamics simulation (Figure 4.14, (1)), when the initial guess is below the time of the first input event ($t_{\mathrm{DE}} = 2.69\,\mathrm{s}$, red and green lines in Figure 4.14), the algorithm continually increments the estimate until Δt converges to the sequence. When the initial guess is above the time of the first input event (blue line), a rollback is performed at the second execution, which directly provides a precise first estimate ($\mu_{\mathrm{e}} \leftarrow t_{\mathrm{DE}}$). The time between the first and the second input events at $t_{\mathrm{DE}} = 5.14\,\mathrm{s}$ is almost equal to the time between the start of the simulation and the first event, and Δt remains approximately constant. As the last input event is further away ($t_{\mathrm{DE}} = 9.15\,\mathrm{s}$), Δt progressively increments. After the last event, Δt continues to increment, which lets the integration method take larger integration steps and reduces the number of synchronization points, improving speed. The Switched RC Circuit simulation (Figure 4.14, (2)) comprises input events that are separated by an almost constant factor, which makes Δt also become approximately constant. The Bouncing Ball simulation (Figure 4.14, (3)) presents regions with sparse and frequent events, Δt adapts accordingly—property (b) in item 2.

In summary, the adaptive algorithm provides a near to optimal efficiency no matter the initial guess of Δt, which frees the designer from having to set a fixed Δt before simulation and thus, before knowing the dynamics of the model.

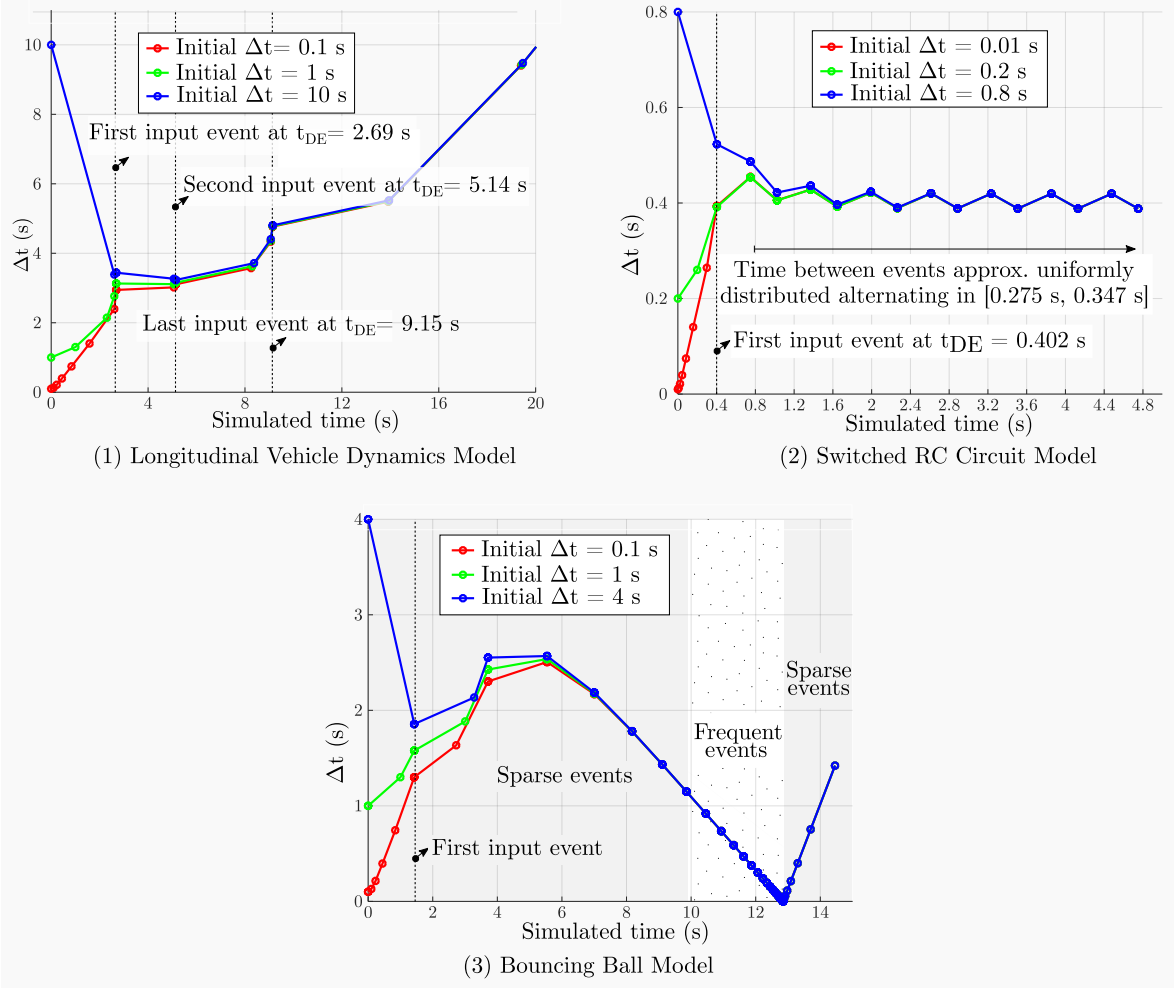


Figure 4.14: Dynamics of the adaptive Δt w.r.t. the simulated time for different models

4.5 Conclusions

In this chapter, we propose a direct synchronization algorithm for the simulation of combined CT/DE models with high accuracy and speed. Our solution is based on direct interactions and a superdense time model. Direct synchronization communicates the CT and DE simulations at the interaction times to preserve simulation accuracy and avoid synchronization overhead. Apart from targeting high-level modeling and simulation in a language that is received by the design community, our solution differentiates from the state of the art in that it allows the CT simulation to optimistically advance over intervals whose size adapts to the frequency of input events. Direct synchronization breaks the accuracy/speed trade-off of fixed-step approaches but adds the modeling costs of having to define a CT modeling interface.

Although simulators such as VHDL-AMS and Verilog AMS already implement a direct approach, these tools are most commonly used for low-level modeling and simulation. Direct synchronization in SystemC enables combined high-level CT/DE modeling and simulation with appropriate accuracy and speed. Other simulators such as Ptolemy II and DEVS-based simulators are mostly used in academic research; given that SystemC is a standardized tool

accepted by the ESL design community, our solution may be also profitable to this community. In addition, our solution preserves its portability and generality by avoiding modifications to the SystemC simulator; it can be used together with the TLM extensions for combined CT/DE virtual prototyping.

Our algorithm allows the CT simulation to optimistically advance over intervals whose size adapts to the frequency of input events. The substantial decrease in synchronization overhead, diminished constraints on the numerical integration steps, and narrowed roll-back costs result in faster simulations. No state-of-the-art approach implements such a mechanism, they are typically limited by the time of the next event, independently of whether or not it is an input to the CT model.

Our algorithm breaks the accuracy/speed trade-off of fixed-step approaches. We have experimentally studied its simulation accuracy and speed and compared them to fixed-step approaches. We show a significant acceleration while maintaining accuracy. We confirm the usefulness of our solution for system simulations.

However, our solution adds some modeling costs. To support direct interactions, it imposes a set of requirements that are expressed in terms of a CT modeling interface. But designers have to pay the cost of defining it manually. In Chapter 5, we show how particular CT models of computation can automatically define this interface to reduce the costs and enable the simulation of CT components from different physical disciplines on top of our algorithm.

Notice also that our solution does not yet exploit the power of parallel computation to accelerate simulation. We explore this path in Chapter 6.

Chapter 5: Direct Synchronization and Continuous-Time Models of Computation

Contents

Ce n'est pas une affaire de volonté, laquelle ne peut pas grand-chose en matière de destin, mais une question de tropisme : se tourner en pensée vers le sommet que l'on convoite, c'est déjà rendre possible son ascension

— Stéphane et Muriel Barbery, Préface à Le sommet des dieux de Jirô Taniguchi et Baku Yumemakura

5.1 Introduction

In Chapter 4, we propose a direct CT/DE synchronization algorithm on top of SystemC. It requires the CT models to comply with a CT modeling interface to support the direct interactions. But this interface increases the modeling cost when designers define it manually. Instead, they typically specify CT models in a graphical representation specific to a physical discipline, e.g., as electrical circuits. It is necessary to support the simulation of these models on top of our algorithm. We base our solution on the concept of model of computation (MoC), which restricts modeling to particular disciplines via a set of primitive modules, channels, ports, and specific operations for their elaboration and simulation.

To illustrate their implementation and simulation on top of our algorithm, we introduce the Signal Flow and Ideally Switched Circuits MoCs in Section 5.2 and Section 5.3, respectively: we describe their modeling primitives, communication channels, ports, and particular definition of our CT modeling interface. Then, in Section 5.4, we make some remarks about the modeling, elaboration, and simulation of both MoCs. In Section 5.5, we present a set of models from power electronics to evaluate the impact on simulation speed of these MoCs. Finally, we present our conclusions in Section 5.6.

5.2 Signal Flow Model of Computation

In this chapter, we consider signal flow models as a set of functional blocks connected by directed signals. The functional blocks apply mathematical operations to the signals. This description corresponds to the block diagram representation of CT system instead of the signal flow representation described in Section 2.4.2.2. However, in received modeling and simulation languages and tools, such as Ptolemy II and SystemC AMS, these models are often referred to as signal flow models. We follow the community usage and call the MoC that we define in this section Signal Flow (SF).

SystemC AMS already provides a Linear Signal Flow MoC on top of SystemC [23] that synchronizes by fixed steps. Inspired by Polemy II's CT MoC [64], our SF MoC includes computational primitives and discrete-event consumer and generator primitives that support direct interactions. The computational primitives include signal sources, adders, gain multipliers, integrators, etc. The discrete-event consumer primitives include multiplexers and demultiplexers that are controlled by boolean discrete-event signals, and integrators whose state is instantaneously set by a real discrete-event signal; these primitives modify the CT model, state, and state conditions. The single discrete-event generator primitive is the threshold crossing detector. Figure 5.1 shows the example of an SF model that represents the equation $\dot{x} = Ax + Bu$ and monitors the value of x for threshold crossing events. SF and other DE models can be interconnected together to constitute combined CT/DE models. By applying graph algorithms on SF models, we can obtain their derivative and output values, as well as their state detection conditions and update rules.

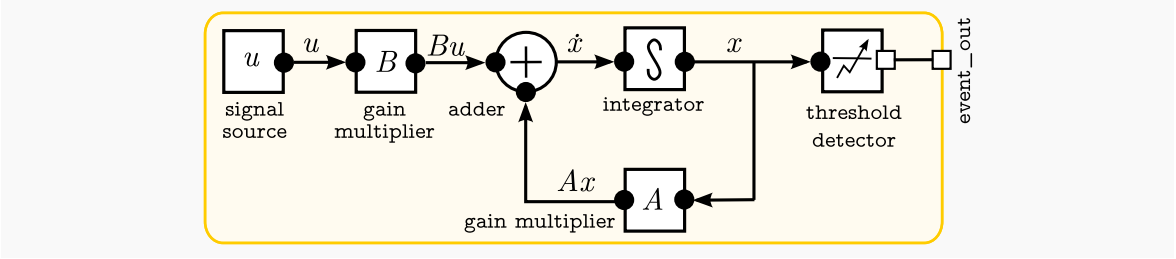


Figure 5.1: Signal Flow model representing equation $\dot{x} = Ax + Bu$

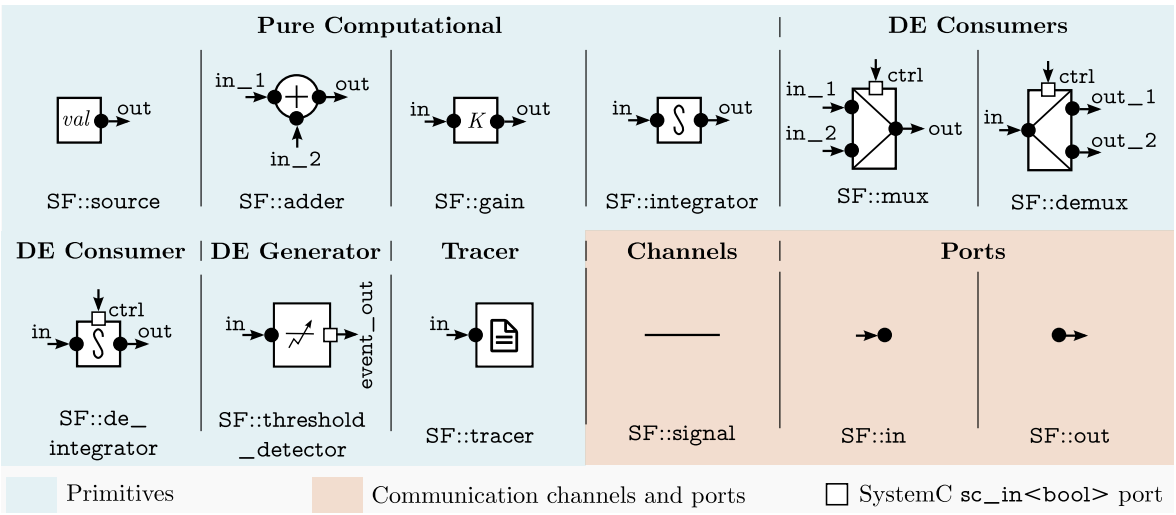


Figure 5.2: Signal Flow modeling primitives

5.2.1 Modeling of Signal Flow Systems

Our SF MoC provides a minimal set of modeling primitives, communication channels, and ports. They are listed in Figure 5.2 and can be grouped as follows:

1. **Pure CT computational primitives**: they allow generating CT signals and operating on them. They are: `SF::source`, which outputs a constant value (*val*) CT signal; SF::adder, which adds two CT input signals and outputs the result in a CT signal; `SF::gain`, which multiplies a CT input signal by a constant factor (K)and outputs the result in a CT signal; and `SF::integrator`, which integrates a CT input signal and outputs the result.

2. **DE consumer primitives**: they consume discrete events and produce changes in the CT model, state, and state conditions. They are: `SF::mux`, which selects between two CT signals according to its DE boolean input controller; `SF::demux`, which routes a CT input signal to one of its two CT outputs according to its DE boolean input controller; and `SF::de_integrator`, which integrates a CT input signal and outputs the result, its state can be set by a DE real input. The `SF::mux` and `SF::demux` primitives modify the CT model and state conditions by selecting and routing signals; `SF::de_integrator` modifies the CT state as a reaction to an input event.

3. **DE generator primitives**: the `SF::threshold_detector` primitive that takes in a CT signal and generates an output event at its DE boolean output port when the signal crosses a given threshold (real `threshold`) in a specified direction (boolean `is_rising`). The `threshold` and `is_rising` values are passed as arguments to the primitive's constructor.

4. **Tracers**: the `SF::tracer` primitive that takes in a CT signal and writes the simulation time and signal value data pairs in a trace file.

5. **Communication channels and ports**: the signal channel (`SF::signal`) can be used to interconnect primitives via their CT signal input and output ports (`SF::in` and `SF::out`, respectively).

SF models follow the SystemC approach: primitives model computation and channels and ports model communication. All primitives are derived from the SystemC `sc_core::sc_module` class. The CT signal channel is derived from the `sc_core::sc_prim_channel` class. The CT input and output ports are derived from the `sc_core::sc_port` class. Class inheritance makes of SF models specializations of SystemC models.

Given a model, the SF MoC automatically executes the elaboration operations for simulation on top of our direct synchronization algorithm, as we describe in Section 5.2.2. More technical details on the implementation of these modeling primitives and each of the methods described in the following sections can be consulted in [117].

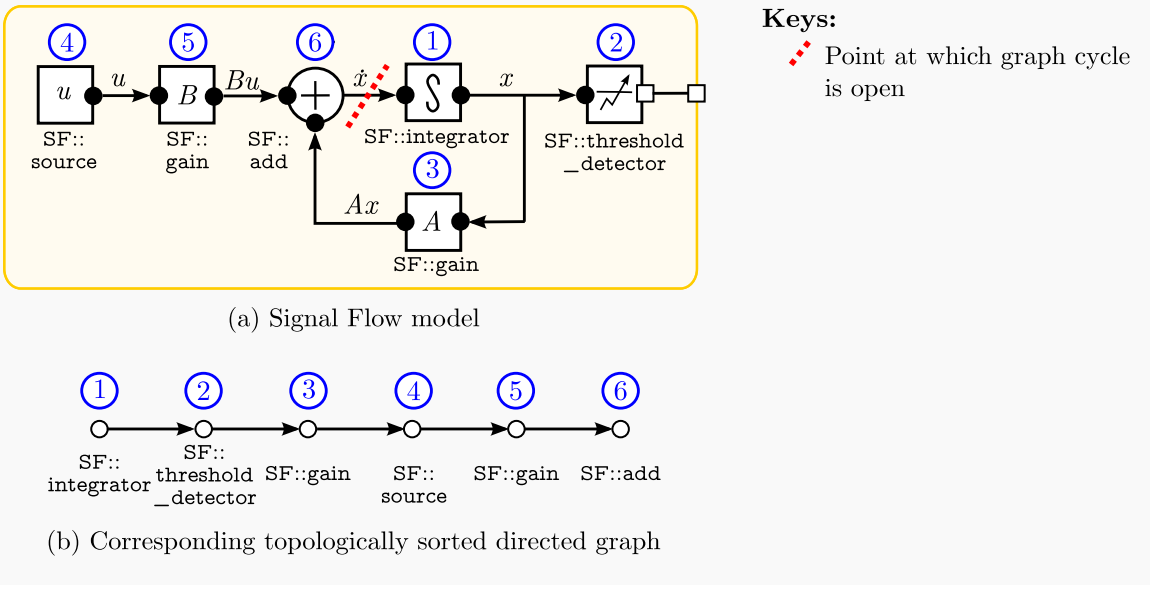
5.2.2 Elaboration of Signal Flow Models

During the SystemC elaboration phase, the SF MoC automatically defines the methods required by our CT modeling interface (Section 4.2.3): Get-Derivatives, which models the CT equations and the instantaneous DE changes in the CT model; Is-Event, which models the state conditions and the instantaneous DE changes in them; Execute-Updates, which models the conditions and rules to instantaneously update the CT model and state as a reaction to discrete events; and Generate-Outputs, which writes output events in the DE output ports. These methods are called during the simulation as part of our direct CT/DE synchronization algorithm. Let us describe the SF MoC definition of each one of them.

5.2.2.1 Get-Derivatives Method

Get-Derivatives is defined to transform the SF model into an equivalent state-space representation. The work in [41] gives the algorithm to obtain the derivative values from the set of interconnected primitives by considering the SF model as an acyclic directed graph. Primitives are vertices and CT signals are edges. Primitives are executed in topological order to get the state derivative values.

Cycles may be present in the model, complicating topological sort, but they can be broken at the integrator primitive inputs. The CT state is defined by the integrator states:



(a) Signal Flow model

(b) Corresponding topologically sorted directed graph

Figure 5.3: Signal Flow model and its execution schedule

to each integrator corresponds one component of the state vector $\boldsymbol{x}$. Integrators are memory components, their state depends on their CT input history from the simulation start time (t_o) until the current time (t_c), but not on the instantaneous input value at the current time—integrating a finite signal over the half-open interval $[t_o, t_c)$ and over the closed interval $[t_o, t_c]$ results in the same values. In other words, integrators can generate their CT output based on their state without needing to know their CT input value at the current time. For this reason, we can break the graph cycle at their inputs without risk of affecting behavior [8].

Once the acyclic directed graph is created, the SF MoC topologically sorts this graph to get an ordered list of primitives. This list gives an execution schedule that starts at the integrator outputs and ends at the integrator inputs. The integrator outputs are fed as inputs to other connected primitives. These primitives execute and generate, in turn, their output values that are fed to other primitives. So on and so forth until the integrator input signals are known, they correspond to the derivatives of the state. Figure 5.3 (a) shows an example SF model: the first to execute is the integrator primitive to output the value of x; then comes the threshold crossing detector and the A multiplier in second and third place, respectively; fourth is the u signal generator and fifth the B multiplier to produce the Bu signal; the last to execute is the adder to produce the value of $\dot{x}$, which is the integrator input. This execution schedule is given by the topologically sorted graph in Figure 5.3 (b).

Algorithm 8 GET-DERIVATIVES($\boldsymbol{x}$, $\boldsymbol{i}$, t)

1: SET-INTEGRATOR-STATE($\boldsymbol{x}$)
2: EXECUTE-ORDERED-PRIMITIVES($\boldsymbol{x}$, $\boldsymbol{i}$, t)
3: **return** GET-DERIVATIVES-FROM-INTEGRATOR-INPUTS()

Algorithm 8 gives the implementation. It invokes SET-INTEGRATOR-STATE, which traverses the set of integrator primitives and sets their state according to $\boldsymbol{x}$. Then, it invokes

Execute-Ordered-Primitives, which executes the primitives in topological order starting by the integrators. Once all primitives have been executed, the values at the integrator input signals are known. They are returned by function Get-Derivatives-From-Integrator-Inputs, which traverses the set of integrators to get and return the derivative values. Notice that the SF MoC passes the DE inputs $\boldsymbol{i}$ to Execute-Ordered-Primitives. Event consumer primitives use these input values for operation. For example, multiplexers select between two CT signals given the value at its DE input port. Such primitives model the discrete event changes in the CT model by affecting the path from the integrator outputs to the integrator inputs, and they also model the DE changes in the state conditions by affecting the path from the integrator outputs to the threshold detector inputs.

5.2.2.2 Is-Event Method

State events can originate from threshold crossing detectors. Algorithm 9 gives the implementation. To detect events, the SF MoC invokes Set-Integrator-State to set the integrator state values from the state vector, then it invokes Execute-Ordered-Primitives to execute the primitives and update all signal values in the model, including those that are threshold detector inputs, and finally, it invokes Is-Threshold-Crossing to test for crossings events. This last method traverses the set of threshold detectors and returns True if there is at least one event and False otherwise.

Algorithm 9 Is-Event($\boldsymbol{x}$, $\boldsymbol{i}$, t)

1: Set-Integrator-State($\boldsymbol{x}$)
2: Execute-Ordered-Primitives($\boldsymbol{x}$, $\boldsymbol{i}$, t)
3: **return** Is-Threshold-Crossing($\boldsymbol{x}, \boldsymbol{i}, t$)

5.2.2.3 Execute-Updates Method

In SF models, the CT state can be updated directly by input events via the `SF::de_integrator` primitive. These integrators have a DE input port that is used to set the integrator state. The SF MoC takes into account the effect of the discrete events on the CT states thanks to Execute-Updates. Algorithm 10 shows the implementation. It invokes Set-Integrator-States-From-De-Inputs, which traverses the set of DE integrators and tests for the value changed event [15] in the connected DE input. If there is an event, it updates the integrator state and its related component in the state vector $\boldsymbol{x}$ according to the signal value. It returns True if there is an update and False otherwise.

Algorithm 10 Execute-Updates($\boldsymbol{x}$, $\boldsymbol{i}$, t)

1: **return** Set-Integrator-States-From-De-Inputs($\boldsymbol{x}, \boldsymbol{i}, t$)

5.2.2.4 Generate-Outputs Method

Output events are generated by the event generator elements. In GENERATE-OUTPUTS, the SF MoC writes the events to the output signals. Algorithm 11 shows the implementation. It sets the integrator states from the state vector (SET-INTEGRATOR-STATE) and executes the primitives to update the signal values (EXECUTE-ORDERED-PRIMITIVES). Then, it traverses the set of SF event generators, such as threshold crossing detectors, and tests for a state event in the element by passing the CT state, DE inputs, and simulation time. If there is a state event, it invokes the *writeOutputEvent* method on the generator to write the event in the corresponding output.

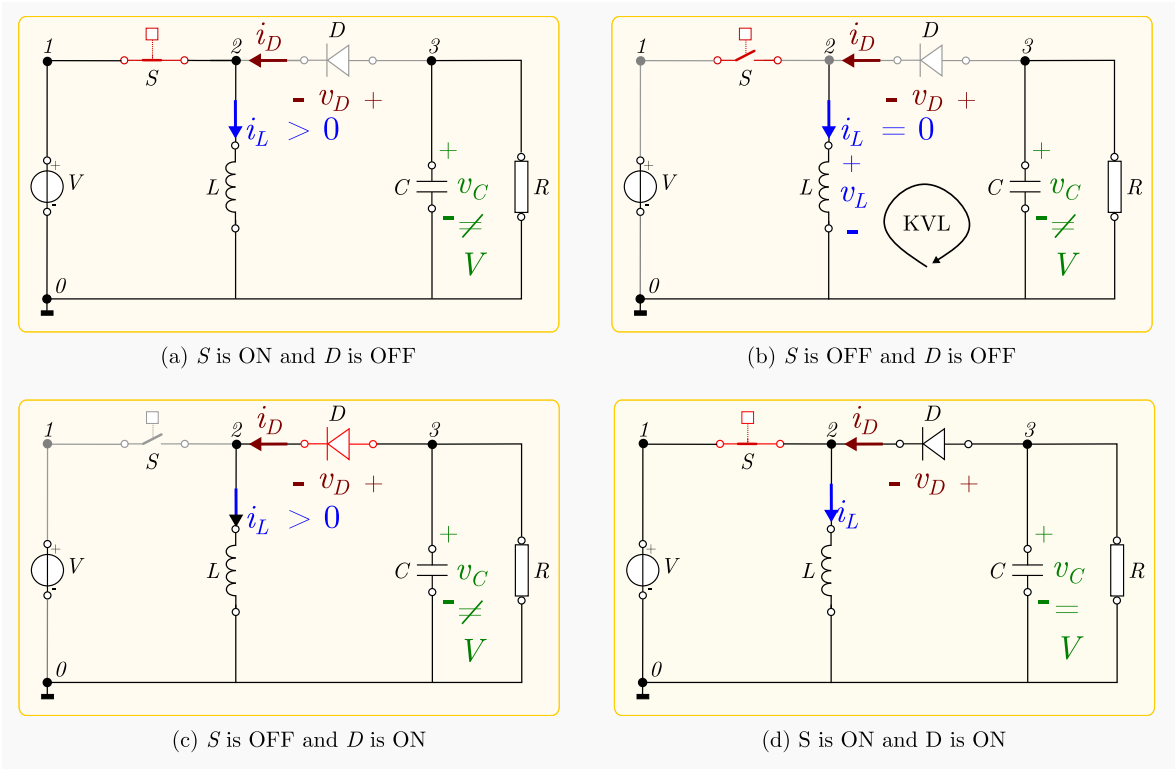
Algorithm 11 GENERATE-OUTPUTS($\boldsymbol{x}$, $\boldsymbol{i}$, t)

```
1: SET-INTEGRATOR-STATE(x)
2: EXECUTE-ORDERED-PRIMITIVES(x, i, t)
3: for all generator ∈ EVENT-GENERATORS do
4:     if generator.hasStateEvent(x, i, t) then
5:         generator.writeOutputEvent(x, i, t)
6:     end if
7: end for
```

5.3 Ideally Switched Circuits Model of Computation

In this section, we define our Ideally Switched Circuits (ISC) MoC. An ideally switched circuit is a network that contains linear elements (resistances, capacitors, inductors, and independent/dependent voltage and current sources) and switches. A switch can be externally controlled by a discrete-event module or internally controlled by the circuit's voltages and currents (e.g., diodes). Switches are ideal: their transitions are instantaneous and they have one discrete state, ON (the voltage across them is zero) or OFF (the current through them is zero). At any instant, the circuit has a particular configuration (*topology*): each switch is either ON or OFF and the circuit is composed of only interconnected linear elements. Figure 5.4 shows the four possible topologies of a Fly-back converter circuit, which contains an externally controlled switch S, and a diode D. To each topology corresponds a set of linear equations that are obtained by applying Kirchhoff's Laws to the interconnected linear elements and that are used for simulation between switching events. Some applications include switching power converters, non-ideal switched capacitor filters, and analog to digital converters [123].

ISC simulation involves a CT part, for the circuit, and a DE part, for the switch control algorithms. The switching events come from either the DE controllers or the circuit voltages and currents, and they produce instantaneous changes in the topology (equations), state, and state conditions [124].



(a) S is ON and D is OFF

(b) S is OFF and D is OFF

(c) S is OFF and D is ON

(d) S is ON and D is ON

Figure 5.4: Fly-back converter circuit topologies, adapted from [10]

5.3.1 Modeling of Ideally Switched Circuit Systems

Our ISC provides a minimal but complete list of predefined primitives, communication channels, and ports. Figure 5.5 shows the primitives in blue and the communication channels and ports in orange, they are grouped as follows:

1. **Pure computational primitives**: the primitives modeling the CT circuit behavior are: independent voltage sources (`ISC::v_source`), independent current sources (`ISC::c_source`), controlled voltage sources (`ISC::cv_source`), controlled current sources (`ISC::cc_source`), capacitors (`ISC::capacitor`), inductors (`ISC::inductor`), and resistors (`ISC::resistor`).

2. **DE consumer primitives**: in the ISC MoC, the only primitive that consumes discrete events is the DE-controlled switch (`ISC::switch_t`), which acts as an open circuit when it is OFF (DE input is FALSE) and as a short-circuit when it is ON (DE input is TRUE), modifying the circuit topology, and the related CT model. As we will see in Section 5.3.2.3, this primitive also modifies the CT state at the switching events.

3. **DE generator primitives**: similar to our Signal Flow MoC, the ISC MoC provides a `ISC::threshold_detector` primitive that takes in a CT signal coming from a metering element and generates an output event at its DE boolean output port when the signal crosses a given threshold (real `threshold`) in a specified direction (boolean `is_rising`).

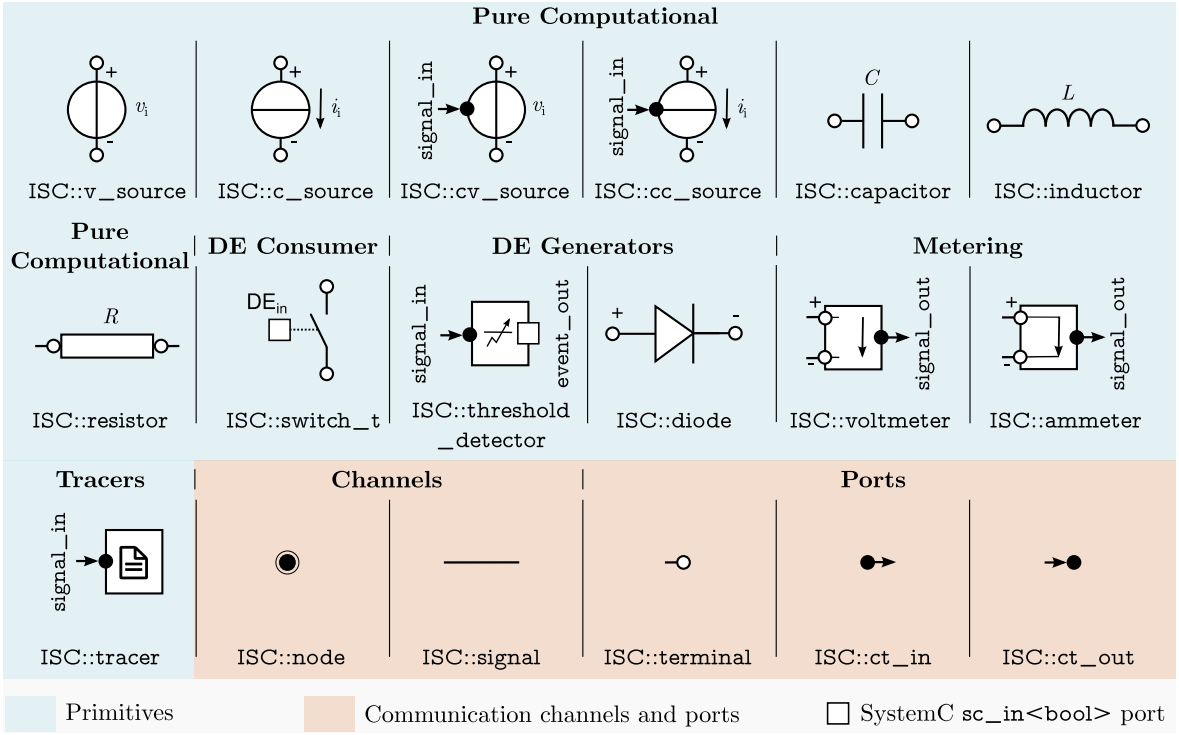


Figure 5.5: Ideally Switched Circuits modeling primitives

The values of `threshold` and `is_rising` are passed as arguments to the primitive's constructor.

In addition, the ISC MoC provides a `ISC::diode` primitive that can be either ON or OFF and that generates state events when its voltage or current changes sign. The effect of these state events is to toggle the diode's state. As the events only affect the own circuit elements, there is no need to write them in the DE outputs.

4. **Metering primitives**: in our ISC MoC, the metering elements are: `ISC::voltmeter`, which measures voltages between two nodes, and `ISC::ammeter`, which measures current in series with a circuit element. The measured value is output in the form of an ISC CT signal that is consumed by the threshold crossing detectors and tracers. They act as intermediary elements that prevent us from having to define separate current and voltage threshold crossing detectors and tracers.

5. **Tracer primitives**: our ISC MoC provides the `ISC::tracer` primitive that takes in a CT signal and writes the simulation time and signal value data pairs in a trace file.

6. **Communication channels and ports**: the electrical primitives have terminals (`ISC::terminal`), which are port-derived classes that can only be bound to node channels (`ISC::node`). The threshold detector, metering, and tracer primitives also have ISC CT signal input and output ports (`ISC::ct_in` and `ISC::ct_out`, respectively), which are ports that can only be bound to ISC CT signal channels (`ISC::signal`). Both `ISC::node` and `ISC::signal` are channels, but the former represents a voltage

point that is shared by the connected primitives and whose value is given w.r.t. a reference node, whereas the second is used to communicate a value read by a metering element, irrespective of whether it is a voltage or current, to the threshold detectors and tracers.

Our ISC MoC provides two basic primitives that model switching behavior: a DE-controlled switch (`ISC::switch_t`) and a diode (`ISC::diode`); other more complex switches, such as thyristors, can be built from these two elements [125].

Similar to the SF MoC, our ISC MoC follows the SystemC modeling approach: primitives model computation and channels and ports model communication. All primitives are derived from the SystemC `sc_core::sc_module` class. The node and CT signal channels are derived from the `sc_core::sc_prim_channel` class. Terminals and CT input and output ports are derived from the `sc_core::sc_port` class. As with SF models, class inheritance makes ISC models specializations of SystemC models.

Designers using this MoC specify models in circuit diagram form. The model elaboration phase is automatically executed by the ISC MoC to define the CT modeling interface described in Section 4.2.3 and required for simulation by our synchronization algorithm. More technical details on the implementation of these modeling primitives and each of the methods described in the following sections can be consulted in [117].

5.3.2 Elaboration of Ideally Switched Circuit Models

In this section, we describe how our ISC MoC automatically defines the Get-Derivatives, Is-Event, Execute-Updates, and Generate-Outputs methods required for simulation by our direct approach (Section 4.2.3).

5.3.2.1 Get-Derivatives Method

The goal of Get-Derivatives is to transform a circuit topology into a set of linear equations in state-space representation. The work in [10] gives the procedure to obtain the state-space representation for each topology in the form of Equation (5.1).

$$\begin{aligned} \dot{\boldsymbol{x}} &= \boldsymbol{A}\boldsymbol{x} + \boldsymbol{B}\boldsymbol{u} \\ \boldsymbol{y} &= \boldsymbol{C}_{\text{imp}}\dot{\boldsymbol{x}} + \boldsymbol{C}_{\text{non-imp}}\boldsymbol{x} + \boldsymbol{D}\boldsymbol{u} \end{aligned} \tag{5.1}$$

where:

- $\boldsymbol{x} = \begin{bmatrix} \boldsymbol{v}_{\text{c}} & \boldsymbol{i}_{\text{l}} \end{bmatrix}^T$ are the state variables: capacitor voltages ($\boldsymbol{v}_{\text{c}}$) and inductor currents ($\boldsymbol{i}_{\text{l}}$).
- $\dot{\boldsymbol{x}} = \begin{bmatrix} \dot{\boldsymbol{v}}_{\text{c}} & \boldsymbol{i}'_{\text{l}} \end{bmatrix}^T$ are the derivatives of the state variables.
- $\boldsymbol{y} = \begin{bmatrix} \boldsymbol{i}_{\text{sc}} & \boldsymbol{v}_{\text{oc}} \end{bmatrix}^T$ are the output currents of ammeters and ON-state switches ($\boldsymbol{i}_{\text{sc}}$) and the voltages of voltmeters and OFF-state switches ($\boldsymbol{v}_{\text{oc}}$).

- $\boldsymbol{u} = \begin{bmatrix} \boldsymbol{v}_\mathrm{E} & \boldsymbol{i}_\mathrm{J} \end{bmatrix}^T$ are the voltages ($\boldsymbol{v}_\mathrm{E}$) and currents ($\boldsymbol{i}_\mathrm{J}$) of the independent voltage and current sources.

- $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}_\mathrm{imp}, \boldsymbol{C}_\mathrm{non-imp}, \boldsymbol{D}$ are the state-space matrices.

The output equations for $\boldsymbol{y}$ include an impulsive $\boldsymbol{C}_\mathrm{imp}\dot{\boldsymbol{x}}$ and non-impulsive component $\boldsymbol{C}_\mathrm{non-imp}\boldsymbol{x}$. The impulsive component allows calculating output values at the switching events, where the state $\boldsymbol{x}$ may change discontinuously implying an impulsive quantity in its derivative $\dot{\boldsymbol{x}}$, as discussed more generally in Section 3.2, item 3. The non-impulsive components allow calculating output values in regular CT operation between events. Both output values are used in the definition of Execute-Update in Section 5.3.2.2.

For example, Equation (5.2) gives the state-space representation of topology (a) in Figure 5.4, where $\dot{\boldsymbol{x}} = \begin{bmatrix} \dot{v}_\mathrm{C} & i'_\mathrm{L} \end{bmatrix}^T$, $\boldsymbol{x} = \begin{bmatrix} v_\mathrm{C} & i_\mathrm{L} \end{bmatrix}^T$, $\boldsymbol{u} = [V]$, and $\boldsymbol{y} = \begin{bmatrix} v_\mathrm{D} \end{bmatrix}$. Similarly Equation (5.3) gives the state-space of topology (c) in Figure 5.4, where the output variable is now the ON-state diode current $\boldsymbol{y} = \begin{bmatrix} i_\mathrm{D} \end{bmatrix}$ [10]. Different topologies have different state-space representations, in what follows we denote topology j's state-space matrices by $\boldsymbol{A}_\mathrm{j}, \boldsymbol{B}_\mathrm{j}, \boldsymbol{C}_\mathrm{non-imp_j}, \boldsymbol{C}_\mathrm{imp_j}, \boldsymbol{D}_\mathrm{j}$.

$$\begin{aligned} \dot{\boldsymbol{x}} &= \begin{bmatrix} -\frac{1}{RC} & 0 \\ 0 & 0 \end{bmatrix} \boldsymbol{x} + \begin{bmatrix} 0 \\ \frac{1}{L} \end{bmatrix} \boldsymbol{u} \\ \boldsymbol{y} &= \begin{bmatrix} 0 & -L \end{bmatrix} \dot{\boldsymbol{x}} + \begin{bmatrix} -1 & 0 \end{bmatrix} \boldsymbol{x} + \begin{bmatrix} 0 \end{bmatrix} \boldsymbol{u} \end{aligned} \tag{5.2}$$

$$\begin{aligned} \dot{\boldsymbol{x}} &= \begin{bmatrix} -\frac{1}{RC} & \frac{1}{C} \\ -\frac{1}{L} & 0 \end{bmatrix} \boldsymbol{x} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \boldsymbol{u} \\ \boldsymbol{y} &= \begin{bmatrix} 0 & 0 \end{bmatrix} \dot{\boldsymbol{x}} + \begin{bmatrix} 0 & 1 \end{bmatrix} \boldsymbol{x} + \begin{bmatrix} 0 \end{bmatrix} \boldsymbol{u} \end{aligned} \tag{5.3}$$

Algorithm 12 gives the implementation. It computes and returns the derivatives values that correspond to the current topology j based on Equation 5.1.

Algorithm 12 Get-Derivatives($\boldsymbol{x}$, $\boldsymbol{i}$, t)

1: **return** $\boldsymbol{A}_\mathrm{j}\boldsymbol{x} + \boldsymbol{B}_\mathrm{j}\boldsymbol{u}$ // $\dot{\boldsymbol{x}} = \boldsymbol{A}_\mathrm{j}\boldsymbol{x} + \boldsymbol{B}_\mathrm{j}\boldsymbol{u}$ for topology j

5.3.2.2 Is-Event Method

State events can originate from threshold crossing detectors (`ISC::threshold_detector`) and from diodes (`ISC::diode`). The state of diodes is controlled by the internal circuit voltages and currents. For example, when an ON-state diode current goes from positive to negative (state event), the diode should immediately turn OFF (reaction); similarly, when an OFF-state diode voltage goes from negative to positive (state event), the diode should immediately turn ON (reaction). In Figure 5.4 (b), diode D must go from OFF to ON

whenever its voltage v_{D} is greater than zero. The simulator detects and locates these state events to update the topology and integrate the right equations.

The external switching events may also trigger state events in the diodes. At the external events, the voltage of the capacitors or the current of the inductors may change discontinuously, which implies an infinite current flow through the capacitor or an infinite voltage across the inductor in zero time (delta impulse). The topologies that imply such discontinuities are not physically possible and are referred to as *inconsistent topologies*. For example, in Figure 5.4, the transition from topology (a), where the inductor current i_{L} is greater than zero, to topology (b), where it suddenly goes to zero, creates a negative delta impulse in the inductor voltage (discharge). The negative impulse in the inductor voltage is reflected as a positive impulse the diode's voltage [1], meaning that the diode should turn ON. These delta impulses indicate the set of diodes to toggle to obtain a consistent topology [126]. State events in diodes triggered by external discrete events are a real-life example of case 4 in Section 4.3.1.2 (state events triggered by input events).

Algorithm 13 IS-EVENT($\boldsymbol{x}$, $\boldsymbol{i}$, t)

1: $\dot{\boldsymbol{x}} \leftarrow$ GET-DERIVATIVES($\boldsymbol{x}, \boldsymbol{i}, t$)
2: $\boldsymbol{y} = \boldsymbol{C}_{\mathrm{imp_i}}\dot{\boldsymbol{x}} + \boldsymbol{C}_{\mathrm{non-imp_i}}\boldsymbol{x} + \boldsymbol{D}_{\mathrm{i}}\boldsymbol{u}$ // CT outputs, Equation (5.1).
3: **return** IS-THRESHOLD-CROSSING($\boldsymbol{x}, \boldsymbol{i}, t$) **or** IS-DIODE-EVENT($\boldsymbol{y}$)

Algorithm 13 shows the implementation: it gets the derivative values and calculates the CT output values according to Equation (5.1), then it invokes methods IS-THRESHOLD-CROSSING and IS-DIODE-EVENT, and returns TRUE if there is a state event and FALSE otherwise. IS-THRESHOLD-CROSSING traverses the set of threshold crossing detectors and tests for the crossing event; it returns TRUE if there is at least one event and FALSE otherwise. IS-DIODE-EVENT takes in the CT output values, traverses the set of diodes, and tests for state events in their CT output control variables in $\boldsymbol{y}$—currents for ON-state and voltages for OFF-state switches: if the control variable changes sign, it returns TRUE, otherwise it returns FALSE. Switches with external events and diodes with impulses are toggled in EXECUTE-UPDATES to obtain the new topology.

5.3.2.3 Execute-Updates Method

When a switch changes its state, the circuit gets a new topology and linear equations. Externally-controlled switching is triggered by the DE controller events; internally-controlled switching is triggered by the diode state events. In Figure 5.4, the transition from topology (a) to (b) is triggered by an external discrete event on S, while the transition from (b) to (c) is triggered by a state event on D. The ISC MoC implements in EXECUTE-UPDATES the conditions and rules to update the circuit equations so that the simulator can make the circuit react to the switching events timely and independently of the steps used for the integration of the equations and synchronization. Notice that as the diode state is updated, the

[1] By applying Kirchhoff's voltage law: if $v_{\mathrm{L}} = v_{\mathrm{C}} - v_{\mathrm{D}}$ and $v_{\mathrm{L}} \to -\infty$ then $v_{\mathrm{D}} \to \infty$ for any finite v_{C}

state conditions to detect the state events also change: for ON-state diodes, these conditions depend on the diode currents, and for OFF-state diodes, they depend on the diode voltages.

Algorithm 14 EXECUTE-UPDATES($\boldsymbol{x}$, $\boldsymbol{i}$, t)

```
1: INTERNAL-SWITCHES ← GET-INTERNAL-SWITCHES-TO-TOGGLE(x, i, t)
2: EXTERNAL-SWITCHES ← GET-EXTERNAL-SWITCHES-TO-TOGGLE(x, i, t)
3: return TOGGLE-SWITCHES(INTERNAL-SWITCHES ∪ EXTERNAL-SWITCHES, x, i, t)
```

Algorithm 14 shows the implementation. It invokes GET-INTERNAL-SWITCHES-TO-TOGGLE to get the set of internally controlled switches (diodes) having a state event, according to the rules explained in Section 5.3.2.2. Then, it invokes GET-EXTERNAL-SWITCHES-TO-TOGGLE to get the set of externally-controlled DE switches whose control input value has changed since the last execution. Finally, it calls TOGGLE-SWITCHES passing the set union of INTERNAL-SWITCHES and EXTERNAL-SWITCHES, the state, the DE inputs, and the simulation time, to update the topology and equations. TOGGLE-SWITCHES sets the new switch states and obtains the new linear state-space circuit matrices ($\boldsymbol{A}_\mathrm{j}, \boldsymbol{B}_\mathrm{j}, \boldsymbol{C}_\mathrm{j}, \boldsymbol{C}_{\mathrm{imp_j}}, \boldsymbol{D}_{\mathrm{non-imp_j}}$, for topology j) by following the algorithm described in [10]; it returns TRUE if at least one switch has been toggled and FALSE otherwise.

5.3.2.4 Generate-Outputs Method

Output generation is similar to the one in the SF MoC: the ISC MoC calculates the CT outputs that contain the values of the ammeters and voltmeters connected to the event generators, such as threshold detectors, then it traverses the set of event generators to test for and write state events on the output ports, as shown in Algorithm 15.

Algorithm 15 GENERATE-OUTPUTS($\boldsymbol{x}$, $\boldsymbol{i}$, t)

```
1: ẋ ← GET-DERIVATIVES(x, i, t)
2: y = C_imp_i ẋ + C_non-imp_i x + D_i u        // CT outputs, Equation (5.1).
3: for all generator ∈ EVENT-GENERATORS do
4:     if generator.hasStateEvent(y) then
5:         generator.writeOutputEvent(y)
6:     end if
7: end for
```

5.4 Remarks about Modeling, Elaboration, and Simulation of Signal Flow and Ideally Switched Circuit Systems

Once the SF and ISC MoCs automatically define the methods required by our modeling interface during the elaboration of a given model. These methods are called during simulation by the direct synchronization algorithm that we propose in Chapter 4. This way, these MoCs can benefit from the speed-ups of our algorithm when compared to fixed-step approaches.

Our CT modeling interface separates modeling and elaboration, which are particular to each CT MoC, from simulation on top of a DE simulator via direct synchronization, which is common to all CT MoCs. This facilitates the definition of new CT MoCs without the need to design and implement a new CT/DE synchronization approach.

In addition, these CT MoCs reduce the modeling cost of defining the interface methods manually. To this end, they provide not only computational primitives but also event generator and consumer primitives. This way, the modeling interface and the CT/DE simulation aspects remain transparent to the designer who only specifies models by interconnecting primitives.

To illustrate modeling and discuss the effect on simulation speed of these MoCs, we present a set of experiments in Section 5.5.

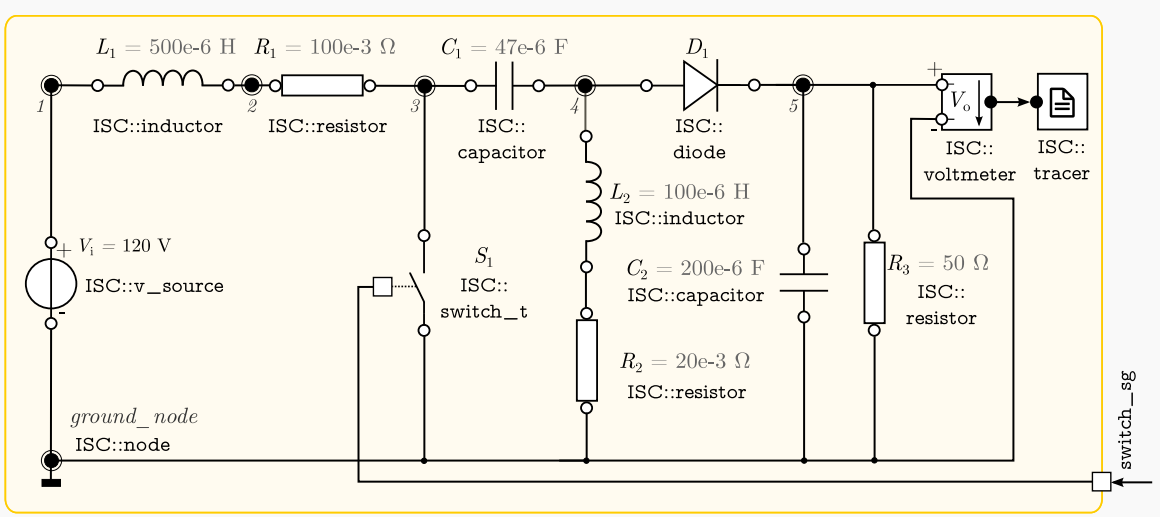
5.5 Experiments

In this section, we model and simulate three systems: Single-Ended Primary-Inductor, Boost, and Cuk power converters. We compare our simulation results to those of three specialized tools at different levels of abstraction: NGSPICE (low level) [127], MATLAB/Simulink Specialized Power Systems Library (medium level) [128], and PLECS (high level) [129]. Together with these models, we present an SF model of an RC Circuit system and its simulation profiling data to gain insights on the computational cost of the elaboration phase and our CT modeling interface method execution. We discuss the speed-up advantages of high-level modeling and simulation and direct synchronization.

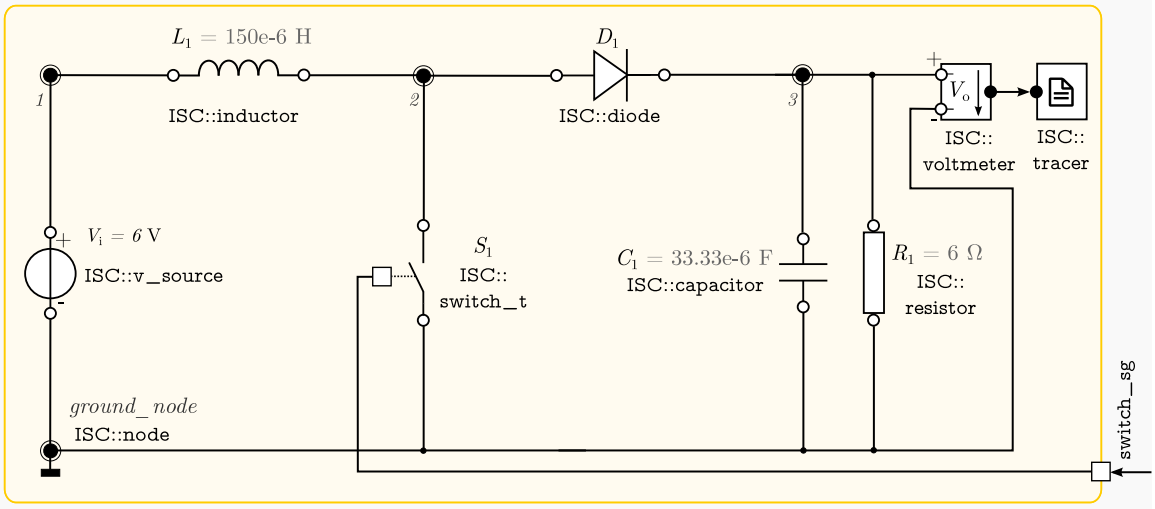
All models and algorithms described in this chapter count with a proof-of-concept implementation that we have made available as open source to the research community in [117].

5.5.1 Converter Systems and Models

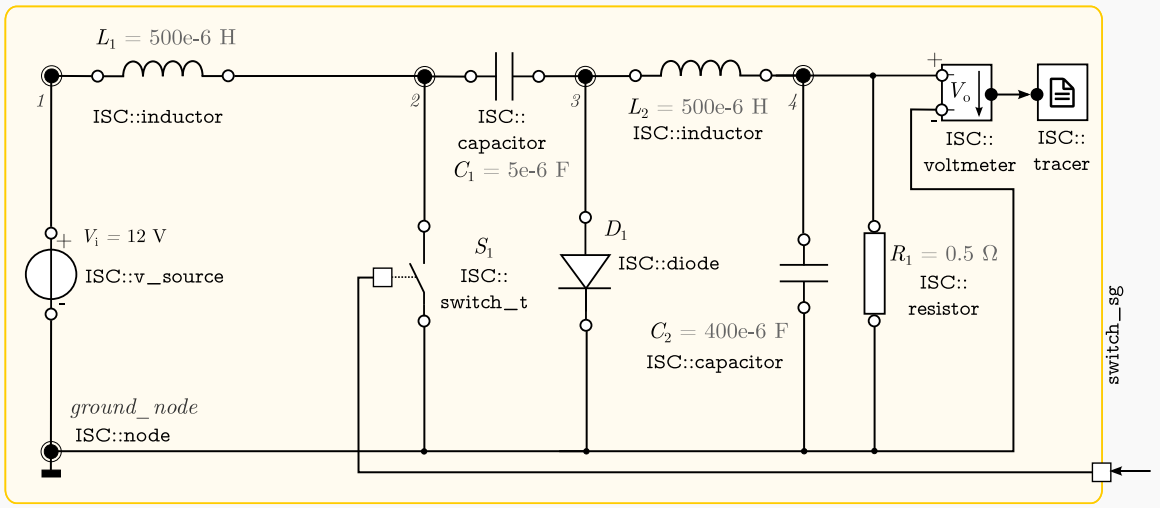
The Single-Ended Primary-Inductor Converter (SEPIC), Boost converter, and Cuk converter, are electronic circuits that convert a DC voltage input to a higher, equal, or lower DC voltage output. Figure 5.6 shows these three ISC models. The SEPIC model is the most complex and it involves 4 energy storage elements: 2 capacitors and 2 inductors, resulting in a 4^{th} order system of ordinary differential equations. The Boost and Cuk converter models involve one capacitor and one inductor each, resulting in 2^{nd} order systems. The three models include an externally controlled switch and a diode that enable $2^2 = 4$ possible different topologies—one for each switch and diode state combination. For all converters, the externally controlled switch state is driven by a DE module that acts as a pulse generator (Figure 5.7): it keeps the switch ON by a fraction of a constant period (t_{on}), after which it transitions to OFF for the remaining of the period (t_{off}), and then back to ON. It triggers the *open* and *close* events in the `switch_sg` signal. The proportion of time that the switch is ON regulates the amount of energy that goes from the input voltage source (V_{i}) to the output (V_{o}). Listing 1 shows a code excerpt where the user interconnects the ISC modeling elements to define the SEPIC



(a) SEPIC converter model in the ISC MoC



(b) Boost converter model in the ISC MoC



(c) Cuk converter model in the ISC MoC

Figure 5.6: Power converter models in the Ideally Switched Circuits model of computation

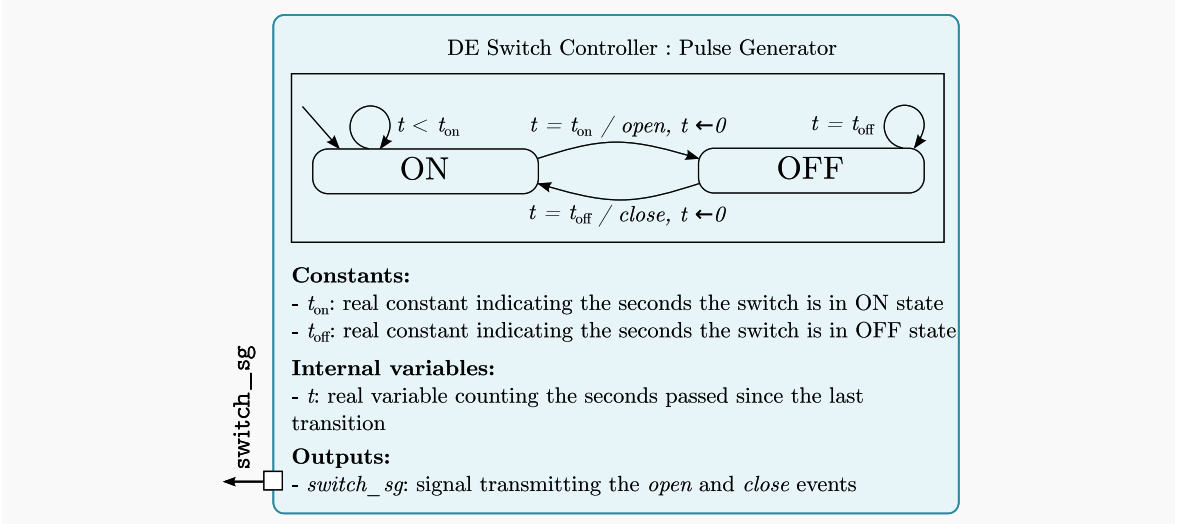


Figure 5.7: Discrete-event switch controller model specified in SystemC

```
sepic::sepic(sc_core::sc_module_name name): // Give a name and a value to the primitives
    switch_ctrl_in("switch_ctrl_in"), source_el("source_el", 120),
    switch_el("switch_el"), ind_1_el("ind_1_el", 500e-6), node_1("node_1"), //...
{   // Interconnect the primitives
    source_el.terminal_a(ground_node);
    source_el.terminal_b(node_1);

    ind_1_el.terminal_a(node_1);
    ind_1_el.terminal_b(node_2);

    res_1_el.terminal_a(node_2);
    res_1_el.terminal_b(node_3);

    switch_el.terminal_a(node_3);
    switch_el.terminal_b(ground_node);
    switch_el.ctrl_in(switch_ctrl_in); // ... Connect the rest of primitives similarly
}
```

Listing 1: Code excerpt of the SEPIC module constructor

model in our ISC MoC, in accordance with the SystemC modeling approach.

To compare the effect on simulation speed of the CT level of abstraction implemented by the ISC MoC, we have modeled similar systems in NGSPICE, MATLAB/Simulink Specialized Power Systems Library, and PLECS. NGSPICE provides low-level detailed models: a diode model, for example, has 43 parameters, and the resulting mathematical representation is nonlinear. MATLAB/Simulink switches and diodes are simpler but still include non-ideal snubber circuits. PLECS models are at high level, their switches and diodes are ideal resulting in linear circuit equations for the different topologies; it is one of the reference industrial tools for high-speed simulation of power electronic systems. We have adjusted the NGSPICE and MATLAB models' non-ideal parameters to make their effect negligible. We take PLECS as the reference to compare the simulation accuracy and speed.

Table 5.1: Wall-clock time for different power converter models

Circuit	Statistics	Tool			
		NGSPICE	**MATLAB**	**PLECS**	**ISC**
SEPIC (0.04 s)	Wall-Clock time (s)	8.48	3.08	0.331	**0.188**
	Speed-Up	0.0390	0.107	1	**1.76**
Boost (0.0024 s)	Wall-Clock time (s)	0.534	0.475	0.0350	**0.0131**
	Speed-Up	0.0655	0.0740	1	**2.67**
Cuk (0.02 s)	Wall-Clock time (s)	0.990	0.938	0.0670	**0.0570**
	Speed-Up	0.0677	0.0714	1	**1.20**

5.5.2 Simulation Results and Discussion

Let us first discuss the effect of the CT level of abstraction on simulation speed for these ISC models and then discuss the cost of elaboration and CT interface methods execution for the SF and ISC models.

5.5.2.1 Effect of the CT Level of Abstraction on Simulation Speed

Table 5.1 shows the wall-clock simulation time and the speed-up w.r.t. PLECS for the three models. NGSPICE takes the longest to simulate because of its very low-level detailed models that reflect non-ideal phenomena. MATLAB/Simulink switches and diodes are simpler but still include non-ideal snubber circuits. Additionally, MATLAB/Simulink deduces the topology equations from start to end at each switching event, which adds non-negligible computational costs. PLECS is between $\frac{3.08\,\mathrm{s}}{0.331\,\mathrm{s}} = 9\,\mathrm{x}$ (SEPIC simulation) and $\frac{0.938\,\mathrm{s}}{0.0670\,\mathrm{s}} = 14\,\mathrm{x}$ (Cuk simulation) faster than MATLAB/Simulink and between $\frac{0.990\,\mathrm{s}}{0.0670\,\mathrm{s}} = 14\,\mathrm{x}$ (Cuk simulation) and $\frac{8.48\,\mathrm{s}}{0.331\,\mathrm{s}} = 25\,\mathrm{x}$ (SEPIC simulation) faster than NGSPICE thanks to its ideal switch models that result in piece-wise linear equations. In turn, our ISC MoC simulation is between $1.20\times$ and $2.67\times$ faster than PLECS: its switches are ideal and simulation profits from the faster numerical solution of the resulting linear state-space equations. In addition, it runs on top of SystemC via our direct CT/DE synchronization algorithm that prevents useless synchronization when communicating the DE switch controllers with the CT power converter models. In general, the more details in the CT models, the slower the simulation. The CT MoCs defined on top of our direct CT/DE synchronization algorithm can benefit from the speed-up of using abstractions suitable to the system level, e.g., ideal switches, and from the speed-up provided by direct synchronization.

It is also necessary to restate that the accuracy of the ISC models is appropriate for high-level simulations. The behavior from the simulation start time until the settling time of these systems, when they reach the steady state, is similar in all tools. As an example, Figure 5.8 which shows the simulated output voltage V_o and the L_1 inductor current i_{L1} for the SEPIC power converter simulation. Although NGSPICE and Matlab account for the rapid nonlinear behavior at the switching instants, this behavior is irrelevant in the longer periods of system simulations. The ideal switch and diode abstractions used in PLECS and our ISC MoC provide the required accuracy.

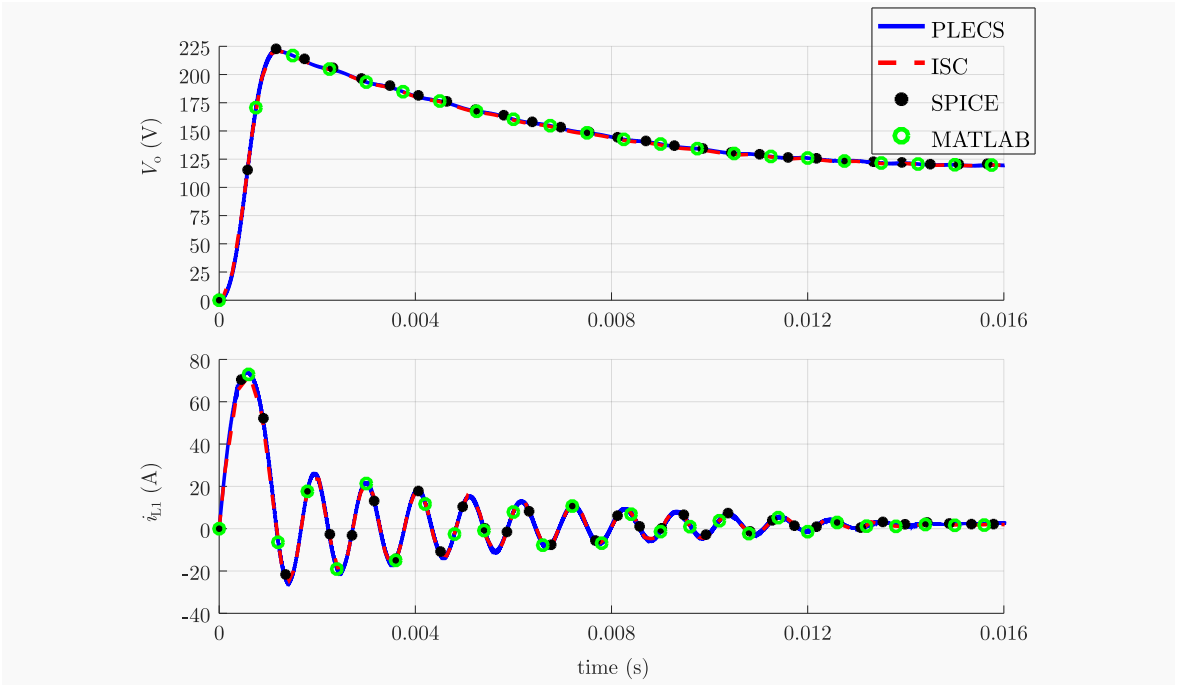


Figure 5.8: SEPIC converter output voltage V_o and inductor L_1 current i_{L1}

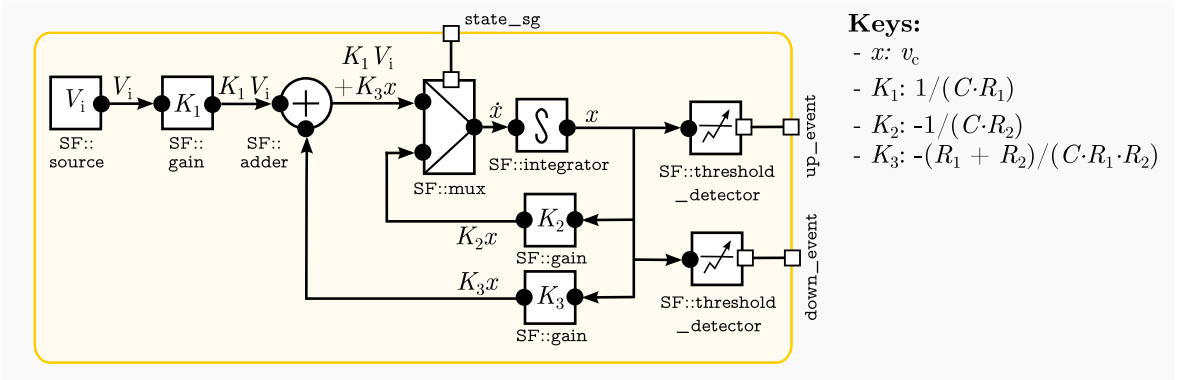


Figure 5.9: Equivalent SF model of the RC circuit presented in Section 4.4.3

As a last remark, we do not compare our results to SystemC AMS Electrical Linear Networks MoC because it does not provide a diode primitive necessary for modeling these systems. However, our results confirm SystemC AMS's insight on the interest of using CT models of computation on top of SystemC for the specification of CT models as part of larger CT/DE system models.

5.5.2.2 Cost of Elaboration and CT Interface Methods Execution

To be able to discuss the cost of the elaboration of ISC and SF models and the CT interface method execution, we present in Table 5.2 the simulation profiling data for the three power converter models in ISC and the SF model of the RC circuit shown in Figure 5.9 and described in Section 4.4.3. End-Of-Elaboration takes between 0.8% and 5.4% of the total wall-

Table 5.2: Percentage wall-clock time for different elaboration and simulation function calls

Function Call	Execution Cost (% of total wall-clock time)			
	SEPIC	**Boost**	**Cuk**	**SF RC Circuit**
End-Of-Elaboration	1.2	5.4	2.4	0.8
Get-Derivatives	49.9	54.9	53.7	63.8
Is-Event	10.5	11	9.9	32.3
Execute-Updates	33.2	24.6	28.9	0.7
Generate-Outputs	5.1	4.2	5.1	2.4
Total	**100**	**100**	**100**	**100**

clock execution time of the five studied methods. Although in this phase the ISC and SF MoCs transform the set of interconnected primitives to a mathematical representation and define the CT modeling interface methods required for direct CT/DE synchronization, these operations are executed only once, occupying a small portion of the total execution time. Among the four CT modeling interface methods, Get-Derivatives has the most impact, consuming around half of the execution time. This method is called several times along execution by the integration algorithm and may involve computational intensive operations: in the ISC MoC, for example, it executes matrix multiplications to get the derivative values. Execute-Updates follows, consuming around 30% of the execution time in the ISC MoC, which is explained by the cost of toggling switches and extracting the equations for the new topologies. It takes a negligible amount of time in the SF MoC example because the model does not include any DE integrator primitive whose state needs to be modified as a reaction to input events. Is-Event is the third by its impact on execution time in the ISC MoC taking around 10% and the second in the SF MoC: it is called several times during integration, one after each integration step. Last, Generate-Outputs is the method with the least impact: it is only executed at the CT/DE synchronization points and, in general, it is not as computationally intensive as calculating CT solutions.

From the above results, the CT MoCs simulated on top of our direct CT/DE synchronization algorithm must pay particular attention to the performance of their definition of these methods, and in particular to Get-Derivatives and Is-Event, as they are called with a very high frequency by the integration procedures. Otherwise, the speed-ups reached by high-level modeling and direct synchronization can be nullified.

5.6 Conclusions

In this chapter, we show that it is possible to simulate models from particular physical disciplines, such as electrical and abstract mathematical, on top of SystemC via our direct CT/DE synchronization algorithm. We implement the support for each physical discipline as a CT model of computation that defines the modeling elements to be used by the designer and the rules for elaborating these models into a simulatable representation. All CT models of computation meet a set of CT modeling requirements that take the form of a common

modeling interface. They comply with SystemC's modeling approach and diminish the CT modeling costs. In addition, they allow using the right high-level CT abstractions and a performant implementation to preserve the speed-ups of direct synchronization.

We draw inspiration from Ptolemy II and SystemC AMS to implement two CT models of computation: Signal Flow and Ideally Switched Circuits. They are similar to SystemC AMS Linear Signal Flow and Electrical Linear Networks, but they include event consumer and generator primitives for direct communication with other DE and CT models. They enable CT modeling in block diagram and electrical circuit forms, unlike Ptolemy II CT models, which can only take the form of block diagrams. Other CT models of computation can also be implemented just by specifying a set of modeling primitives and communication channels and ports, and by defining our CT modeling interface methods.

Our CT models of computation comply with SystemC's modeling approach. They serve to relieve the modeling cost that direct synchronization imposes on the designer.

We have experimentally studied the impact of the implementation of these models of computation on simulation speed by considering two factors: the CT level of abstraction and the definition of our CT modeling interface. First, we confirm that the more details in the models, the slower the simulation; it is important to implement the right CT abstractions to be able to attain the high speeds needed in system simulations, e.g., by using ideal vs. non-ideal and linear vs. nonlinear components. Second, we identify that the methods specifying the CT equations and the state conditions constitute execution hotspots as they are invoked frequently by the integration procedures; the implementation should beware of their computational performance. Using the right high-level CT abstractions and a performant implementation allow preserving the speed-ups of direct synchronization.

We believe that other specialized CT models of computation (mechanical, hydraulic, etc.) might as well be defined and benefit from these speed-ups.

Now that we have proposed a direct CT/DE synchronization algorithm in Chapter 4 and shown the implementation of different CT models of computation on top of it in this chapter, let us now explore the path of parallel simulation for simulation acceleration in Chapter 6.

Chapter 6: Parallel Direct Continuous-Time and Discrete-Event Synchronization

Contents

Ça pouvait être là, ou ailleurs... sur les quais, chez un bouquiniste, où une simple carte postale exotique pouvait faire naître des poésies ou des récits. Comme une discipline qui allie l'errance et la création — un cheminement sans but qui condense l'esprit

— Nicolas de Crécy, Visa transit volume 2

6.1 Introduction

In Chapter 4, we propose a sequential algorithm for direct CT/DE synchronization on top of SystemC. We implement it as a SystemC process that handles the simulation and synchronization of a CT module. This algorithm speeds up simulation when compared to fixed-step approaches. However, a CT/DE simulation containing multiple CT modules involves multiple synchronization processes, one per CT module. SystemC evaluates these processes sequentially, underusing parallel computational resources if available.

In this chapter, we propose an algorithm for parallel direct CT/DE synchronization based on our direct synchronization approach. Indeed, given that it allows the CT simulation to evolve over optimistic intervals whose size can be modified without affecting the simulation accuracy, it is reasonable to expect that, when multiple CT modules are present, we can make their integration intervals overlap and simulate them in parallel. We describe this algorithm in Section 6.2, and prove its causality, completeness, and liveness properties in Section 6.3. Then, in Section 6.4, we discuss typical challenges in parallel simulation, such as the preservation of causality, the prevention of race conditions, and the efficient use of parallel resources. In Section 6.5, we evaluate and discuss simulation speed-up based on a set of experiments. Finally, we present our conclusions in Section 6.6.

6.2 Algorithm

We now propose an algorithm to execute in parallel all CT modules in a combined CT/DE model and directly synchronize them with the DE SystemC simulation. We focus here only on CT parallel execution and make the only assumption that the DE part of the simulation follows SystemC semantics. To this aim, we use a unique SystemC process that is launched at `end_of_elaboration` and executed during simulation. The evaluation of this single process continues to be sequential from the DE simulator perspective but still parallel from the process internal perspective. This way, we avoid modifications to the standard SystemC simulator.

Algorithm 16 PARALLEL-SYNCHRONIZATION

```
1: while true do
2:     SELECT-MODULES()
3:     EXECUTE()
4:     SYNCHRONIZE-END-TIMES()
5:     SCHEDULE-REACTIVATION()
6: end while
```

Algorithm 16 gives the process' top-level algorithm. It consists of four phases that are represented graphically in Figure 6.1. The process is initially suspended and it can be reactivated in three ways: during SystemC's initialization phase, by self-reactivation, or because of the occurrence of an input event. At reactivation, SELECT-MODULES is the first phase, it selects the CT modules to run in parallel which are either all CT modules—during initialization or at a self-reactivation—or the ones sensitive to the input event; they will be simulated until

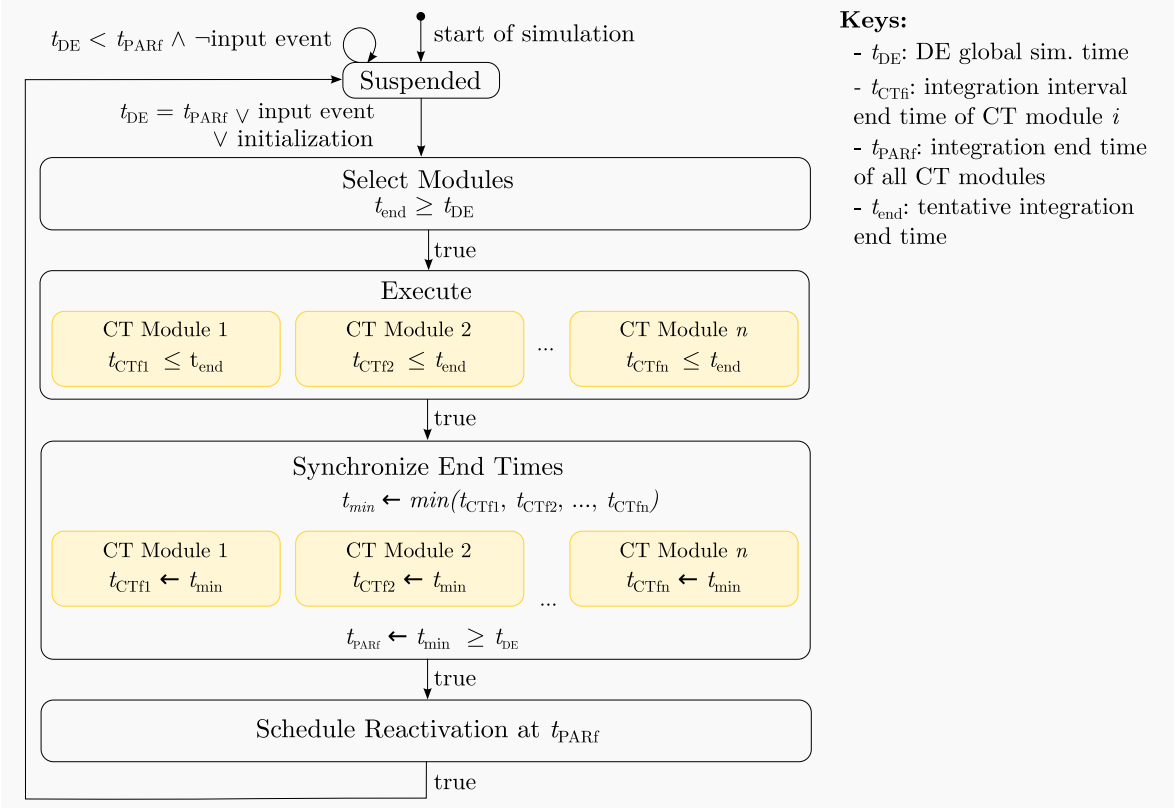


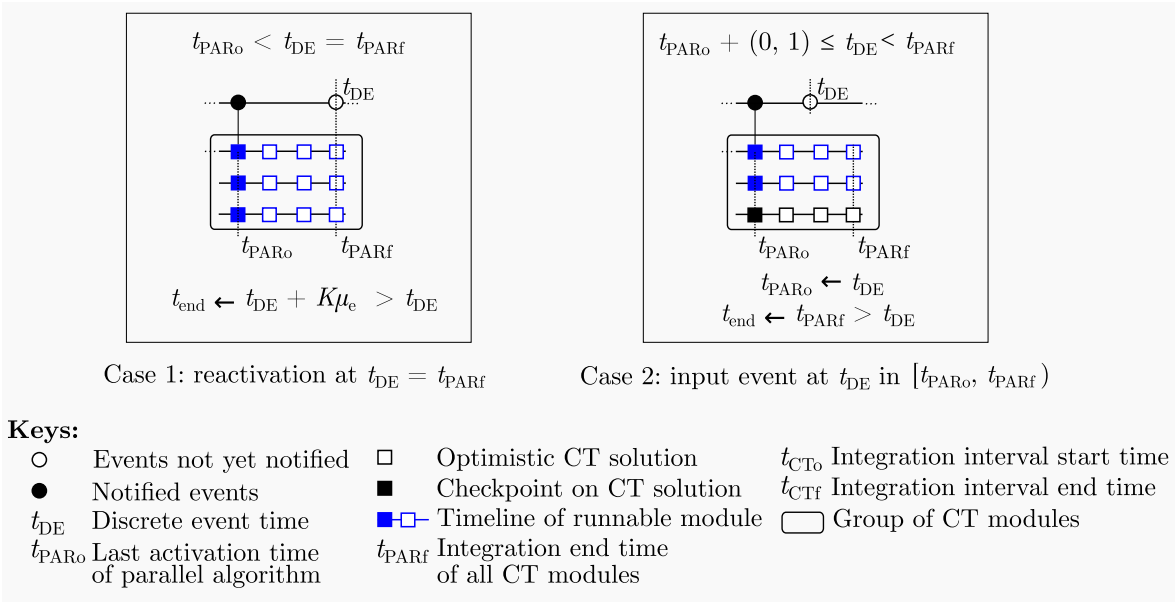
Figure 6.1: Parallel direct synchronization process overview

the tentative end of execution time t_{end}. EXECUTE is the second phase, it generates outputs for the selected modules and runs them in parallel from the current global simulation time (t_{DE}) to the tentative end of execution time ($t_{\text{CT}_\text{f}} = t_{\text{end}}$) or until they detect a state event ($t_{\text{CT}_\text{f}} < t_{\text{end}}$). SYNCHRONIZE-END-TIMES follows; if one or more CT modules detect a state event, their end of integration interval time will be less than the tentative end of execution time ($t_{\text{CT}_\text{f}} < t_{\text{end}}$); SYNCHRONIZE-END-TIMES finds the minimum integration interval end time ($t_{\text{min}} \leftarrow min(t_{\text{CT}_{\text{f1}}}, t_{\text{CT}_{\text{f2}}}, ..., t_{\text{CT}_{\text{fn}}})$, for modules 1, 2, ..., n) and synchronizes the CT modules to this time so that they can be executed in parallel at the next parallel reactivation time ($t_{\text{PAR}_\text{f}} \leftarrow t_{\text{min}}$). Finally, SCHEDULE-REACTIVATION (Algorithm 16, line 5) schedules the next reactivation at t_{PAR_f}. Let us explain each phase in detail.

6.2.1 Module Selection

The reactivation of PARALLEL-SYNCHRONIZATION occurs during the SystemC initialization phase, at a self-reactivation, or an input event. SELECT-MODULES is the first phase and its goal is to select the modules to run in parallel and add them to the RUNNABLE-MODULES set. These modules are either all CT modules—during initialization or at a self-reactivation—or those activated by the input event according to the following cases (Figure 6.2):

1. **Reactivation at** $t_{\text{DE}} = t_{\text{PAR}_\text{f}}$: suppose that the activation occurs either during initialization ($t_{\text{DE}} = t_{\text{PAR}_\text{f}} = (0, 0)$) or at self-reactivation at $t_{\text{DE}} = t_{\text{PAR}_\text{f}} > (0, 0)$. All



Case 1: reactivation at $t_{DE} = t_{PARf}$ Case 2: input event at t_{DE} in $[t_{PARo}, t_{PARf})$

Figure 6.2: Graphical representation of SELECT-MODULES

CT modules have initial conditions or solutions at this time—calculated during the previous execution and they can be executed over a new integration interval. To this end, SELECT-MODULES adds them to the RUNNABLE-MODULES set and assigns a new tentative end of execution time (t_{end}) to a value in the future following the adaptive strategy $t_{end} \leftarrow t_{DE} + K \cdot \mu_e$ discussed in Section 4.3.3—with the only difference that μ_e is now the average time between input events to all CT modules, we assume their activation frequencies to be similar. They are runnable and will execute until t_{end} or until they detect a state event.

2. **Reactivation by an input event at** $t_{DE} < t_{PAR_f}$: suppose that, during the previous execution, the modules had advanced their local times t_{CT_f} to $t_{PAR_f} > t_{DE}$ and that a reactivation had been scheduled at this time. However, an input event reactivates the process and the DE global time is only $t_{DE} < t_{PAR_f}$. The sensitive modules have to be executed to handle the event, which invalidates their optimistic solutions between t_{DE} and t_{PAR_f}. SELECT-MODULES calls GET-MODULES-ACTIVATED-BY-INPUT to get the sensitive modules and stores them in the RUNNABLE-MODULES set. GET-MODULES-ACTIVATED-BY-INPUT is a method that traverses the CT-MODULES set, stores, and returns the modules whose inputs have changed since the last activation. Then, SELECT-MODULES assigns the tentative end of execution time t_{end} to t_{PAR_f} to try to catch them up with the rest of modules that have already calculated solutions at this time. They are runnable and will execute until t_{end} or until they detect a state event.

In all cases, SELECT-MODULES stores the last activation time in t_{PAR_o}. At the end of this phase, the RUNNABLE-MODULES set contains the modules to execute from $t_{PAR_o} = t_{DE}$

to t_{end} or until the time of the next state event. Algorithm 17 summarizes this discussion. CT-Modules is the set of all CT modules in the model.

Algorithm 17 Select-Modules

```
1: if t_DE = t_PAR_f then                          // Input event at t_DE < t_PAR_f
2:     Runnable-Modules ← CT-Modules
3:     t_end ← t_DE + K · μ_e
4: else                                            // Input event at t_DE < t_PAR_f
5:     Runnable-Modules ← Get-Modules-Activated-By-Input()
6:     t_end ← t_PAR_f
7: end if
8: t_PAR_o ← t_DE
```

6.2.2 Parallel Execution

We base our discussion in this section directly on the algorithm rather than on an figure because the Execute phase invokes the Handle-Reactivation and Calculate-Solutions functions of our sequential algorithm and whose graphical representations we discuss in Section 4.3.1.1 and Section 4.3.1.2, respectively.

The goal of Execute (Algorithm 18) is to generate outputs for the runnable modules and to execute them in parallel. Outputs are generated only if $t_{\text{DE}} = t_{\text{PAR}_\text{f}}$ (Algorithm 18, lines 1–4) in order to preserve causality. They are generated sequentially to overcome the lack of protection to race conditions on regular SystemC channels. In general, writing output values is not a time-consuming operation when compared to the cost of calculating CT solutions. Then, Execute runs in parallel the modules in Runnable-Modules (Algorithm 18, lines 6–9): it invokes Handle-Reactivation (Algorithm 2) and Calculate-Solutions (Algorithm 4) for each module. It omits Schedule-Reactivation because scheduling is now the responsibility of the parallel synchronization algorithm and not of each single CT module, as we will explain in Section 6.2.4. Handle-Reactivation continues to restore the CT state and create checkpoints (see Section 4.3.1.1), but we have slightly modified it to omit output generation as it has already been done sequentially: Algorithm 4, line 13 is commented out. Calculate-Solutions continues to compute optimistic solutions and detect state events, but we have also slightly modified it to receive t_{end} to be used as the tentative integration interval end time: Algorithm 4, line 3 is replaced by $t_{\text{CT}_\text{f}} \leftarrow t_{\text{end}}$, t_{end} contains the tentative end of execution time for all CT modules.

All runnable modules are given the same integration interval end time t_{end}, but not all of them may advance until this time ($t_{\text{CT}_\text{f}} < t_{\text{end}}$ for some of them). During the calculation of solutions, as discussed in Section 4.3.1.2, some modules may detect a state event at t_{se} between t_{DE} and t_{end} (for module i, $t_{\text{CT}_\text{fi}} \leftarrow t_{\text{se}}$); others may detect a state event at t_{DE} (for module i, $t_{\text{CT}_\text{fi}} \leftarrow t_{\text{DE}} + (0, 1)$). The leftover modules may advance until t_{end}. In these situations, it is profitable to omit the solutions beyond the minimum state event time so that all modules are synchronized and can reactivate at the same future time or at the next

Algorithm 18 EXECUTE

```
1: if t_DE = t_PARf then                                   // Sequential output generation
2:     for all module ∈ RUNNABLE-MODULES do
3:         module.GENERATE-OUTPUTS(module.x, module.i_cp, t_DE)
4:     end for
5: end if
6: parfor module ∈ RUNNABLE-MODULES do                     // Parallel execution
7:     module.HANDLE-REACTIVATION()
8:     module.CALCULATE-SOLUTIONS(t_end)
9: end parfor
```

microstep to continue parallel execution over their overlapping integration intervals. This is our strategy to increase parallel resource usage.

6.2.3 Synchronization of Continuous-Time Module Execution

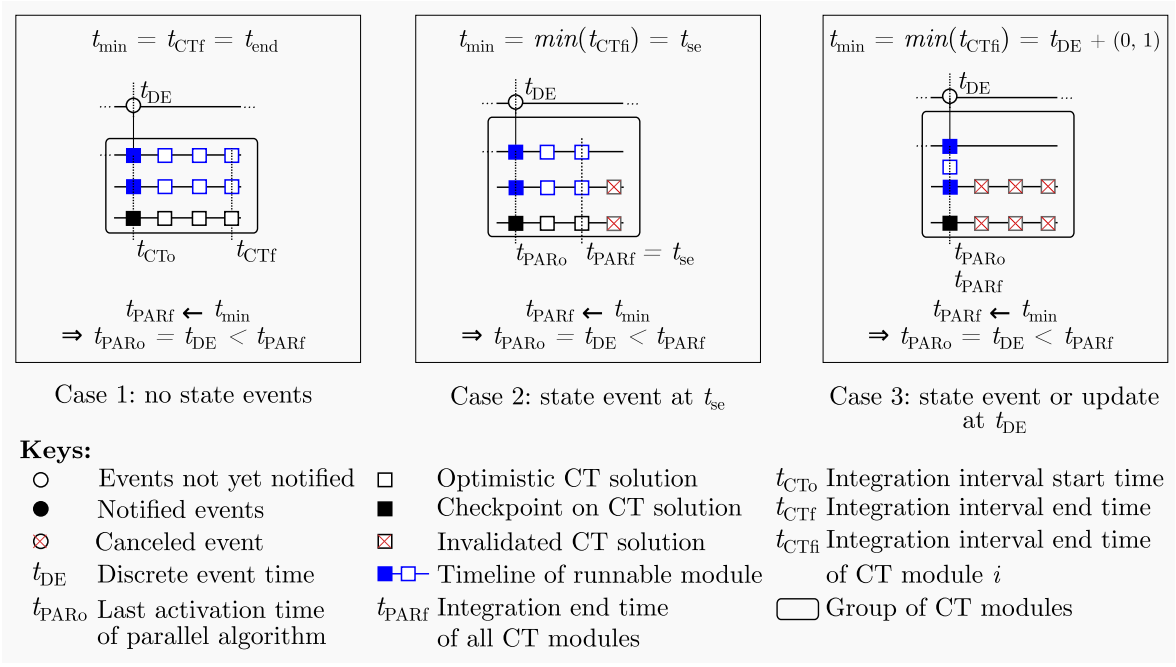


Figure 6.3: Graphical representation of SYNCHRONIZE-END-TIMES

SYNCHRONIZE-END-TIMES synchronizes the integration interval end time of all CT modules so that they can be reactivated at the same time and executed in parallel. This is a strategy to increase parallel resource usage to circumvent the problem found by the DE state-of-the-art solutions where there is a risk of having too few processes to evaluate. To this aim, SYNCHRONIZE-END-TIMES traverses the set of CT modules to get the minimum integration interval end time and stores it in t_{min}. Different values for t_{min} imply the different cases shown in Figure 6.3:

1. **No state events detected during execution**: $t_{min} = t_{CT_f} = t_{end}$ where t_{CT_f} is the

same for every module. The modules are already synchronized at t_{end}. The process sets the next parallel reactivation time t_{PAR_f} to $t_{\min}$.

2. **State event detected at t_{se} such that $t_{\text{DE}} < t_{\text{se}} < t_{\text{CT}_\text{f}}$**: the process stores the time of the state event with the earliest timestamp in $t_{\min}$. The modules that had calculated solutions beyond this time have to synchronize; they restore their last checkpoint and recompute solutions in parallel until $t_{\min}$. They are now synchronized at $t_{\min}$. The process sets t_{PAR_f} to $t_{\min} = t_{\text{se}}$ to schedule a reactivation in the future.

3. **State event or state/model update detected at t_{DE}**: $t_{\min}$ gets $t_{\text{DE}} + (0, 1)$. The modules that had calculated solutions beyond this time have to synchronize, so they restore their last checkpoint at $t_{\text{CT}_\text{o}} = t_{\text{DE}}$. They are now synchronized at $t_{\text{DE}} + (0, 1)$. The process sets t_{PAR_f} to $t_{\min} = t_{\text{DE}} + (0, 1)$ to schedule a reactivation at the next microstep.

At the end of Synchronize-Module-End-Time, $t_{\text{PAR}_\text{f}} \geq t_{\text{DE}} + (0, 1)$, which ensures simulation time progress in the abscense of infinite delta cycles (we assume that the designer avoids them during modeling).

Algorithm 19 Synchronize-End-Times

```
1: t_min ← t_end
2: for module ∈ CT-Modules do                          // Find minimum end time
3:     t_min ← Min(t_min, module.t_CTf)
4: end for
5: parfor module ∈ CT-Modules do          // Set end time for all CT modules to t_min
6:     module.Synchronize-Module-End-Time(t_min)
7: end parfor
8: t_PARf ← t_min                                      // Set end time for parallel CT group
```

Algorithm 20 Synchronize-Module-End-Time($t_{\min}$)

```
1: if t_CTf > t_min then
2:     x ← x_cp                                                  // Restore last checkpoint
3:     Integrate(t_CTo, t_end, x, i)                             // Integrate until t_min
4:     t_CTf ← t_min                                             // Set module end time
5: end if
```

Algorithm 19 summarizes the previous discussion. It traverses the CT-Modules set to find $t_{\min}$. Then, it invokes Synchronize-Module-End-Time (Algorithm 20) in parallel on each CT module passing $t_{\min}$. Synchronize-Module-End-Time (Section 6.2.3) is a method that synchronizes the module to the given time $t_{\min}$ by restoring the last checkpoint at t_{CT_o}, integrating the module's equation from this checkpoint to $t_{\min}$, and setting t_{CT_f} to $t_{\min}$. Notice that an optimized version would store the timestamped solution points during the previous execution in an auxiliary data structure to be able to restore the closest solution point to $t_{\min}$, thus reducing the number of additional integration steps to get the solution exactly at $t_{\min}$.

After all modules have synchronized their end of integration interval time, reactivation will be scheduled to continue parallel module execution.

6.2.4 Reactivation Scheduling

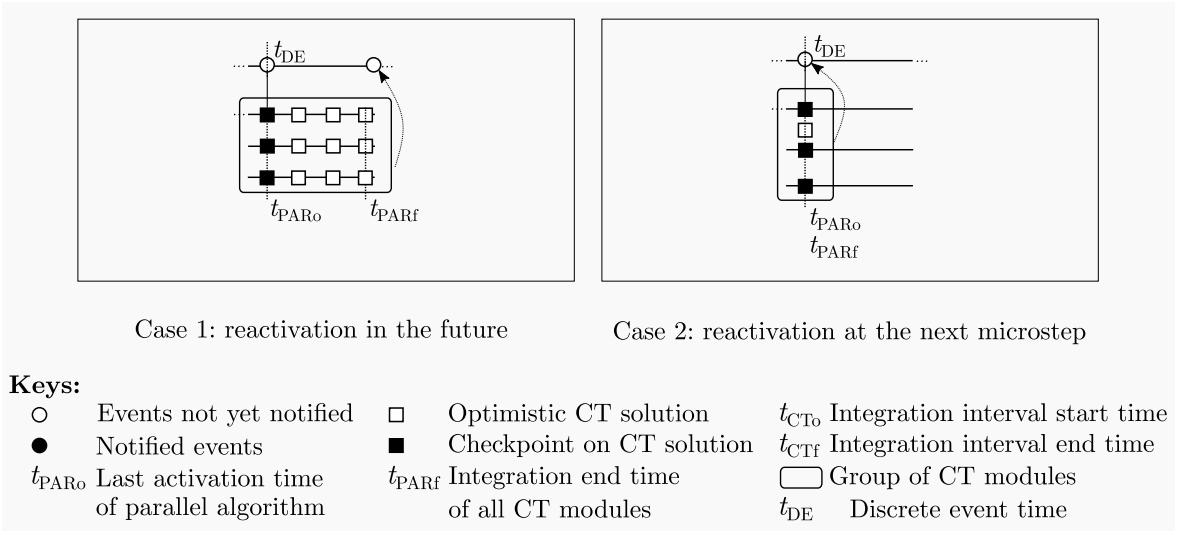


Figure 6.4: Graphical representation of SCHEDULE-REACTIVATION

The goal of SCHEDULE-REACTIVATION(Algorithm 21) is to schedule the next reactivation of the PARALLEL-SYNCHRONIZATION process at $t_{\mathrm{PAR_f}}$. Depending on this time, reactivation is scheduled either in the future (Figure 6.4, case 1) or at the next microstep (Figure 6.4, case 2). Its definition is similar to the one in Section 4.3.1.3, but uses instead $t_{\mathrm{PAR_f}}$ to which the CT modules are synchronized. The *input_events_list* contains all input events to all CT modules, which makes the top-level PARALLEL-SYNCHRONIZATION algorithm sensitive to them. Algorithm 21 summarizes this discussion.

Algorithm 21 SCHEDULE-REACTIVATION

1: WAIT(RE($t_{\mathrm{PAR_f}} - t_{\mathrm{DE}}$) **or** *input_events_list*)

6.3 Parallel Synchronization Algorithm Properties

Our parallel synchronization algorithm preserves the causality, liveness, and completeness properties of the sequential algorithm that we propose in Section 4.3. In particular:

6.3.1 Discussion on Causality and Completeness

Lemma 4. *At the end of each evaluation of PARALLEL-SYNCHRONIZATION (Algorithm 16),* $t_{\mathrm{PAR_f}} > t_{\mathrm{DE}}$.

Proof. The next parallel reactivation time $t_{\mathrm{PAR_f}}$ is only assigned at SYNCHRONIZE-END-TIMES (Algorithm 19, line 8) with the value of $t_{\min}$. In turn, $t_{\min}$ contains the minimum

integration interval end time of the CT modules (Algorithm 19, lines 2–4), which is a value between $t_{\mathrm{DE}} + (0, 1) > t_{\mathrm{DE}}$ and the tentative end of integration interval time t_{end} (Algorithm 18, line 8). Finally, t_{end} contains either $t_{\mathrm{DE}} + K \cdot \mu_{\mathrm{e}} > t_{\mathrm{DE}}$ (Algorithm 17, line 3), or $t_{\mathrm{PAR_f}} > t_{\mathrm{DE}}$ (Algorithm 17, line 6). These two values are both in the future w.r.t. t_{DE}, then $t_{\mathrm{PAR_f}} > t_{\mathrm{DE}}$ at the end of each evaluation of Parallel-Synchronization. □

Lemma 5. *At the end of each evaluation of Parallel-Synchronization (Algorithm 16), all CT modules are synchronized with solutions at* $t_{\mathrm{PAR_f}}$.

Proof. The next parallel reactivation time $t_{\mathrm{PAR_f}}$ is only assigned at Synchronize-End-Times (Algorithm 19, line 8) with the value of t_{min}, which contains the minimum integration interval end time of the CT modules (Algorithm 19, lines 2–4). The CT modules that had calculated solutions beyond this time restore their state to a previous checkpoint and recalculate solutions up until t_{min} (Algorithm 19, lines 5–7). No more solutions are calculated after this point in the algorithm, then all CT modules are synchronized with solutions at $t_{\mathrm{PAR_f}} = t_{\mathrm{min}}$ at the end of each evaluation of Parallel-Synchronization. □

Theorem 3. *Parallel-Synchronization (Algorithm 16) is causal, i.e., it does not generate events in the past w.r.t. the global simulation time* t_{DE}.

Proof. We need to show that (a) whenever it is executed, $t_{\mathrm{PAR_f}} \geq t_{\mathrm{DE}}$ and (b) generated outputs correspond to solutions at t_{DE}. At the end of each evaluation of Parallel-Synchronization, $t_{\mathrm{PAR_f}} > t_{\mathrm{DE}}$ (Lemma 4) and the process schedules a reactivation at this time (Algorithm 21); at the next reactivation, t_{DE} has advanced at most until $t_{\mathrm{PAR_f}}$—otherwise, the DE simulator would have skipped the reactivation event at $t_{\mathrm{PAR_f}}$. This implies that whenever the process is executed, $t_{\mathrm{PAR_f}} \geq t_{\mathrm{DE}}$, which verifies proposition (a). Proposition (b) is verified as follows: Generate-Outputs is invoked only if comparison $t_{\mathrm{DE}} = t_{\mathrm{PAR_f}}$ yields True (Algorithm 18, lines 1–5), time at which all CT modules are synchronized and have solutions (Lemma 5). From (a) and (b), outputs are generated only if $t_{\mathrm{DE}} = t_{\mathrm{PAR_f}}$ and they correspond to solutions at this time, then Parallel-Synchronization is causal. □

Theorem 4. *Parallel-Synchronization (Algorithm 16) is complete in the sense that it handles all possible cases of reactivation.*

Proof. We need to show that Parallel-Synchronization handles all the reactivation cases that we describe in Section 4.3.1.1. This can be verified as follows: the first phase executed by the process is Select-Modules (Algorithm 17) and it handles two cases. First, $t_{\mathrm{DE}} = t_{\mathrm{PAR_f}}$, which implies that either the activation occurs during initialization ($t_{\mathrm{DE}} = t_{\mathrm{PAR_f}} = (0, 0)$) or at a reactivation at $t_{\mathrm{DE}} = t_{\mathrm{PAR_f}}$, time at which all CT modules are synchronized. In both situations, they are all made runnable (Algorithm 17, line 2) and, as shown by Theorem 2 in Section 4.3.4, they are capable of handling the input events according to cases 1, 2, 3, and 6 of Handle-Reactivation (Algorithm 2). In the second case, $t_{\mathrm{DE}} < t_{\mathrm{PAR_f}}$, which implies that, during the previous execution, all CT modules had advanced their local times t_{CT} to $t_{\mathrm{PAR_f}}$ (Algorithm 19, lines 5–8) and a reactivation had been scheduled at this time

(Algorithm 21, line 1); but an input event occurs ($t_{\mathrm{DE}} < t_{\mathrm{PAR_f}}$) and the sensitive modules have to be executed to handle the event: these modules are made runnable (Algorithm 17, line 5) and, as shown by Theorem 2 in Section 4.3.4, they are capable of handling the input events according to cases 4 and 5 of HANDLE-REACTIVATION (Algorithm 2). Reactivation at $t_{\mathrm{DE}} < t_{\mathrm{PAR_o}}$ is not possible as it would imply that time goes backwards ($t_{\mathrm{PAR_o}}$ contains the t_{DE} of the previous activation, Algorithm 17, line 8). Reactivation at $t_{\mathrm{DE}} > t_{\mathrm{PAR_f}}$ is not possible as it would imply that the DE kernel skipped the reactivation event at $t_{\mathrm{PAR_f}}$. There are no more branches to explore, thus PARALLEL-SYNCHRONIZATION is complete. □

6.3.2 Discussion on Liveness

PARALLEL-SYNCHRONIZATION (Algorithm 16) does not stagnate simulation by itself: it schedules reactivations at $t_{\mathrm{PAR_f}}$ (Algorithm 21, line 1) which is either in the future w.r.t. t_{DE} or at the next delta cycle for a finite number of consecutive times (Lemma 4). The only way we can have an infinite amount of delta cycles that prevent simulation to progress is when one or several CT modules advance time to $t_{\mathrm{DE}} + (0, 1)$ at every execution. Such a situation would indicate the presence of a zero-delay loop in the model, which is an error that designers should avoid (as discussed in Section 4.3.4.2).

6.4 Challenges in Parallel Simulation

In Section 3.6, we have described some of the challenges that the state-of-the-art approaches face when trying to parallelize the simulation of DE SystemC models. They concern the preservation of causality, the prevention of race conditions, and the efficient use of parallel resources. In this section, we discuss how our algorithm overcomes these challenges and we give some additional constraints on our CT modeling approach.

6.4.1 Preservation of Causality

In DE parallel simulation, events could be generated and consumed out of order, possibly harming causality. We have proved the causality of our algorithm in Section 6.3, i.e., all CT module output events are generated at the current global simulation time. Notice also that, in our algorithm, parallel execution occurs only during the evaluation phase, similar to the DE parallelization approaches described in Section 3.6.1. The unmodified SystemC simulator continues to execute the update, delta notification, and time notification phases sequentially, without the risk of out-of-order event processing.

6.4.2 Prevention of Race Conditions

Race conditions on shared data are a common issue in parallel DE simulation on SystemC. They are present in different places, such as shared variables between processes, unprotected channels, direct access to memory locations (DMI protocol), and unprotected access to the simulator state. We circumvent these problems by modeling and simulation constraints.

First, to prevent race conditions on shared variables, the CT modules are not allowed to communicate by mechanisms other than regular SystemC channels, following a strict separation of concerns between computation and communication. This constraint is natural to our definition of CT module in Section 4.2. Their parallel simulation in EXECUTE (Algorithm 21) requires the invocation of HANDLE-REACTIVATION (Algorithm 2) and CALCULATE-SOLUTIONS (Algorithm 4). But the variables to which these methods have write access ($\boldsymbol{x}$, $\boldsymbol{x}_{\text{cp}}$, $\boldsymbol{i}_{\text{cp}}$, t_{CT_o}, t_{CT_f}, etc.) are maintained by each CT module separately. They are not shared and race conditions cannot occur. This analysis also applies to the parallel execution of SYNCHRONIZE-MODULE-END-TIME (Section 6.2.3). Special attention should be taken for the implementation of INTEGRATE (integration algorithm) and LOCATE (root location algorithm), which are directly or indirectly invoked by HANDLE-REACTIVATION, CALCULATE-SOLUTIONS and SYNCHRONIZE-MODULE-END-TIME. They must be thread-safe. In addition, GET-DERIVATIVES, IS-EVENT, and EXECUTE-UPDATES, defined by the designer or by the model of computation, are indirectly called during execution: they must also be thread-safe.

Second, our CT modules do not directly implement TLM protocols, but they can be connected via regular DE signals to DE modules implementing TLM. These DE modules act as intermediary connections between the CT modules and the rest of the virtual prototype. Handling DMI race conditions is delegated to the DE parallelization method, if necessary.

Third, in SystemC, writing values to channels is not thread-safe. As discussed in Section 6.2.2, our CT modules generate outputs sequentially, avoiding this type of race condition. We assume that the computation cost of GENERATE-OUTPUT is smaller than the cost of calculating solutions. Otherwise, thread-safe channels would have to be provided.

Last, simulator state access in SystemC is unprotected. Our algorithm accesses this state in two ways: read-only access to the global simulation time t_{DE} without risk of race conditions; and write access for scheduling via SystemC's `wait` function. As described in Section 6.2.4, our PARALLEL-SYNCHRONIZATION process handles scheduling for all CT modules by only one call to this function at each evaluation, there is no need to protect this single call. Notice that the CT models of computation implemented on top of this strategy should, then, be constrained to avoid direct write access to the simulator state.

6.4.3 Efficient Use of Parallel Resources

One issue with parallelizing only the process evaluation phase is that, for models at high levels of abstraction, the average number of runnable processes is low, which limits parallel resource usage. This would be particularly true for CT modules, as each independent module has its own CT dynamics and would set its own integration interval end time t_{CT_f} different from the other modules. To solve this problem, we manipulate t_{CT_f} to synchronize the reactivation of the CT modules, similar to the state-of-the-art approaches that set the same global quantum to all TLM models to increment the number of processes at each delta cycle (Section 3.6.4). We can make another analogy to the "tasks with duration" approach [107], in which the designer specifies a duration for the execution of the tasks associated with each discrete event

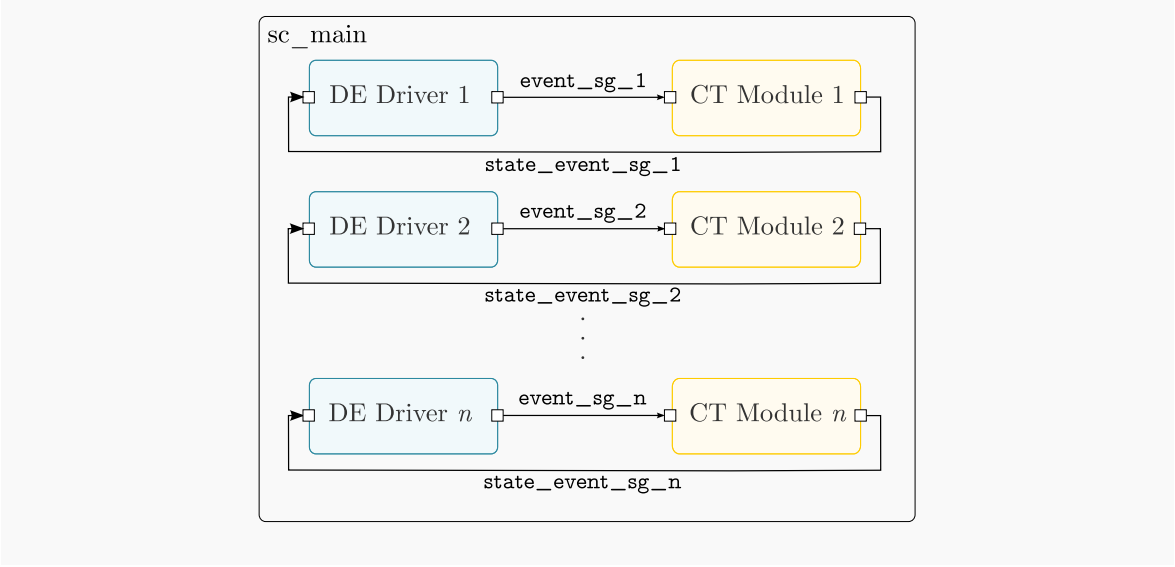


Figure 6.5: Synthetic model for parallel simulation experiments

and the simulator executes in parallel the tasks whose durations overlap. In CT modules, integration does occur over an interval or "duration". We synchronize $t_{\mathrm{CT_f}}$ so that their start times at the next reactivation overlap. We assume that the group of CT modules have a similar average time between events: synchronizing one CT module with a high density of events with other that has low density would slow down the execution of the second. However, we do not require these average times to be the same.

6.5 Experiments

In this section, we study our parallel direct CT/DE synchronization algorithm's speed-up taking as reference our direct synchronization algorithm proposed in Section 4.3. In particular, we study and discuss the effect on simulation speed of different factors such as the number of processing threads, the required CT simulation precision, the number of CT modules, and the complexity of the CT equations. Let us describe the model used for these experiments.

6.5.1 Model

We propose a synthetic model with a variable number of CT modules and a variable complexity of equations. Figure 6.5 shows this model: it contains n CT modules, module i receives a boolean event signal `event_sg_`i from a DE driver module i, selects between two different CT equations based on the signal value, and generates a boolean state event signal `state_event_sg_`i when its CT state crosses a given threshold ($\boldsymbol{x} \geq \boldsymbol{threshold}$, with $\boldsymbol{x} \in \mathbb{R}^m$ and $\boldsymbol{threshold} \in \mathbb{R}^m$ for $m \in \mathbb{N}$). DE driver i writes an event to signal `event_sg_`i with a mean time between events of μ and a standard deviation σ: not all drivers generate events at the same time so the CT modules do not have to reactivate at the same delta cycles.

The CT modules are of two types according to their complexity. The first type implements

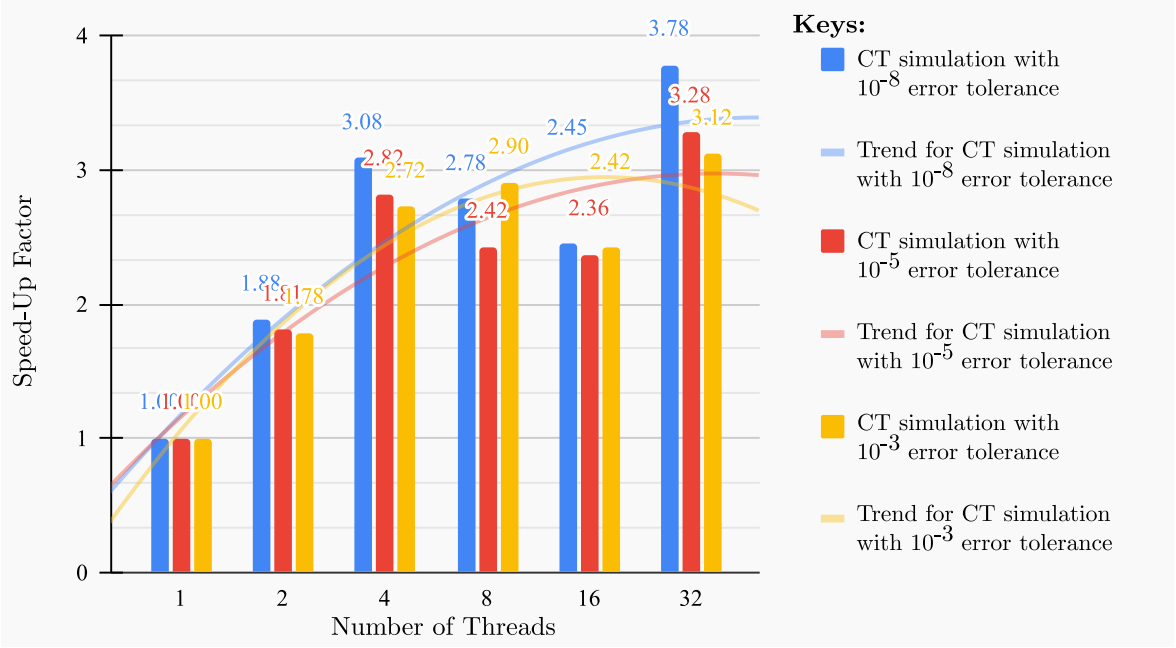


Figure 6.6: Parallel simulation speed-up vs. number of threads

the nonlinear Equation (6.1). The second type is simpler and implements the linear Equation (6.2). In these equations, $\boldsymbol{K}_1, \boldsymbol{K}_2, ..., \boldsymbol{K}_{11} \in \mathbb{R}^m$.

$$\dot{\boldsymbol{x}} = \begin{cases} \boldsymbol{K}_1 \cdot \boldsymbol{x} + \boldsymbol{K}_2 \cdot cos(t_{\text{CTf}}) + \boldsymbol{K}_3 \cdot sin(t_{\text{CTf}}), & \text{if } \texttt{event\_sg\_n} = \text{TRUE} \\ \boldsymbol{K}_4 \cdot \boldsymbol{x} + \boldsymbol{K}_5 \cdot cos(t_{\text{CTf}}) + \boldsymbol{K}_6 \cdot sin(t_{\text{CTf}}) + \boldsymbol{K}_7, & \text{otherwise} \end{cases} \tag{6.1}$$

$$\dot{\boldsymbol{x}} = \begin{cases} \boldsymbol{K}_8 \cdot \boldsymbol{x} + \boldsymbol{K}_9, & \text{if } \texttt{event\_sg\_n} = \text{TRUE} \\ \boldsymbol{K}_{10} \cdot \boldsymbol{x} + \boldsymbol{K}_{11}, & \text{otherwise} \end{cases} \tag{6.2}$$

6.5.2 Simulation Results and Discussion

For the simulation of this model, let us discuss the effect on speed-up of the number of processing threads and CT simulation precision, and the effect on speed-up of the number of CT modules and their complexity.

6.5.2.1 Speed-Up vs. Number of Threads and CT Simulation Precision

We run our model with $n = 128$ CT modules implementing Equation (6.1) on a machine with 32 hardware threads. Figure 6.6 shows the speed-up vs. the number of threads. We calculate the speed-up as $speedup = \frac{t_{\text{seq}}}{t_{\text{par}}}$, where t_{seq} is the simulation wall-clock time of our sequential synchronization algorithm presented in Chapter 4 and t_{par} is the simulation wall-clock time of our parallel synchronization algorithm presented in this chapter. We simulate the CT part with relative and absolute error tolerances of 10^{-8} (higher precision), 10^{-5}, and 10^{-3} (lower

Simulation Part	**CPU Utilization (%)**				
	Used Threads				**Total**
	1	**2**	**3**	**4**	
Parallel part of EXECUTE and SYNCHRONIZE-END-TIMES	2.9	8.8	25.3	53	90
Sequential SystemC	3.4				3.4
Spin and Overhead	-	-	-	-	6.6
All parts					100

Table 6.1: CPU utilization during simulation

precision). This figure shows that the higher the number of hardware threads, the higher the parallel simulation speed-up. It also shows that the higher the required precision when solving the CT equations, the higher the parallel simulation speed-up.

Speed-up increases with the number of hardware threads. For example, in the simulation with 10^{-8} error tolerance the speed-up goes from $1.88\times$ when using 2 hardware threads to $3.78\times$ when using 32 hardware threads. However, the threading efficiency ($\frac{speed-up}{number\ of\ threads}$) decreases: it goes from $\frac{speedup}{number\ of\ threads} = \frac{1.88}{2} = 0.94$ for 2 threads to $\frac{3.78}{32} = 0.12$ for 32 threads. In other words, with more threads, each thread executes useful work for a smaller fraction of time. One of the main reasons for this behavior is the overhead of launching, maintaining, and terminating the processing threads. Another reason is memory contention. To explain why, let us discuss the simulation profiling results in Table 6.1.

Table 6.1 shows the CPU utilization during simulation. The parallel portion of EXECUTE (Algorithm 18) and SYNCHRONIZE-END-TIMES (Algorithm 19), occupy 90% of the total CPU time, of which 2.9% executes exclusively in one thread, 8.8% in 2 threads, 25.3% in 3 threads, and the rest of the time in all 4 threads. If, in addition, we take into account the sequential DE SystemC simulation, and the spin and overhead time of the parallel implementation, then the effective CPU utilization is 2.69 out of 4 threads. This poor utilization is caused mostly by functions that allocate and deallocate memory, such as the C++ `new` operator, which is one of the most active functions as it is called from the numerical integration algorithms that create several temporary CT state instances during simulation. Memory contention reduces the effective amount of threads that can execute in parallel. A more performant implementation would pay attention to this technical problem.

Table 6.1 also shows that 6.6% of the total CPU utilization is spent in spinning and parallel execution overhead. It is reasonable to expect that this proportion increases with the number of threads, degrading efficiency. Notice also that, for this model, the sequential portion of the simulation accounts only for 3.4% of the total CPU utilization. We have designed our synthetic model this way to be able to study the CT parallel simulation speed-up. Real-life models would have a more important DE sequential portion; integrating our solution with the DE state-of-the-art approaches may reveal useful.

Finally, as Figure 6.6 also shows, the higher the required CT simulation precision, the higher the attained simulation speed-up. With more precision, more computation is required in the numerical integration algorithms, which increases the parallel portion of the simulation. With a greater parallel portion, comes greater speed-up for the same number of hardware

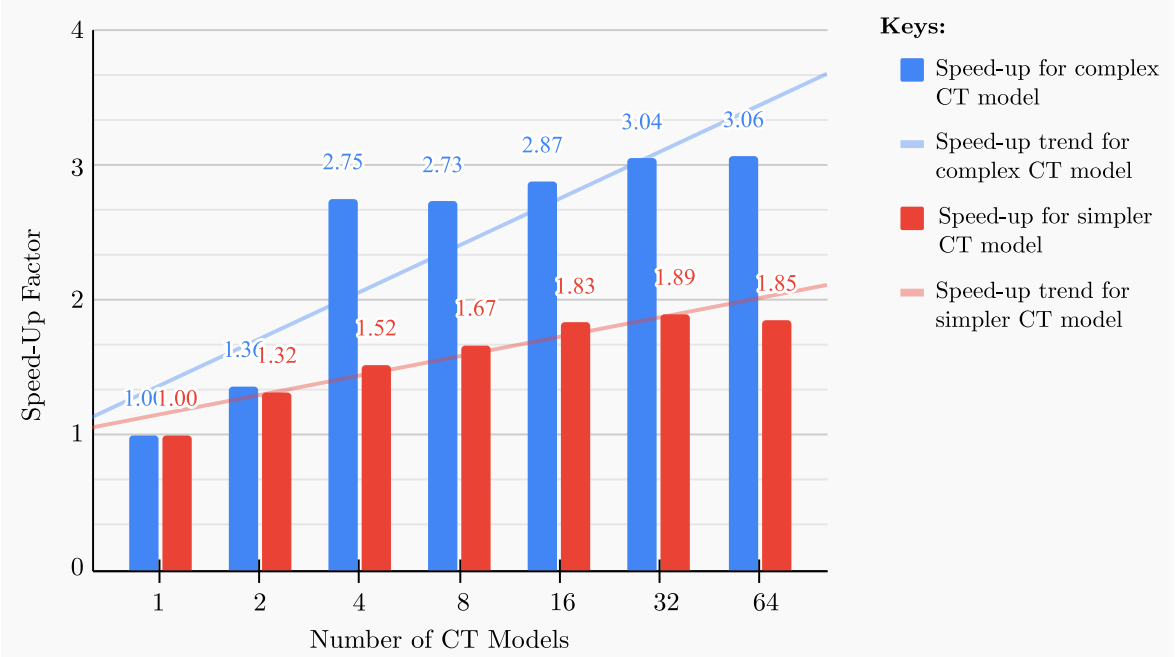


Figure 6.7: Parallel simulation speed-up vs. number of continuous-time modules

threads, according to Amdahl's law [130]. In a combined CT/DE simulation, it is worthy to execute in parallel the CT modules if the results are required to be at high precision.

6.5.2.2 Speed-Up vs. Number of Simulated CT Modules and their Complexity

We run our model with a variable number of CT modules on a machine with 4 hardware threads. Figure 6.6 shows the speed-up vs. the number of CT modules. We run the same experiments for the complex modules implementing Equation (6.1) and the simpler modules implementing Equation (6.2). This figure shows that the higher the number of CT modules, the higher the parallel simulation speed-up. It also shows that the more complex the CT equations, the higher the parallel simulation speed-up.

Speed-up increases with the number of modules. However, the speed-up increase with each additional CT module is smaller each time. For example, in the simulation of the complex CT modules, the speed-up increase is of $(2.75 - 1.36) \times = 1.39 \times$ when going from 2 to 4 CT modules but only of $(3.04 - 2.75) \times = 0.29 \times$ when going from 4 to 32 CT modules. We can not expect to reach the upper theoretical speed-up limit given by Amadahl's law for a fixed number of processors simply by adding more CT modules, as each module contributes not only to the parallel portion of the simulation but also to the sequential one (e.g., by sequential output generation in our algorithm). The speed-up increase for each additional CT module has diminishing returns.

Speed-up increases with the complexity of the equations. More computation is required when the equations are complex, which increases the portion of the simulation that can be simulated in parallel and its speed-up gain. It is worthy to use our parallel synchronization algorithm when the CT models are complex.

As a last remark, notice that the speed-ups in this section are reported w.r.t our sequential simulation algorithm that, in turn, already provides an important speed-up when compared to fixed-step approaches. Parallel direct synchronization further accelerates simulation.

6.6 Conclusions

In this chapter, we propose a parallel direct CT/DE synchronization algorithm to speed up the simulation of models involving multiple CT modules. Our solution is based on a single synchronization process that groups these components, synchronizes their execution, and simulates them in parallel. Although research on parallel DE simulation is rich, its ideas have not been applied yet to combined CT/DE simulation. Inspired by these approaches, our solution faces challenges such as the preservation of causality, the prevention of race conditions, and the efficient use of parallel resources.

Research on parallel simulation of DE models in SystemC is rich and has already shown its benefits for high-level simulation acceleration. Parallel CT/DE simulation faces similar challenges. One of them is the preservation of causality. Following the approaches in Section 3.6.1, we circumvent this problem by executing in parallel only SystemC's evaluation phase: the delta notification and timed notification phases remain sequential and, as a consequence, in time order. In addition, our CT modules calculate tentative solutions in the future, but they are constrained to generate output events only when these solutions have been confirmed, thus preventing processing events that have later to be canceled.

Another challenge is race condition prevention. The DE approaches handle race conditions from shared variables either by restricting modeling, by analyzing dependencies during compilation, or by dynamically monitoring memory accesses to detect and correct conflicts. We choose the way of restricting modeling as our definition of CT module in Chapter 4 already enforces separate CT states and variables and exclusive communication by events. To handle race conditions, our solution writes sequentially to channels, delegates DMI support to the DE simulation approach, and schedules events in the unprotected SystemC simulator only from a single process.

A third challenge concerns the use of parallel resources. The CT modules probably reactivate at different times, limiting the use of parallel resources and the achievable acceleration. State-of-the-art DE approaches either relax time constraints (allowing for small timing errors) or synchronize the execution quantum of TLM models or run the DE models over durations to increase the number of processes to be executed. Our solution modifies the CT integration intervals to make them overlap, this way the number of modules to simulate in parallel at each activation is increased.

We have experimentally studied and compared the simulation accuracy and speed of our proposed parallel direct synchronization algorithm to our sequential direct approach of Chapter 4. We show that it further speeds up the simulation. We confirm the interest in parallel simulation, especially when high CT simulation precision is required or the model involves a big number of CT modules with complex equations.

Chapter 7: Conclusions

Contents

Ce n'est pas le fruit de l'expérience, mais l'expérience elle-même qui est le but
— Walter Pater, La renaissance

7.1 Conclusions

In this thesis, our purpose is to accelerate the simulation of CT/DE models to support system design use cases that demand high simulation accuracy and speed. Our solution lets the CT models directly interact with the DE domain via events. Based on such interactions, we propose a CT/DE synchronization algorithm that reaches high simulation accuracy and speed, a CT modeling interface that can be implemented by different CT models of computation to simulate models from different physical disciplines, and a parallel synchronization algorithm that provides a higher speed-up. These contributions constitute our answers to the three research questions that we exposed in Section 2.7, more specifically:

1. *What CT/DE synchronization strategies other than the intermediary discrete-time representations apply to high-level simulation and how can they be exploited to accelerate simulation?*

   The CT and DE simulations can be synchronized at their actual interaction event times. Such a strategy maintains high accuracy in the results by communicating the relevant features in the original CT and DE signals without delay. It speeds up simulation by cutting the cost of unnecessary synchronization. As it does not depend on any predefined step but on the actual interaction times, direct synchronization breaks the accuracy/speed trade-off of fixed-step approaches.

   Direct synchronization on top of SystemC enables high-level modeling and simulation use cases such as CT/DE virtual prototyping. The closest SystemC extension serving this use case is SystemC AMS, but it implements only fixed-step synchronization. Other state-of-the-art simulation languages and tools, such as VHDL-AMS and Ptolemy II, implement direct synchronization, but they either do not target high levels of abstraction, or their strategy is limited by the next event time. Different from all state-of-the-art approaches, our solution lets the CT simulation execute over optimistic intervals whose size dynamically adapts to the frequency of input events, diminishing the constraints on the numerical integration steps and narrowing the roll-back costs. Our solution collects event frequency information to speed up the simulation.

   Direct synchronization requires a superdense time model to ensure the simultaneity, ordering, and causality of the events; our solution interprets SystemC time as a superdense time to meet this requirement. Direct synchronization also requires modeling support to the direct interactions; our solution gathers the modeling requirements in a set of CT interface methods. The manual definition of such methods imposes an additional modeling effort to the designer, but this effort can be relieved by the implementation of CT models of computation that come to solve, in addition, our second research question.

2. *How to support the simulation of models from particular physical disciplines in the form of electrical circuits and other graphical representations while still using a generic*

CT/DE simulation engine?

Components from particular physical disciplines can be modeled via particular CT models of computation; they can execute on the same simulation engine by complying with a common CT modeling interface. A model of computation defines domain-specific modeling elements, elaboration, and simulation operations. The modeling interface enables support to the direct interactions and consists of a set of methods that represent the discrete event changes in the CT model, state, and state conditions, and the generation of state events. These methods are automatically defined by the model of computation during elaboration and called by our direct synchronization algorithm during simulation. They separate modeling and elaboration, which are specific to each CT model of computation, from simulation, which is common to all of them.

Inspired from SystemC AMS and Ptolemy II, we implement the Signal Flow and Ideally Switched Circuits CT models of computation. Their modeling primitives can be grouped as pure computational, event consumers, and event generators, and serve to specify models in block diagram and electrical circuit forms. They reduce the modeling effort by automatically defining the interface methods. In addition, they favor simulation speed-up in three ways: by implementing the right CT modeling abstractions, e.g., using ideal vs. non-ideal and linear vs. nonlinear components; by ensuring a performant implementation; and by relying on our direct synchronization algorithm.

Although this solution reaches high speeds, it still executes sequentially. The way of parallel execution for simulation acceleration is still free, which leads us to our third research question.

3. *How to apply parallel simulation algorithms to accelerate CT/DE simulation?*

   CT/DE models containing multiple CT modules that communicate by direct interactions can be simulated in parallel by a single SystemC process that groups them all, handles their joint consumption and generation of events, synchronizes their activation times, and evaluates them in parallel.

   Although research on parallel DE simulation is rich, no state-of-the-art approach deals with the parallel execution of high-level CT/DE models on top of SystemC. Similar to these approaches, our solution faces problems related to preserving causality, preventing race conditions, and improving parallel execution resource usage. First, our algorithm executes in parallel only the evaluation phase, SystemC's delta and time notification phases remain sequential, preserving causality; in addition, our algorithm generates output events only when the optimistically computed CT solutions have been confirmed so that they can not later be invalidated.

   Second, this algorithm prevents race conditions between processes based on our definition of a CT module: the state of a CT module can only be changed by other modules by events, thus ensuring that no two processes access this state simultaneously. To prevent race conditions in unprotected channels, it generates outputs sequentially, which does

not harm speed as this operation consumes only a small portion of time when compared to computing CT solutions. Finally, to prevent race conditions in the simulator state access during scheduling, this algorithm delegates the operation to a single process.

And third, inspired by the "tasks with duration" approach from parallel DE simulation, our algorithm improves resource usage by synchronizing the local times so that all CT modules can be reactivated at the same simulation time points and execute over the same duration or interval.

This algorithm constraints the CT models of computation to define thread-safe CT interface methods.

This solution speeds up simulation when compared to our sequential direct synchronization algorithm, which already provides an important speed-up w.r.t. fixed-step approaches. Parallel direct synchronization further accelerates simulation.

In brief, on the basis of direct CT/DE interactions, we propose a direct synchronization algorithm that provides high accuracy and speed, a CT interface that can be defined by different CT models of computations to support the simulation of models from different physical disciplines and benefit from these speed-ups, and a parallel synchronization algorithm that provides even higher speed-ups. Combined CT/DE modeling and simulation with high accuracy and speed is an aid to the design of high confidence and performant complex cyber-physical systems with short design cycles and reduced costs.

7.2 Future Works

Research in this area can deepen in the following directions:

- *Direct synchronization and CT roll-back*: our direct synchronization algorithm adapts the CT integration interval sizes to the time between input events. This approach does not avoid CT roll-backs when the interval size is larger than the time to the next CT input event. It is of interest to avoid or reduce the CT roll-back costs. One possible solution is presented in [62] in the context of VHDL AMS: the models are analyzed before simulation to identify the input events capable of affecting the CT part, their timestamps are used to set the integration interval size, thus avoiding roll-back. However, this approach does not always work since, over the course of the simulation, additional CT input events with earlier timestamps can be generated. Besides, the standard SystemC simulator neither allows identifying such events nor provides functions to obtain the time to a particular scheduled event. Further research to implement this and other strategies to prevent roll-backs with as few modifications to the SystemC simulator as possible is required.

- *CT models of computation*: we have implemented a Signal Flow and a Ideally Switched Circuits model of computation. Their primitives are at high level of abstraction to

support the high simulation speeds required by complex cyber-physical system design. But these systems often involve components from other physical disciplines, such as mechanical and chemical. To support their modeling and simulation, we consider it worthy to investigate the high-level abstractions in these disciplines and implement suitable high-level models of computation. One challenge is to provide primitives that abstract away behaviors that may be too detailed and too slow to simulate, e.g., primitives that replace fast physical phenomena with discrete events. Another challenge is to implement appropriate algorithms to transform the models into CT equations that are easy to simulate, e.g., piece-wise linear vs. nonlinear equations. This way, such models of computation can simulate on top of our direct synchronization algorithm without the risk of slowing it down.

- *Parallel DE simulation integration*: our parallel direct CT/DE synchronization algorithm handles the parallel simulation and synchronization of the CT modules in a CT/DE model. It works well when the CT simulation cost is important. To accelerate simulations where the CT and DE simulation cost is more balanced, it could be valuable to integrate our solution with one of the state-of-the-art DE parallel simulation approaches; TLM approaches would be preferred given that we assume our modules to be part of system models.

- *Improvements to our parallel simulation solution*: our parallel simulation solution assumes that the CT modules reactivate with similar frequencies. A cyber-physical system may include components with very different CT dynamics and time constants. Future studies could address the parallel simulation of CT modules with very different reactivation frequencies, possibly by grouping only the modules whose dynamics are similar. An online clustering algorithm may be useful for this aim.

Appendix A: CT Modeling Code Examples

In this section, we illustrate our proposed CT modeling with the code for the Bouncing Ball model described in Section 4.4.2.

We begin with the definition of the class BallCt in Listing 2 that models the CT dynamics of the bouncing ball. This class inherits from the `sct_core::ct_module` class that we provide as base class for all CT modules (line 1). It declares two standard SystemC input signal ports of type `bool` that receive the bounce and stop input events (lines 7–8). It declares two standard SystemC output signal ports of type `bool`, that output the collision and low energy events (lines 11–12). It has a constructor (lines 18–20) that takes in the module's name (`name`), the boolean parameter `use_adaptive`that allows the selection of between the adaptive (TRUE) and fixed (FALSE) Δt strategies, the boolean parameter `avoid_rollback` that allows using the time of the next discrete event (TRUE) or not (FALSE), and the real parameter `interval_length` that gives an initial guess for the value of Δt if the adaptive strategy is used or its value for the whole simulation if the fixed strategy is used; these parameters are used by the base class `sct_core::ct_module` to properly configure the synchronization process. It declares the four basic CT module functions GET-DERIVATIVES, IS-EVENT, EXECUTE-UPDATES, and GENERATE-OUTPUTS(lines 26–30), whose definition we described in the paragraphs that follow. Finally, it declares a set of private variables and utility functions (lines 32–42) used in the differential equations, state conditions, rules to update the state, and output generation functions.

Listing 3 shows the implementation of GET-DERIVATIVES for this model. It takes in, the value of the state (`x`), a non-const reference to the derivative values (`dxdt`), and the current CT simulation time (`t`). Notice that, in this practical implementation, it is not necessary to pass the DE input values as arguments because the CT module has access to them and to their checkpoints via the auxiliary object `ode_system::inputs` provided by the `sct_core::ct_module` class. This object automatically manages the creation of checkpoints on the DE inputs during simulation and allows accessing the input values by using the `operator[]` (line 4). The method selects between two sets of equations based on the `stopped` value from the DE inputs (line 7), and sets the derivative values accordingly (lines 8–9 and 13–14).

Listing 4 shows the implementation of IS-EVENT for this model. It takes in the state (`x`) and time (`t`) and returns a boolean indicating the occurrence of a state event. The method

```
class BallCt : public sct_core::ct_module {
    public:
        ////////////////////////////////////////////////////////////
        // PORTS
        ////////////////////////////////////////////////////////////
        // DE Inputs
        sc_core::sc_in<bool> bounce_in;         // Bounce in event
        sc_core::sc_in<bool> stop_in;           // Stop event

        // DE Outputs
        sc_core::sc_out<bool> collision_out;    // Collision event
        sc_core::sc_out<bool> low_energy_out;   // Low energy event


        ////////////////////////////////////////////////////////////
        // Constructor
        ////////////////////////////////////////////////////////////
        BallCt(sc_core::sc_module_name name,
                bool use_adaptive = true, bool avoid_rollback = true,
                double interval_length = DELTA_T_BALL_MODEL);


        ////////////////////////////////////////////////////////////
        // CT MODEL FUNCTIONS
        ////////////////////////////////////////////////////////////
        void get_derivatives(const sct_core::ct_state &x, sct_core::ct_state &dxdt,
                             sct_core::ct_time t);
        bool is_event(const sct_core::ct_state &x, sct_core::ct_time t);
        bool execute_updates(sct_core::ct_state &x, sct_core::ct_time t);
        void generate_outputs(const sct_core::ct_state &x, sct_core::ct_time t);

    private:
        // System parameters
        double g;                       // Gravity constant
        double min_x_threshold;         // Position threshold for bouncing
        double min_v_threshold;         // Speed threshold for low energy
        double elasticity_factor;       // Elasticity factor when bouncing

        // Other parameters and utility methods
        bool last_bounce_in_value;      // Auxiliary variable
        bool BallCt::is_collision_event(const sct_core::ct_state &x)
        bool BallCt::is_low_energy_event(const sct_core::ct_state &x);
};
```

Listing 2: Bouncing Ball Model: Class Definition

```
1  void BallCt::get_derivatives(const sct_core::ct_state &x, sct_core::ct_state &dxdt,
2                               sct_core::ct_time t) {
3      // Get the value from the stop_in port
4      bool stopped = sct_core::ode_system::inputs[stop_in];
5
6      // Equations when bouncing
7      if (!stopped) {
8          dxdt[0] = x[1];
9          dxdt[1] = -g;
10     }
11     // Equations when stopped, all derivatives to zero.
12     else {
13         dxdt[0] = 0;
14         dxdt[1] = 0;
15     }
16 }
```

Listing 3: Bouncing Ball Model: Get-Derivatives

```
1  bool BallCt::is_event(const sct_core::ct_state &x, sct_core::ct_time t){
2      bool events = false; // Return variable
3      bool stopped = ode_system::inputs[stop_in]; // Get DE input value
4      // Test state conditions if ball is bouncing
5      if (!stopped) {
6          events = is_collision_event() || is_low_energy_event();
7      }
8      return events;
9  }
10 // Predicate that tells the occurrence of a collision event
11 bool BallCt::is_collision_event(const sct_core::ct_state &x) {
12     return  x[0] <= min_x_threshold && x[1] < 0;
13 }
14
15 // Predicate that tells the occurrence of a low energy event
16 bool BallCt::is_low_energy_event(const sct_core::ct_state &x) {
17     return x[0] <= min_x_threshold && fabs(x[1]) < min_v_threshold;
18 }
```

Listing 4: Bouncing Ball Model: Is-Event

reads the `stopped` value from the DE inputs (line 3), if the ball is bouncing, it tests the `is_collision_event` and `is_low_energy` event predicates (line 5–7), and returns the result (line 8). The `is_collision_event` predicate is a utility method that returns true if the ball position is close to zero and if its speed is negative (line 11). The `is_low_energy_event` predicate is a utility method that returns true if the ball position is close to zero and if its speed is also close to zero (line 14). Notice that, in this method, the CT module has access to the DE inputs via the auxiliary object `ode_system::inputs`.

Listing 5 shows the implementation of Execute-Updates for this model. It takes in the state (`x`) and time (`t`) and returns a boolean indicating that the state has been updated (lines 5 and 7). At each bounce or transition in the boolean value of the `bounce_in` signal (line 2), it instantaneously changes the ball speed as a reaction (line 4). Notice that, in this method, the CT module has access to the DE inputs via the auxiliary object `ode_system::inputs`.

```
1  bool BallCt::execute_updates(sct_core::ct_state &x, sct_core::ct_time t) {
2      if(ode_system::inputs[bounce_in] != last_bounce_in_value){
3          last_bounce_in_value = ode_system::inputs[bounce_in];
4          x[1] = -elasticity_factor*x[1]; // Discontinuity in ball speed
5          return true;                    // Indicate discontinuity
6      }
7      return false;                       // Indicate no discontinuity
8  }
```

Listing 5: Bouncing Ball Model: EXECUTE-UPDATES

```
1  void BallCt::generate_outputs(const sct_core::ct_state &x, sct_core::ct_time t) {
2      if (is_low_energy_event(x)) {
3          low_energy_out.write(true);                 // Generate low energy event
4      }
5      if (is_collision_event(x)) {
6          collision_out.write(!collision_out.read()); // Generate collision event
7      }
8  }
```

Listing 6: Bouncing Ball Model: GENERATE-OUTPUTS

Listing 6 shows the implementation of GENERATE-OUTPUTS for this model. It takes in the state (`x`) and time (`t`) and generates the DE outputs. It writes TRUE to the signal connected to the `low_energy_out` port if there is a low energy event (line 3). It writes the boolean complement of the signal connected to the `collision_out` port to the same signal if there is a collision event (line 3). Notice that, in this method, the CT module has access to the DE inputs via the auxiliary object `ode_system::inputs`.

Notice that we provide all classes in the namescape `sct_core` such as `sct_core::ct_module` (base class for CT modeling), `sct_core::ct_state` (class representing a CT space as a vector of `double` values), and `sct_core::ct_time` (class with a useful representation of time for the CT time domain) as part of our simulator. We provide the complete ready-to-run proof-of-concept implementation implementations of all models and algorithms discussed in this thesis as open-source in the repository https://gricad-gitlab.univ-grenoble-alpes.fr/tima/sls/projects/systemc-ct .

Publications

Journals

- Fernandez-Mesa, B.J., Andrade, L., and Pétrot, F. *Synchronization of Continuous Time and Discrete Events Simulation in SystemC.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 40(7), pp. 1450-1463. Available at: `https://www.doi.org/10.1109/TCAD.2020.3019204`

International Conferences

- Fernandez-Mesa, B.J., Andrade, L., and Pétrot, F. *Simulation of Ideally Switched Circuits in SystemC.* Asia and South Pacific Design Automation Conference (ASP-DAC 2021), Jan 2021, Tokyo, Japan. Available at: `https://www.doi.org/10.1145/3394885.3431417`

- Fernandez-Mesa, B.J., Andrade, L., and Pétrot, F. *Accurate and Efficient Continuous Time and Discrete Events Simulation in SystemC.* 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 2020, pp. 370-375. Available at: `https://www.doi.org/10.23919/DATE48585.2020.9116300`

- Fernandez-Mesa, B.J., Andrade, L., and Pétrot, F. *Electronic System Level Design of Heterogeneous Systems: a Motor Speed Control System Case Study.* 2019 17th IEEE International New Circuits and Systems Conference (NEWCAS), Munich, Germany, 2019, pp. 1-4. Available at: `https://www.doi.org/10.1109/NEWCAS44328.2019.8961289`

List of Acronyms

AMS Analog and Mixed-Signal

CT Continuous-Time

DAE Differential Algebraic Equation

DE Discrete-Event

DEVS Discrete-Event System Specification

DMI Direct Memory Interface

DT Discrete-Time

DTDF Dynamic Timed Data Flow

ELN Electrical Linear Networks

ESL Electronic System Level

FMI Functional Mockup Interface

FPGA Field-Programmable Gate Array

HLA High Level Architecture

ISC Ideally Switched Circuits

LSF Linear Signal Flow

MoC Model of Computation

ODE Ordinary Differential Equation

PDAE Partial Differential-Algebraic Equations

PDE Partial Differential Equations

QSS Quantized State Systems

RTL Register-Transfer Level

SEPIC Single-Ended Primary-Inductor Converter

SF Signal Flow

TDF Timed Data Flow

TLM Transaction Level Modeling

Bibliography

[1] The Clemson University Vehicular Electronics Laboratory, "Clemson Vehicular Electronics Laboratory: Electronic Transmission Control." [Online]. Available: `https://cecas.clemson.edu/cvel/auto/systems/transmission_control.html`, 2021. (visited on 2021/05/31). xiii, 2

[2] CSS Electronics, "CAN Bus Explained - A Simple Intro (2021)." [Online]. Available: `https://www.csselectronics.com/screen/page/simple-intro-to-can-bus/language/en`, 2021. (visited on 2021/05/31). xiii, 2

[3] A. Hemani, "Charting the EDA Roadmap," *IEEE Circuits and Devices Magazine*, vol. 20, no. 6, pp. 5–10, 2004. xiii, 8

[4] R. L. Burden and J. D. Faires, "Initial Value Problems for Ordinary Differential Equations," in *Numerical Analysis*, Brooks/Cole, 2018. xiii, 14

[5] X. Liyi, L. Bin, Y. Yizheng, H. Guoyong, G. Jinjun, and Z. Peng, "A Mixed-Signal Simulator for VHDL-AMS," in *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, pp. 287–292, 2001. xiii, 25, 28

[6] L. Xiao, Y. Ye, and B. Li, "A New Synchronization Algorithm for VHDL-AMS Simulation," *Journal of Computer Science and Technology*, vol. 17, no. 1, pp. 28–37, 2002. xiii, 28, 29

[7] H. Ghasemi and Z. Navabi, "A New Synchronization Algorithm for (VHDL-AMS) Mixed Signal Simulation," in *IEEE International Symposium on Communications and Information Technology, 2004. ISCIT 2004.*, vol. 2, pp. 1078–1083, IEEE, 2004. xiii, 28, 29

[8] E. A. Lee, "Constructive Models of Discrete and Continuous Physical Phenomena," *IEEE Access*, vol. 2, pp. 797–821, 2014. xiii, 10, 21, 22, 30, 72, 87

[9] D. C. Black, J. Donovan, B. Bunton, and A. Keist, *SystemC: From the Ground Up*, vol. 71. Springer Science & Business Media, 2009. xiii, 38

[10] A. Massarini and U. Reggiani, "An Efficient Algorithm for the Formulation of State Equations and Output Equations for Networks with Ideal Switches," in *ISCAS*

2001. The 2001 IEEE International Symposium on Circuits and Systems (Cat. No. 01CH37196), vol. 3, pp. 521–524, IEEE, 2001. xiv, 90, 92, 93, 95

[11] Cyber-Physical Systems | NIST, "Cyber-Physical Systems | NIST." [Online]. Available: `https://www.nist.gov/el/cyber-physical-systems`, 2021. (visited on 2021/06/01). 2

[12] Cyber-Physical Systems - a Concept Map, "Cyber-Physical Systems - a Concept Map." [Online]. Available: `https://ptolemy.berkeley.edu/projects/cps/`, 2019. (visited on 2019/10/21). 2

[13] "IEEE Standard for VHDL Language Reference Manual," *IEEE Std 1076-2019*, pp. 1–673, 2019. 2, 3, 24

[14] "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language," *IEEE STD 1800-2009*, pp. 1–1285, 2009. 2, 3

[15] IEEE Computer Society, *1666-2011 IEEE Standard for Standard SystemC Language Reference Manual*. IEEE, 2012. 2, 3, 36, 37, 39, 50, 60, 64, 66, 68, 88

[16] MathWorks, "MATLAB Documentation." [Online]. Available: `https://fr.mathworks.com/help/matlab/index.html`, 2021. (visited on 2021/03/30). 3, 35

[17] Wolfram, "Wolfram Mathematica." [Online]. Available: `https://www.wolfram.com/mathematica/`, 2021. (visited on 2021/05/31). 3

[18] D. Broman, C. Brooks, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter, "Determinate Composition of FMUs for Co-Simulation," in *Proceedings of the Eleventh ACM International Conference on Embedded Software*, p. 2, IEEE Press, 2013. 3

[19] J. S. Dahmann and K. L. Morse, "High Level Architecture for Simulation: An Update," in *Proceedings. 2nd International Workshop on Distributed Interactive Simulation and Real-Time Applications (Cat. No. 98EX191)*, pp. 32–40, IEEE, 1998. 3

[20] "IEEE Standard VHDL Analog and Mixed-Signal Extensions," *IEEE Std 1076.1-2017 (Revision of IEEE Std 1076.1-2007)*, pp. 1–672, 2018. 3, 25

[21] "Verilog-AMS Language Reference Manual," *Accellera Systems Initiative*, 2014. 3, 29

[22] F. Pêcheux, C. Lallement, and A. Vachoux, "VHDL-AMS and Verilog-AMS as Alternative Hardware Description Languages for Efficient Modeling of Multidiscipline Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 2, pp. 204–225, 2005. 3, 29

[23] IEEE Computer Society, *1666.1-2016 - IEEE Standard for Standard SystemC(R) Analog/Mixed-Signal Extensions Language Reference Manual*. IEEE, 2012. 3, 39, 41, 84

[24] J. Lygeros, C. Tomlin, and S. Sastry, "Hybrid Systems: Modeling, Analysis and Control," *Electronic Research Laboratory, University of California, Berkeley, CA, Tech. Rep. UCB/ERL M*, vol. 99, 2008. 6, 24

[25] F. Pêcheux, C. Grimm, T. Maehne, M. Barnasconi, and K. Einwich, "SystemC AMS Based Frameworks for Virtual Prototyping of Heterogeneous Systems," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–4, IEEE, 2018. 6, 39

[26] M. Barnasconi, "SystemC AMS Extensions: Solving the Need for Speed," *DAC Knowledge center*, 2010. 6

[27] A. Vachoux, C. Grimm, and K. Einwich, "SystemC-AMS Requirements, Design Objectives and Rationale," in *2003 Design, Automation & Test in Europe (DATE)*, pp. 388–393, IEEE, 2003. 6

[28] S. Adhikari, M. Barnasconi, M. Damm, K. Einwich, P. Floyd, D. Genius, C. Grimm, M.-M. Louërat, T. Mähne, F. Pêcheux, *et al.*, "SystemC Analog/Mixed-Signal User's Guide-User Perspective on IEEE Std. 1666.1-2016," *Accellera Systems Initiative*, 2020. 6

[29] P. J. Mosterman, "Hybrid Dynamic Systems: Modeling and Execution," 2007. 6

[30] S. Abdi, J. Peng, H. Yu, D. Shin, A. Gerstlauer, R. Doemer, and D. Gajski, "System-On-Chip Environment. SCE Version 2.2.0 Beta. Tutorial. UC Irvine.." [Online]. Available: `http://www1.ece.neu.edu/~specc/sce/sce-tutorial/html/AR.html`, 2021. (visited on 2021/03/29). 6

[31] "Verification, Validation, Testing of ASIC and SOC designs – What are the differences?.." [Online]. Available: `http://verificationexcellence.in/verification-validation-testing-soc/`, 2021. (visited on 2021/03/29). 7

[32] M. Barnasconi, M. Dietrich, K. Einwich, T. Vörtler, J.-P. Chaput, M.-M. Louërat, F. Pêcheux, Z. Wang, P. Cuenot, I. Neumann, *et al.*, "UVM-SystemC-AMS Framework for System-Level Verification and Validation of Automotive Use Cases," *IEEE Design & test*, vol. 32, no. 6, pp. 76–86, 2015. 7

[33] B. P. Zeigler, A. Muzy, and E. Kofman, "Hierarchy of System Specifications," in *Theory of Modeling and Simulation: Discrete Event & Iterative System Computational Foundations*, Academic Press, 2018. 7

[34] K. Ogata, *System Dynamics*, vol. 13. Pearson/Prentice Hall Upper Saddle River, NJ, 2004. 10, 11

[35] M. Barnasconi, K. Einwich, C. Grimm, T. Maehne, and A. Vachoux, "Advancing the SystemC Analog/Mixed-Signal (AMS) Extensions," *Open SystemC Initiative*, 2011. 10, 40

[36] G. F. Franklin, J. D. Powell, and A. Emami-Naeini, *Feedback Control of Dynamic Systems*. Pearson London, 2015. 10, 11

[37] F. E. Cellier and E. Kofman, *Continuous System Simulation*. Springer Science & Business Media, 2006. 11, 12

[38] A. Agarwal and J. Lang, *Foundations of Analog and Digital Electronic Circuits*. Elsevier, 2005. 12

[39] F. E. Cellier and J. Greifeneder, *Continuous System Modeling*. Springer Science & Business Media, 2013. 13

[40] S. J. Mason, "Feedback Theory-Some Properties of Signal Flow Graphs," *Proceedings of the IRE*, vol. 41, no. 9, pp. 1144–1156, 1953. 13

[41] J. Liu, "Continuous Time and Mixed-Signal Simulation in Ptolemy II," tech. rep., Dept. of EECS, University of California, Berkeley, CA, 1998. 14, 64, 66, 86

[42] R. L. Burden and J. D. Faires, *Numerical Analysis*. Belmont: Brooks/Cole, 2011. 14, 15, 59

[43] B. Fernández-Mesa, L. Andrade, and F. Pétrot, "Electronic System Level Design of Heterogeneous Systems: a Motor Speed Control System Case Study," in *2019 17th IEEE International New Circuits and Systems Conference (NEWCAS)*, pp. 1–4, IEEE, 2019. 17

[44] R. M. Fujimoto, "Parallel Discrete Event Simulation," *Communications of the ACM*, vol. 33, no. 10, pp. 30–53, 1990. 17, 42, 43

[45] P. Amodio and L. Brugnano, "Parallel Solution in Time of ODEs: Some Achievements and Perspectives," *Applied Numerical Mathematics*, vol. 59, no. 3-4, pp. 424–435, 2009. 17

[46] A. A. B. Pritsker and N. R. Hurst, "GASP IV: a Combined Continuous-Discrete FORTRAN-Based Simulation Language," *Simulation*, vol. 21, no. 3, pp. 65–70, 1973. 20, 21

[47] F. E. Cellier, "Combined Continuous/Discrete Simulation: Applications, Techniques and Tools," in *Proceedings of the 18th Conference on Winter Simulation*, pp. 24–33, 1986. 20, 22

[48] O. Maler, Z. Manna, and A. Pnueli, "From Timed to Hybrid Systems," in *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*, pp. 447–484, Springer, 1991. 20, 23

[49] P. J. Mosterman, "An Overview of Hybrid Simulation Phenomena and their Support by Simulation Packages," in *International Workshop on Hybrid Systems: Computation and Control*, pp. 165–177, Springer, 1999. 20, 21, 22

[50] B. P. Zeigler, "Embedding DEV&DESS in DEVS," in *DEVS Integrative Modeling & Simulation Symposium*, vol. 7, p. 18, 2006. 20, 22, 34

[51] B. P. Zeigler, A. Muzy, and E. Kofman, *Theory of Modeling and Simulation: Discrete Event & Iterative System Computational Foundations*. Academic Press, 2018. 22, 24, 34

[52] A. D. Ames, A. Abate, and S. Sastry, "Sufficient Conditions for the Existence of Zeno Behavior," in *Proceedings of the 44th IEEE Conference on Decision and Control*, pp. 696–701, IEEE, 2005. 23, 67

[53] Z. Manna and A. Pnueli, "Verifying Hybrid Systems," in *Hybrid Systems*, pp. 4–35, Springer, 1992. 23

[54] X. Liu, E. Matsikoudis, and E. A. Lee, "Modeling Timed Concurrent Systems," in *International Conference on Concurrency Theory*, pp. 1–15, Springer, 2006. 23

[55] F. Cremona, M. Lohstroh, D. Broman, E. A. Lee, M. Masin, and S. Tripakis, "Hybrid Co-Simulation: It's About Time," *Software & Systems Modeling*, vol. 18, no. 3, pp. 1655–1679, 2019. 23, 24

[56] D. Broman, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter, "Requirements for Hybrid Cosimulation Standards," in *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, pp. 179–188, ACM, 2015. 23

[57] C. Ptolemaeus, *System Design, Modeling, and Simulation: using Ptolemy II*, vol. 1. Ptolemy. org Berkeley, 2014. 24, 30

[58] J. Nutaro, "Toward a Theory of Superdense Time in Simulation Models," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 30, no. 3, pp. 1–13, 2020. 24

[59] E. Christen and K. Bakalar, "VHDL-AMS A Hardware Description Language for Analog and Mixed-Signal Applications," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 46, no. 10, pp. 1263–1272, 1999. 25, 26, 27

[60] W. L. Lynch Jr, "VHDL (VHSIC (Very High Speed Integrated Circuits) Hardware Descriptive Languages) Prototype Simulator.," tech. rep., Air Force Institute of Technology, Wright-Patterson Air Force Base School of Engineering, 1986. 26

[61] M. Zwolinski, C. Garagate, Z. Mrcarica, T. Kazmierski, and A. Brown, "Anatomy of a Simulation Backplane," *IEEE Proceedings-Computers and Digital Techniques*, vol. 142, no. 6, pp. 377–385, 1995. 28

[62] H. R. Ghasemi and Z. Navabi, "An Effective VHDL-AMS Simulation Algorithm with Event Partitioning," in *18th International Conference on VLSI Design held jointly with*

4th International Conference on Embedded Systems Design, pp. 762–767, IEEE, 2005. 28, 122

[63] E. A. Lee and H. Zheng, "Leveraging Synchronous Language Principles for Heterogeneous Modeling and Design of Embedded Systems," in *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pp. 114–123, 2007. 32

[64] J. Liu, "Continuous Time and Mixed-Signal Simulation in Ptolemy II," in *Dept. of EECS, University of California, Berkeley, CA*, Citeseer, 1998. 32, 84

[65] E. Kofman, M. Lapadula, and E. Pagliero, "PowerDEVS: A DEVS-Based Environment for Hybrid System Modeling and Simulation," *School of Electronic Engineering, Universidad Nacional de Rosario, Tech. Rep. LSD0306*, pp. 1–25, 2003. 34

[66] J. Nutaro, "ADEVS." [Online]. Available: `https://web.ornl.gov/~nutarojj/adevs/`, 2021. (visited on 2021/03/30). 34

[67] J. J. Nutaro, *Building Software for Simulation: Theory and Algorithms, with Application in C++*. Wiley Online Library, 2011. 34

[68] H. Praehofer, "System Theoretic Formalisms for Combined Discrete-Continuous System Simulation," *International Journal of General System*, vol. 19, no. 3, pp. 226–240, 1991. 34

[69] E. Kofman and S. Junco, "Quantized-State Systems: a DEVS Approach for Continuous System Simulation," *Transactions of The Society for Modeling and Simulation International*, vol. 18, no. 3, pp. 123–132, 2001. 35

[70] A. Benveniste, T. Bourke, B. Caillaud, J.-L. Colaço, C. Pasteur, and M. Pouzet, "Building a Hybrid Systems Modeler on Synchronous Languages Principles," *Proceedings of the IEEE*, vol. 106, no. 9, pp. 1568–1592, 2018. 35

[71] A. Benveniste, T. Bourke, B. Caillaud, and M. Pouzet, "Non-Standard Semantics of Hybrid Systems Modelers," *Journal of Computer and System Sciences*, vol. 78, no. 3, pp. 877–910, 2012. 35

[72] A. Benveniste, B. Caillaud, H. Elmqvist, K. Ghorbal, M. Otter, and M. Pouzet, "Structural Analysis of Multi-Mode DAE Systems," in *Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control*, pp. 253–263, 2017. 35

[73] MathWorks, "Numerical Integration and Differential Equations." [Online]. Available: `https://fr.mathworks.com/help/matlab/numerical-integration-and-differential-equations.html`, 2021. (visited on 2021/03/30). 35

[74] MathWorks, "Simulation Phases in Dynamic Systems." [Online]. Available: `https://fr.mathworks.com/help/stateflow/chart-programming.html`, 2021. (visited on 2021/03/30). 35

[75] X. Pan, C. Zivkovic, and C. Grimm, "Virtual Prototyping of Heterogeneous Automotive Applications: Matlab, SystemC, or both?," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, pp. 544–549, ACM, 2019. 36

[76] H. Al-Junaid, T. Kazmierski, and L. Wang, "SystemC-A Modeling of an Automotive Seating Vibration Isolation System," 2006. 39

[77] L. L. Andrade Porras, *Principles and implementation of a generic synchronization interface between SystemC AMS models of computation for the virtual prototyping of multi-disciplinary systems.* Theses, Université Pierre et Marie Curie - Paris VI, Jan. 2016. 39, 41

[78] G. Biagetti, M. Caldari, M. Conti, and S. Orcioni, "Extending systemc to analog modeling and simulation," in *Languages for System Specification*, pp. 229–242, Springer, 2004. 39

[79] G. Biagetti, M. Giammarini, M. Ballicchia, M. Conti, and S. Orcioni, "SystemC-WMS: Wave Mixed Signal Simulator for nonlinear Heterogeneous Systems," *International Journal of Embedded Systems*, vol. 6, no. 4, pp. 277–288, 2014. 39

[80] F. Herrera and E. Villar, "A Framework for Heterogeneous Specification and Design of Electronic Embedded Systems in SystemC," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 12, no. 3, pp. 1–31, 2008. 39

[81] S. H. A. Niaki, M. K. Jakobsen, T. Sulonen, and I. Sander, "Formal Heterogeneous System Modeling with SystemC," in *Proceeding of the 2012 Forum on Specification and Design Languages*, pp. 160–167, IEEE, 2012. 39, 46

[82] H. D. Patel and S. K. Shukla, "Towards a Heterogeneous Simulation Kernel for System-Level Models: a SystemC Kernel for Synchronous Data Flow Models," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 8, pp. 1261–1271, 2005. 39

[83] J. Zhu, I. Sander, and A. Jantsch, "HetMoC: Heterogeneous Modelling in SystemC," in *2010 Forum on Specification & Design Languages (FDL 2010)*, IET, 2010. 39

[84] T. J. Kazmierski and H. J. Aljunaid, "Synchronization of Analogue and Digital Solvers in Mixed-Signal Simulation on a SystemC Platform," 2003. 39

[85] H. Al-Junaid and T. Kazmierski, "An Extension to SystemC to allow Modeling of Analogue and Mixed Signal Systems at Different Abstraction Levels," 2004. 39

[86] M. Damm, C. Grimm, J. Haas, A. Herrholz, and W. Nebel, "Connecting SystemC-AMS mMdels with OSCI TLM 2.0 Models Using Temporal Decoupling," in *2008 Forum on Specification, Verification and Design Languages*, pp. 25–30, IEEE, 2008. 40

[87] L. Andrade, T. Maehne, A. Vachoux, C. B. Aoun, F. Pêcheux, and M.-M. Louërat, "Pre-Simulation Symbolic Analysis of Synchronization Issues between Discrete Event and Timed Data Flow Models of Computation," in *2015 Design, Automation & Test in Europe (DATE)*, pp. 1671–1676, IEEE, 2015. 41

[88] C. B. Aoun, L. Andrade, T. Maehne, F. Pêcheux, M.-M. Louërat, and A. Vachoux, "Pre-Simulation Elaboration of Heterogeneous Systems: The SystemC Multi-Disciplinary Virtual Prototyping Approach," in *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pp. 278–285, IEEE, 2015. 41

[89] F. Bouchhima, M. Briere, G. Nicolescu, M. Abid, and E. Aboulhamid, "A SystemC/Simulink Co-Simulation Framework for Continuous/Discrete-Events Simulation," in *2006 IEEE International Behavioral Modeling and Simulation Workshop*, IEEE, 2006. 41

[90] R. Dömer, "Seven Obstacles in the Way of Standard-Compliant Parallel SystemC Simulation," *IEEE Embedded Systems Letters*, vol. 8, no. 4, pp. 81–84, 2016. 42

[91] R. Dömer, W. Chen, X. Han, and A. Gerstlauer, "Multi-Core Parallel Simulation of System-Level Description Languages," in *16th Asia and South Pacific Design Automation Conference (ASP-DAC 2011)*, pp. 311–316, IEEE, 2011. 42, 43

[92] D. Becker, M. Moy, and J. Cornet, "Parallel Simulation of Loosely Timed SystemC/TLM Programs: Challenges Raised by an Industrial Case Study," *Electronics*, vol. 5, no. 2, p. 22, 2016. 43, 45

[93] B. Chopard, P. Combes, and J. Zory, "A Conservative Approach to SystemC Parallelization," in *International Conference on Computational Science*, pp. 653–660, Springer, 2006. 43

[94] P. Ezudheen, P. Chandran, J. Chandra, B. P. Simon, and D. Ravi, "Parallelizing SystemC Kernel for Fast Hardware Simulation on SMP Machines," in *2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, pp. 80–87, IEEE, 2009. 43

[95] C. Schumacher, R. Leupers, D. Petras, and A. Hoffmann, "ParSC: Synchronous Parallel Systemc Simulation on Multi-Core Host Architectures," in *2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pp. 241–246, IEEE, 2010. 43

[96] L. Ainey, A. Efrati, and S. Weiss, "Parallel Cycle-Accurate SystemC Kernel," in *2014 IEEE 28th Convention of Electrical & Electronics Engineers in Israel (IEEEI)*, pp. 1–5, IEEE, 2014. 43

[97] W. Chen and R. Dömer, "Optimized Out-of-Order Parallel Discrete Event Simulation using Predictions," in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 3–8, IEEE, 2013. 43

[98] Z. Cheng, E. Arasteh, and R. Dömer, "Event Delivery using Prediction for Faster Parallel SystemC Simulation," in *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 357–362, IEEE, 2020. 43

[99] C. Roth, S. Reder, H. Bucher, O. Sander, and J. Becker, "Adaptive Algorithm and Tool Flow for Accelerating Systemc on Many-Core Architectures," in *2014 17th Euromicro Conference on Digital System Design*, pp. 137–145, IEEE, 2014. 43

[100] E. Viaud, F. Pêcheux, and A. Greiner, "An Efficient TLM/T Modeling and Simulation Environment Based on Conservative Parallel Discrete Event Principles," in *Proceedings of the Design Automation & Test in Europe Conference*, vol. 1, pp. 1–6, IEEE, 2006. 44

[101] A. Mello, I. Maia, A. Greiner, and F. Pecheux, "Parallel Simulation of SystemC TLM 2.0 Compliant MPSoC on SMP Workstations," in *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pp. 606–609, IEEE, 2010. 44, 45

[102] I. M. Pessoa, A. Mello, A. Greiner, and F. Pêcheux, "Parallel TLM Simulation of MPSoC on SMP Workstations: Influence of Communication Locality," in *2010 International Conference on Microelectronics*, pp. 359–362, IEEE, 2010. 44, 45

[103] C. Schumacher, J. H. Weinstock, R. Leupers, G. Ascheid, L. Tosoratto, A. Lonardo, D. Petras, and T. Grötker, "LegaSCi: Legacy SystemC Model Integration into Parallel SystemC Simulators," in *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pp. 2188–2193, IEEE, 2013. 44

[104] J. H. Weinstock, C. Schumacher, R. Leupers, G. Ascheid, and L. Tosoratto, "Time-Decoupled Parallel SystemC Simulation," in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–4, IEEE, 2014. 44

[105] J. H. Weinstock, R. Leupers, and G. Ascheid, "Parallel SystemC Simulation for ESL Design Using Flexible Time Decoupling," in *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pp. 378–383, IEEE, 2015. 44

[106] J. H. Weinstock, R. Leupers, G. Ascheid, D. Petras, and A. Hoffmann, "SystemC-Link: Parallel SystemC Simulation using Time-Decoupled Segments," in *2016 Design,*

Automation & Test in Europe Conference & Exhibition (DATE), pp. 493–498, IEEE, 2016. 44

[107] M. Moy, "Parallel Programming with SystemC for Loosely Timed Models: A Non-Intrusive Approach," in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 9–14, IEEE, 2013. 45, 113

[108] G. Funchal and M. Moy, "jTLM: an Experimentation Framework for the Simulation of Transaction-Level Models of Systems-on-Chip," in *2011 Design, Automation & Test in Europe*, pp. 1–4, IEEE, 2011. 45

[109] J. Peeters, N. Ventroux, T. Sassolas, and L. Lacassagne, "A SystemC TLM Framework for Distributed Simulation of Complex Systems with Unpredictable Communication," in *Proceedings of the 2011 Conference on Design & Architectures for Signal & Image Processing (DASIP)*, pp. 1–8, IEEE, 2011. 45

[110] N. Ventroux, J. Peeters, T. Sassolas, and J. C. Hoe, "Highly-Parallel Special-Purpose Multicore Architecture for SystemC/TLM Simulations," in *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, pp. 250–257, IEEE, 2014. 45

[111] N. Ventroux and T. Sassolas, "A New Parallel SystemC Kernel Leveraging Manycore Architectures," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 487–492, IEEE, 2016. 45, 46

[112] G. Busnot, T. Sassolas, N. Ventroux, and M. Moy, "Standard-Compliant Parallel SystemC Simulation of Loosely-Timed Transaction Level Models," in *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 363–368, IEEE, 2020. 45

[113] G. Busnot, T. Sassolas, N. Ventroux, and M. Moy, "Standard-Compliant Parallel SystemC Simulation of Loosely-Timed Transaction Level Models: From Baremetal to Linux-Based Applications Support," *Integration*, 2021. 45

[114] C. Sauer, H.-M. Bluethgen, and H.-P. Loeb, "Distributed, Loosely-Synchronized SystemC/TLM Simulations of Many-Processor Platforms," in *Proceedings of the 2014 Forum on Specification and Design Languages (FDL)*, vol. 978, pp. 1–8, IEEE, 2014. 46

[115] T. Vörtler, K. Einwich, M. Hassan, and D. Große, "Using Constraints for SystemC AMS Design and Verification," *DVCon Europe*, 2018. 46

[116] K. Ahnert and M. Mulansky, "Odeint—Solving Ordinary Differential Equations in C++," in *AIP Conference Proceedings*, vol. 1389, pp. 1586–1589, 2011. Available: `https://www.boost.org/doc/libs/1_66_0/libs/numeric/odeint/doc/html/index.html`. 59, 70

[117] TIMA Laboratory, "SystemC CT." [Online]. Available: https://gricad-gitlab.univ-grenoble-alpes.fr/tima/sls/projects/systemc-ct, 2021. (visited on 2021/11/05). 68, 86, 92, 96

[118] MathWorks, "Modeling an Automatic Transmission Controller." [Online]. Available: https://fr.mathworks.com/help/simulink/slref/modeling-an-automatic-transmission-controller.html, 2021. (visited on 2021/05/12). 68

[119] P. Shakouri, J. Czeczot, and A. Ordys, "Simulation Validation of Three Nonlinear Model-Based Controllers in the Adaptive Cruise Control System," *Journal of Intelligent & Robotic Systems*, vol. 80, no. 2, pp. 207–229, 2015. 68

[120] "Modeling an Automatic Transmission Controller." [Online]. Available: https://www.mathworks.com/help/simulink/slref/modeling-an-automatic-transmission-controller.html, 2019. (visited on 2019/26/07). 69

[121] "COSEDA Technologies GmbH | SystemC AMS Proof-of-Concept." [Online]. Available: https://www.coseda-tech.com/systemc-ams-proof-of-concept, 2019. (visited on 2019/02/09). 70

[122] E. A. Lee, "Fundamental Limits of Cyber-Physical Systems Modeling," *ACM Transactions on Cyber-Physical Systems*, vol. 1, no. 1, p. 3, 2017. 72, 74

[123] A. Massarini and U. Reggiani, "Computer-Aided Time-Domain Large-Signal Analysis of Networks with Switches," in *Proceedings of IEEE International Symposium on Industrial Electronics*, vol. 2, pp. 567–572, IEEE, 1996. 89

[124] D. Bedrosian and J. Vlach, "Time-Domain Analysis of Networks with Internally Controlled Switches," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 39, no. 3, pp. 199–212, 1992. 89

[125] R. J. Dirkman, "The Simulation of General Circuits Containing Ideal Switches," in *1987 IEEE Power Electronics Specialists Conference*, pp. 185–194, IEEE, 1987. 92

[126] A. Massarini and M. Kazmierczuk, "A New Representation of Dirac Impulses in Time-Domain Computer Analysis of Networks with Ideal Switches," in *1996 IEEE International Symposium on Circuits and Systems. Circuits and Systems Connecting the World. ISCAS 96*, vol. 1, pp. 565–568, IEEE, 1996. 94

[127] H. Vogt, M. Hendrix, and P. Nenzi, "Ngspice User' Manual Version 32." http://ngspice.sourceforge.net/docs/ngspice-manual.pdf, May 2020. 96

[128] The MathWorks, Inc., "Specialized Power Systems - MATLAB/Simulink." https://www.mathworks.com/help/physmod/sps/specialized-power-systems.html, 2020. 96

[129] "PLECS PIL | Plexim – The Simulation Platform for Power Electronic Systems." `https://www.plexim.com/products/plecs`, May 2021. 96

[130] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, pp. 483–485, 1967. 117