CONTINUOUS TIME AND MIXED-SIGNAL SIMULATION IN PTOLEMY II

UCB/ERL Memorandum M98/74

Jie Liu

Department of Electrical Engineering and Computer Science
University of California, Berkeley

liuj@eecs.berkeley.edu

12/15/98

Abstract

This report studies the continuous time and mixed-signal simulation techniques in the Ptolemy II environment. Unlike the nodal analysis representation usually seen in circuit simulators, the continuous time systems are modeled as signal-flow block diagrams in Ptolemy II. This representation is suitable for system-level specification, and the interaction semantics with other models of computation can be easily studied and implemented. The numerical solving methods for ordinary differential equations are discussed from the tagged-signal point of view and implemented in the continuous time domain. The breakpoint handling techniques are essential for performing correct simulation and supporting the interaction with other domains. Mixed-signal simulation of continuous time and discrete event models is discussed. Event detection can be performed using the breakpoint handling mechanism. The coordination of the execution of the two models are discussed. The result shows that when a continuous subsystem is embedded in a discrete event system, the inner system must run ahead of the global time and be able to roll back. Based on the result, a correct and efficient simulation strategy is presented. As a case study, the mixed-signal simulation techniques are applied to a micro accelerometer with sigma-delta kind of digital feedback.

Acknowledgments

First of all, I would like to express my gratitude to my research advisor, Professor Edward A. Lee for introducing me to this research area. It is his continuous support, guidance, and encouragement that make this project possible. He gave extremely valuable comments on the manuscript of this report, from typos to issues of mathematical completeness and accuracy.

I wish to thank Professor Kristofer Pister for serving as the second reader for this report. It is his extraordinary MEMS lectures that inspired me for the case study.

I am grateful to the Ptolemy group for the free and positive atmosphere. The discussion with the group members are always helpful. Xiaojun Liu, Yuhong Xiong, James Lundblad, and Lukito Muliadi read the draft of the report and provided excellent suggestions.

This research is part of the Ptolemy project, which is supported by the Defense Advanced Research Projects Agency (DARPA), the State of California MICRO program, and the following companies: The Cadence Design Systems, Hewlett Packard, Hitachi, Hughes Space and Communications, Motorola, NEC, and Philips.

Finally, I want to thank my family for the love and encouragement, and thank my parents for their constantly support in spirit. This report is dedicated to them.

J. L.

Table of Contents

List of Figures

1 Introduction

Mixed-signal simulation aims to help the design of systems with both discrete event and continuous time parts, or discrete event systems that interact with a continuous environment. In a discrete event system, the signals of interest contain events located discretely on the time axis. An event consists of a value and a time stamp. This model, which can deal with both functional information and timing information, is widely used in modeling digital hardware [3] [27], embedded software [5] and communication systems [6]. The signals of continuous time systems are waveforms that have value at all the time points. Continuous time models are usually used for analog circuits, physical processes, sensors, and accurators. For a typical embedded system, the core part is usually clock driven digital circuits, like a micro-controller running a real-time program, a DSP processor running a signal processing algorithm, and some digital application-specific circuits. The environment of the system, the sensors and the accurators, are continuous time subsystems that interact with the digital core. Seamlessly simulating the entire system could help designers understand the overall behavior of the system, and provide timing information to aid the further refinement of the design.

Our work integrates the simulation of continuous time systems with discrete event systems under the Ptolemy II infrastructure [23]. The target is general purpose system-level simulation, which makes no assumptions about the properties of the systems, like the time constant, stability, or equal separation of events. Systems are represented as block diagrams, in which each block is a component of the system that processes input signals and produces output signals. The blocks communicate by signals transferred on the arcs. This representation suggests a different implementation of the continuous time simulation algorithms than the nodal analysis methods usually seen in circuit simulators like SPICE and Saber. In our representation, the conversions of events and waveforms are straightforward, and the interaction of the discrete event system and continuous time system can be analyzed formally under a uniform tagged-signal model.

The simulation of systems that consist of continuous and discrete parts has been studied in various contexts in the last decade. Typical work can be found in the circuit simulation community, under the term of "*mixed-mode simulation*" [24], and in the control community, under the term of "*digital control systems*" [21] and "*hybrid systems*" [2].

The mixed-mode simulation technique targets electronic systems that contain both analog and digital circuits. The digital part of the system is considered an abstraction of the analog system. A common view is that "*Digital parts are merely analog parts that are over driven.*" The values of the digital signals (events) are discretized into levels, and the events are triggered from the continuous part by threshold crossings. As a result, there is only a finite number of event values. Since in the digital part of the system only certain events, like logical switches, are interesting, the simulation of the whole system can be dramatically faster than simulating the entire system under the continuous time model. However, the event happening time are usually not accurately detected in this class of techniques. They rely on the global stability of the circuit, and iterative simulation methods to insure steady state accuracy [4]. Inaccuracy in detecting events makes tight feedback that crosses the digital/analog boundary difficult to simulate.

In most computer-based control work, as well as in digital signal processing, the events are viewed as samples of continuous signals. The event values could be real numbers, but the event time stamps are equally separated. This regularity of event times simplifies the simulation such that the discrete and continuous parts can execute alternately.

Accurately detecting events has been studied under the hybrid systems context in the control literature [12] [15] [19]. But in hybrid systems, the model that interacts with the continuous time model is finite state machines (FSM). The FSM model is not a timed model, so it cannot specify the timing information of the discrete systems. In our work, we borrowed the event detection techniques from the hybrid system literature and used them in mixed-signal simulations.

Report Outline

The rest of the report is organized as follows. Chapter 2 introduces the Ptolemy II infrastructure and the continuous time system representation. Chapter 3 presents the design and implementation of the CT domain in Ptolemy II, focusing on the semantic analysis and how the semantics is implemented. Chapter 4 first briefly introduces the discrete event domain in Ptolemy II. Then it discusses the techniques for mixed-signal simulation. The conversion of discrete events and continuous waveforms, and the coordination of execution are discussed in detail. Chapter 5 provides a case study to show how the mixed-signal simulation is applied to simulate a micro accelerometer. Chapter 6 concludes the work and discuss directions for future work. Appendix A gives a survey of root finding techniques for nonlinear functions, which is used in event detection.

2 Continuous Time System Representation

Since Ptolemy II is the environment for our implementation, we introduce its infrastructure and terminology in this Chapter. Then the underlying mathematical model for continuous systems — ordinary differential equations (ODEs) — is briefly reviewed. We list the existence and uniqueness condition of the solution of ODEs, and discuss two classes of widely used numerical methods for solving them. Finally, we present how the ODE models can be specified using the abstract graphical syntax of Ptolemy II.

2.1 Ptolemy II Infrastructure

Ptolemy II is a complete redesign of the Ptolemy environment [9], which supports heterogeneous modeling and design of concurrent systems. The design principle of Ptolemy II is that choosing the *models of computation* plays an important role in the process of designing complex embedded systems. A model of computation is the set of "laws of physics" that governs the interaction of the components in a model. The Ptolemy II software provides an infrastructure that allows designers to explore and integrate different models of computation to achieve the overall design goals. Each model of computation is called a *domain* in Ptolemy II.

Ptolemy II core packages provide functionality for constructing and traversing an abstract clustered graph, encapsulating data and parameters, transferring data, sequencing execution, invoking graph and mathematical algorithms etc. We will highlight some key packages that will be used in the representation of CT systems. This section is intended as a summary and not as an in-depth discussion. We strongly suggest that the reader refer to the Ptolemy II design document [23] for a better understanding.

2.1.1 The Kernel Package

Ptolemy II uses a graphical representation for concurrent and hierarchical systems. The kernel package defines a small set of Java classes that implement a data structure that supports uninterpreted clustered graphs. A graph consists of entities that have ports, plus relations that connect the entities through the ports. For example, in Figure 1, `E1` and `E2` are entities with ports `P1`and `P2` respectively. `R1` is a relation that *links* the ports. The ports are said to be *connected*.

A hierarchical graph can be constructed by making an entity in one level of the graph contain another graph. In this case, the container is called a `CompositeEntity`, and the contained graph elements are instances of `ComponentEntity`, `ComponentPort` and `ComponentRelation`. Note that composite entities themselves may be contained by another composite entity, so that the graph can be arbitrarily nested.

Figure 2 shows an example of a hierarchical graph. `E0` is a *top-level* CompositeEntity that has no container. `E0` contains `E3` and `E4`, while `E3` further contains `E1` and `E2`. Note that `P3` of `E3` has two kinds of links. The one that links to `R3`, which is at the same level of hierarchy as `E3`, is called a *outside link* or simply a *link*; the other that links to `R1`, which is contained by `E3`, is called an *inside link*.

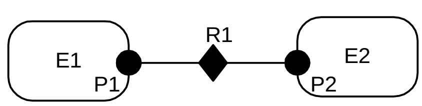


FIGURE 1. Entities, ports and relations.

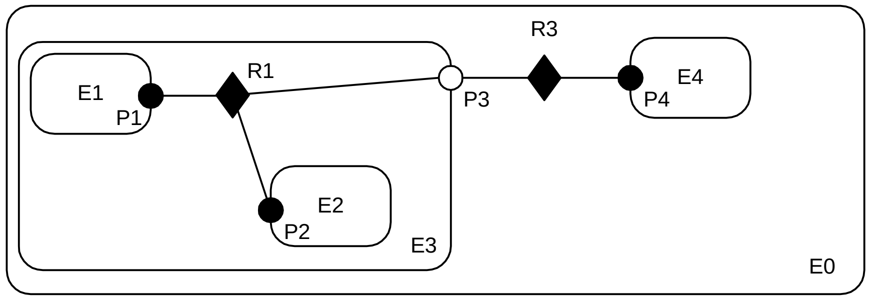


FIGURE 2. A hierarchical graph.

A CompositeEntity can be *opaque*, in the sense that when looking from the outside, none of its containees can be observed. Otherwise, it is called *transparent*. Eliminating the transparent CompositeEntities from a graph is called *flattening* the graph. A set of methods, whose names start with “deep”, are used to access the flattened graph. In our example, if `E3` is opaque, calling `P4.deepConnectedPorts()` will return `P3`. However, if `E3` is transparent, the same method will return `P1` and `P2`. In the latter case, `P3` is a *transparent port*.

2.1.2 The Actor Package

While the kernel package only provides an abstract graph syntax; the actor package attaches the semantics of message passing and execution to it. This semantics is shared by a number of models of computation.

Messages are encapsulated in *tokens.* A port that supports message passing is called an *IOPort*. Relations that link IOPorts are *IORelations*. An IOPort can be an *input port*, *output port*, or both, depending on whether it can receive tokens, send tokens or both. An input port contains receivers, which hold tokens before they are consumed. The way receivers handle tokens is a crucial part of the semantics of the domains. An `IOReltion` that links more than one input port acts like a *fork*, such that for each token that is sent from a linked output port, all the linked input ports receive it. IOPorts and IORelations have *width* which is greater than or equal to one. The width indicates the number of channels through which the data can be transferred. An IOPort with width greater than one is called a *multiport*, distinguishing it from a *singleport*, which has width one.

The `Executable` interface defines how an object can be invoked. An *execution* is defined to be one invocation of initialize(), followed by any number of iterations, followed by one invocation of wrapup(). The initialize() method is assumed to be invoked exactly once during the lifetime of an execution of an application. It may be invoked again to restart an execution. An *iteration* is defined to be one invocation of prefire(), any number of invocation of fire(), and one invocation of postfire(). The wrapup() method will be invoked exactly once per execution, when the execution terminates. The methods initialize(), prefire(), fire(), postfire(), and wrapup() are called the *action methods*.

An `Actor` is an executable entity. There are two types of actors, `AtomicActor`, which is a single entity, and `CompositeActor`, which is an aggregation of actors. AtomicActors implement the executable interface and add their own functionality. The execution of the components of a CompositeActor is governed by a `Director`. Directors also implement the executable interface, and all the executable functions of a CompositeActor are delegated to its director. The director of the container of a composite actor is called the *executive director* for the composite actor. If a composite actor has no director, then it is called *transparent*, and it relies on its executive director to execute. In general, the top-level composite actor (which has no containers) should have a director, and it has no executive director.

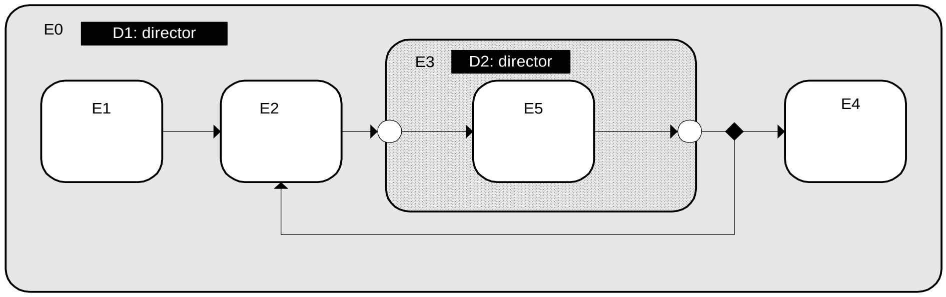


FIGURE 3. Hierarchical graphs with inputs and outputs.

To follow the convention of block diagrams, we use a slightly different visual notation for the clustered graph that is built by the actor package classes. As shown in Figure 3, the IOPorts of atomic actors are omitted, so are the IORelations that link only one input port and one output port. The connections are drawn as arcs with arrows that point to the input ports. IOPorts for composite actors are drawn as white circles, and IORelations that act as forks are drawn in black diamonds. In Figure 3, E1, E2, E4, and E5 are AtomicActors, while E0 and E3 are CompositeActors. D1 and D2 are Directors for E0 and E3 respectively. D1 is the executive director for E3.

2.2 Ordinary Differential Equations

We choose ordinary differential equations as the underlying mathematical model for continuous time systems because it is widely used, has mature numerical solving methods, and the system has a deterministic (unique) solution under simple conditions.

In particular, we consider the system dynamic which can be formulated as an initial value ODE problem:

$$\begin{aligned} \dot{x} &= f(x, u, t) \\ x(t_0) &= x_0 \end{aligned} \tag{1}$$

where, $t \in \Re$, $t \geq t_0$, a real number, is the time. At any time t, $x \in \Re^n$, an n-tuple of real numbers, is the state of the system; $u \in \Re^m$ is the m-dimensional input of the system; $\dot{x} \in \Re^n$ is the derivative of x with respect to time t, i.e. $\dot{x} = \frac{dx}{dt}$.

The solution to this set of ODEs is a waveform in the n-dimensional state space such that at time t, the derivative of the waveform for given t and $u(t)$ is $f(x(t), u(t), t)$. This waveform is also called the *state trajectory* or simply the *trajectory* of (1). It can be shown that under the following conditions on f, a unique solution to (1) exists [10]. Here we denote by D a set in $\Re$ which contains at most finite number of points per unit interval, and $\Re \backslash D = \Re \cap \overline{D}$ the difference set of $\Re$ and D.

Theorem 1. [Existence and uniqueness of the solution of an ODE] Consider the initial value ODE problem (1). If f satisfies the conditions:

1. [*Continuity Condition*] Let D be the set of possible discontinuity points; it may be empty. For each fixed $x \in \Re^n$ and $u \in \Re^m$, the function $f: \Re \backslash D \to \Re^n$ in (1) is continuous. And $\forall \tau \in D$, the left-hand and right-hand limit $f(x, u, \tau^-)$ and $f(x, u, \tau^+)$ are finite.

2. [*Lipschitz Condition*] There is a piecewise continuous bounded function $k : \Re \to \Re^+$, where $\Re^+$ is the set of non-negative real numbers, such that $\forall t \in \Re, \forall \zeta, \xi \in \Re^n, \forall u \in \Re^m$

$$\|f(\xi, u, t) - f(\zeta, u, t)\| \le k(t)\|\xi - \zeta\| . \qquad (2)$$

Then, for each initial condition $(t_0, x_0) \subseteq \Re \times \Re^n$ there exists a *unique* continuous function $\psi: \Re \to \Re^n$ such that,

$$\psi(t_0) = x_0 \qquad (3)$$

and

$$\dot{\psi}(t) = f(\psi(t), u(t), t) \qquad \forall t \in \Re \backslash \mathrm{D} . \qquad (4)$$

This function $\psi(t)$ is called the *solution* through (t_0, x_0) of the ODE (1).

◆

A proof of the theorem can be found in [10]. Usually, only the solution on a finite time interval $[t_0,t_f]$ is interesting. The least upper bound of $k(t)$ on this interval is called the *Lipschitz constant* $\boldsymbol{L}$ in $[t_0,t_f]$. I.e.

$$\boldsymbol{L} = max(k(t)) \qquad \text{for } t \in [t_0, t_f] . \qquad (5)$$

Note that $\psi(t)$ is a continuous waveform. In general, it is not in an analytical form, and it is impossible for digital computers to find, or even represent it in an infinitely precise way. The simulation of (1) can only be performed on a discrete set of time points. Let $[t_0,t_f]$ be the time interval of interest, we denote

$$Tc = \{t_0, t_1, t_2, \ldots t_n, \ldots t_f\}, Tc \subset [t_0, t_f] \qquad (6)$$

such that

$$t_0 < t_1 < t_2 < \ldots < t_n < \ldots < t_f . \qquad (7)$$

A “snapshot” is taken on each of these time points. Here we denote by:

- t_n : the *n*-th time point, to explicitly show the discretization of time. We also write *t* if the order *n* is not important.
- $x[t_i, t_j]$: the (continuous) state trajectory from time t_i to t_j;
- $x(t_n)$: the *precise* solution of (1) at time t_n;
- x_{t_n} : the *numerical* solution of (1) at time t_n;
- $h_n = t_n - t_{n-1}$: step size of the discretization of time. We also write h if the position *n* in the sequence is not important. For accuracy reason, h may not be uniform.

Most numerical ODE solving methods try to find values x_{t_n} that approximate $x(t_n)$ in the increasing order of *n*. We call these methods the “time marching” methods, and will use these methods in this report.

According to the mean-value theorem (see e.g. [1]), if $h_{n+1} = t_{n+1} - t_n$ is small enough, then

$$x(t_{n+1}) = x(t_n) + h_{n+1} \cdot \dot{x}(\rho) \qquad (8)$$

for some $\rho \in [t_n, t_{n+1}]$. The time marching methods assume $x_{t_n} = x(t_n)$ and approximate $\dot{x}(\rho)$ in different ways using the “known information” at some history time points up to time t_n. Then by applying (8), we can compute $x_{t_{n+1}} \approx x(t_{n+1})$. Since applying (8) is the same as integrating $f(x, u, t)$ on the interval $[t_n, t_{n+1}]$, the process of computing $x_{t_{n+1}}$ from x_{t_n} is called one *integration step*.

We highlight two classes of widely used methods.

1. *Linear multistep methods.* The Linear Multistep (LMS) Methods have the general form:

$$\sum_{i=0}^{k-1} \alpha_i x_{t_{n-i}} + h_n \sum_{i=0}^{k-1} \beta_i \dot{x}_{t_{n-i}} = 0, \tag{9}$$

where $\alpha's$ and $\beta's$ are algorithm parameters, and k is the number of history points used. In particular, a two step LMS method, known as the *Trapezoidal Rule* (TR), has the form:

$$x_{t_n} = x_{t_{n-1}} + \frac{h_n}{2}(\dot{x}_{t_n} + \dot{x}_{t_{n-1}}). \tag{10}$$

This is proven to be absolutely stable[1] and the most accurate among all second order LMS methods. Notice that by plugging in the system dynamic (1) into (10), we obtain a set of algebraic equations of x_{t_n} at each time point t_n. The algebraic equations can be solved by iterative methods like the Newton-Raphson method [22], which find the fixed point of the equation from an initial guess. Also notice that LMS methods with order greater than 2 are not able to self start. They rely on lower order methods to provide the histories.

2. *Runge-Kutta methods*. The Runge-Kutta (RK) methods, instead of using the history points of the previous solutions, create interpolation points at each integration step to approximate $\dot{x}(\rho)$. In general, the k point Runge-Kutta methods has the form:

$$x_{t_n} = x_{t_{n-1}} + \sum_{i=0}^{k-1} B_i K_i \tag{11}$$

where

$$K_0 = h_{n-1} \cdot f(x_{t_{n-1}}, u(t_{n-1}), t_{n-1}), \tag{12}$$

$$K_i = h_{n-1} \cdot f\left(\sum_{j=1}^{i} A_{ij} \cdot K_j \ , u(t_{n-1} + h \cdot c_i), t_{n-1} + h \cdot c_i\right), \quad i = 1\ldots(k-1), \tag{13}$$

where $A \text{and} B$ are algorithm parameters, $c_i = \sum_{j=1}^{i} A_{ij}$, and k is the number of intermediate points used to calculate $\dot{x}(\rho)$. The first order RK method, also called the Forward Euler method, has the form:

$$x_{t_n} = x_{t_{n-1}} + h_n \cdot \dot{x}_{t_{n-1}}. \tag{14}$$

The RK methods make no assumption on the history and can self start. It is also easier to adjust step sizes than with the LMS methods. On the other hand, although they have sophisticated order and step size control mechanisms, the RK methods do not have a mature stability theory.

LMS methods, particularly the TR method, are widely used in circuit simulators, like SPICE and Saber[2] while RK methods are widely used in simulating mechanical and control systems [26], as in Matlab[3].

---

1. Absolute stable roughly means the local errors do not accumulate when the integration steps increase.
2. A commercial analog simulator from Analogy Inc.
3. A commercial product from Mathworks Inc.

2.3 Block Diagrams for CT Systems

There are usually two ways to specify a continuous time system, the conservative law model and the signal-flow model [14]. The conservative law model defines a system by its physical components, which specifies relations of *cross* and *through* variables, and the *conservative laws* are used to compile the component relations into global system equations. For example, in circuit simulations, the cross variables are voltages, the through variables are currents, and the conservative laws are the Kirchhoff's laws. This model directly reflects the physical components of a system, thus is easy to construct from a potential implementation. The actual mathematical form of the system is hidden behind the scene.

In the signal-flow model, entities in the system are maps that define the mathematical relation between the input and output signals. Entities communicate by passing signals. This model directly reflects the mathematical relations among signals, and is more convenient for specifying the systems that do not have an explicit implementation yet.

In the CT domain of Ptolemy II, the signal-flow model is chosen as the interaction semantics. The conservative law semantics may be used within an entity to define its I/O relation. There are four major reasons for this decision:

1. *The signal-flow model is more abstract*. Ptolemy is focused on the system-level design and behavior simulation. It is usually the case that at this stage of a design, the users are working with simplified mathematical models of a system, and the implementation details are unknown or not cared about.

2. *The signal flow model is more flexible and extensible*, in the sense that it is easy to make topology changes in the problem level. For models like hybrid systems, it is more convenient to manipulate the system at this level.

3. *The signal flow model is consistent with other models of computation in Ptolemy II*. Most models of computation in Ptolemy use message-passing as the interaction semantics. Choosing the signal-flow model for CT makes it consistent with other domains, so the interaction of heterogeneous systems is easy to study and implement.

4. *The signal flow model is compatible with the conservative law model*. For physical systems that are based on conservative laws, it is always possible to wrap them into an entity in the signal flow model. The inputs of the entity are the excitations, like the voltages on voltage sources, and the outputs are the variables that the rest of the system may be interested in.

To model a full system with inputs and outputs, we add an output function *g*() to the system dynamics (1). So an entire continuous time system is specified by:

$$\begin{aligned} \dot{x} &= f(x, u, t) \\ y &= g(x, u, t) \\ x(t_0) &= x_0 \end{aligned} \tag{15}$$

where $y \in \Re^l$ is the l-dimensional output of the system. We consistently call the equation $\dot{x} = f(x, u, t)$ the system dynamics. Notice that adding the output function *g*() has nothing to do with the system dynamics. The simulation strategy of (15) is that, at each time point t_n, find the value of the state x_{t_n}, then apply the output function *g*() to get the output of the system y_{t_n}.

A block diagram representation of (15) is shown in Figure 4:

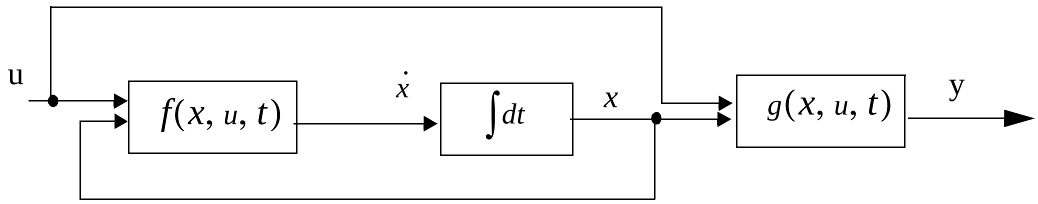


FIGURE 4. A conceptual block diagram for continuous time systems.

The signals in Figure 4, u, $\dot{x}$, x, and y, are continuous waveforms flowing from one block to the next. Since time is shared by all blocks, it is not considered an input. At any fixed time t, if the "snapshot" values $x(t)$ and $u(t)$ are given, $\dot{x}(t)$ and $y(t)$ can be found by evaluating $f(\ )$ and $g(\ )$, which can be achieved by firing the respective blocks.

3 Design and Implementation of the CT Domain

3.1 CT Semantics

Block diagrams only provide a syntax for specifying systems. The simulation or execution of the systems is defined by the semantics. Continuous time systems have strong semantic properties that distinguish them from other models of computation. This section briefly discusses the semantics of the CT model using the *tagged signal model*, which is a meta model to compare the denotational semantics of some concurrent models of computation [17].

3.1.1 Time

In the tagged signal model, an *event e* is a member of $E = T \times V$, where in this case $T = \Re$ is a set of *tags* representing time, and *V* is a set of *values*. If we consider the models with an earliest time t_0, like in (15), then $T = [t_0, \infty)$. Tags in general can be used to model time, precedence relationship, synchronization points etc. Values represent the operands and results of computations. A *signal* $s \in S$ is a set of events, where $S = \wp(E)$ is the *power set* (a set of all subsets) of all events.

In the continuous time model, the set of tags of any signal is the whole set *T*. This indicates the follows.

1. A "snapshot" of all the signal values at t is called the *behavior of a system at time* t. A system must have a behavior at any time in *T*. The execution engine, although marching at discrete time points, should be capable of finding the behavior of the system at any time of interest.
2. In signal flow models, function evaluations are achieved by firing corresponding actors, which consume the input tokens and produce output tokens. If an actor in the CT domain has n inputs, then there must be exactly one token at each input port when the actor is fired, and the time of all the events must be the same. The result of a firing is one event token at each output port with the same time. So each (single) port of a CT actor can only hold one token at any time, and this token must be consumed before the next token is generated.

Time is *global* in a CT system, in the sense that all the blocks share the same (continuous) clock. This global notion of time forces a total order of the behavior of the system. Time marching ODE solvers are consistent with this total order, such that the behavior of the system is found in the order of the time points.

3.1.2 Fixed-point Behavior

A process P is a subset of S^N, where S^N is the set of all *N-tuples* of signals. A particular $\mathbf{s} = [s_1, s_2 \ldots s_N] \subseteq S^N$ is said to *satisfy* the process if $\mathbf{s} \in P$. If $N = N_i \cup N_o$ and $N_i \cap N_o = \Phi$, where S^{N_i} is the set of input signals and S^{N_o} is the set of output signals, then the process is a *functional* process that maps the input signals to the output signals. Functional processes can be considered as a set of constrains that the signals must satisfy.

For a continuous time system, at any time point t_n, the functional processes reduce to the *firing function* of the actors. The composition of the functional processes is then a set of firing functions that map the input signal values at an instance to the output signal values at that instant. This set of functions could be either *explicit* or *implicit*. An explicit function is "operational," which means that it explicitly assigns the output values from the input values. An implicit function is "denotational," which means that it only specifies the relationship that the output values and the input values must satisfy. For example, an algebraic equation:

$$v_o = \varphi(v_i, v_o) \tag{16}$$

is an implicit function from input v_i to output v_o if for each v_i there is a unique v_o that satisfies (16).

An integrator is a functional map from $\dot{x}$ to *x*. The integration methods are just algebraic approximations of integrators with finite precision. Different integration methods may reduce a feedback loop with integrators to explicit or implicit maps. For example, as described in section 2.3, at time t_n, the linear multistep method of order *k* replaces the integrator by a function as shown in (9). Notice that the feedback loop may reduce to an explicit function, if $\beta_0 = 0$, or an implicit function if $\beta_0 \neq 0$.

Generally, at any time point t_n, a continuous time system dynamics is reduced to one of the following two forms:

$$x_{t_n} = F_E(x_{t_{n-1}}) \tag{17}$$

or

$$x_{t_n} = F_I(x_{t_n}) \tag{18}$$

where *x* is the state of the system, $F_E(\ )$, and $F_I(\ )$ are derived from time t_n, input $u(t_n)$, function *f*, and the history of *x* and $\dot{x}$. For example, the forward Euler solver, shown in (14), is in the form of (17); the trapezoidal rule solver, shown in (10) is in the form of (18). Solving (17) or (18) at a particular time point is called an *iteration* of the CT simulation.

Equation (17) can be solved simply by a function evaluation and an assignment. But the solution of (18) is the *fixed point* of *F*, which may not exist, may not be unique, or may not be able to be found. The *contraction mapping theorem* [8] shows the existence and uniqueness of the fixed point solution, and provides one way to find the fixed point.

Here we consider the Euclidean space $\Re^n$, which is the *n*-dimensional vector space of reals with the Euclidean norm (2-norm) as the metric. I.e. $\forall a = [a_1, a_2, \ldots a_n] \in \Re^n$,

$$\|a\| = \sqrt{\sum_{i=1}^{n} a_i^2}\,. \tag{19}$$

For a vector $x \in \Re^n$, a function $F: \Re^n \to \Re^n$ is called a *local contraction map at x*, if it satisfies the following two conditions:

a) There exists $\varepsilon > 0$, such that $\forall \xi, \eta \in \Re^n$ that satisfy $\|\xi - x\| \leq \varepsilon$ and $\|\eta - x\| \leq \varepsilon$, it is true that $\|F(\xi) - x\| \leq \varepsilon$ and $\|F(\eta) - x\| \leq \varepsilon$. I.e. ξ, η and their images are all within an ε ball centered at *x*.

b) For any ξ and η in a), $\exists 0 \leq \delta < 1$ such that

$$\|F(\xi) - F(\eta)\| \leq \delta \|\xi - \eta\|\,. \tag{20}$$

The largest ε such that this is true is called the *contraction radius at x*.

Theorem 2. [Contraction Mapping Theorem.] If $F: \Re^n \to \Re^n$ is a local contraction map at *x* with contraction radius ε, then there exists a unique fixed point of *F* within the ε ball centered at *x*. I.e. there exists a unique $\sigma \in \Re^n$, $\|\sigma - x\| \leq \varepsilon$, such that $\sigma = F(\sigma)$. And $\forall \sigma_0 \in \Re^n$, $\|\sigma_0 - x\| \leq \varepsilon$, the sequence

$$\sigma_1 = F(\sigma_0),\ \sigma_2 = F(\sigma_1),\ \sigma_3 = F(\sigma_2),\ \ldots \tag{21}$$

converges to σ.

◆

Theorem 2 is a special case of the *Banach fixed point theorem*, since $\Re^n$ with the above metric is a *complete metric space* [8].

Theorem 2 can be used to find the solution of (18) at each time point. The semantics indicates that the fixed point is an instantaneous behavior, so the F_I function in (18) should keep unchanged during one iteration of the simulation. This means the topology of the system, all the parameters, and all the internal states that the firing function depends on should maintain unchanged.

3.1.3 Discontinuity

Theorem 1 allows the function *f* to be discontinuous at a countable number of discrete points, which are also called the *breakpoints*. These breakpoints may be caused by the discontinuity of input signal *u*, or by an intrinsic property of *f*. In theory, the solutions at these discontinuous points are not well defined. But the left and right limit at these points are. So instead of solving the ODE at those points, we would actually try to find the left and right limit.

A breakpoint may be known beforehand, in which case it is called an *expected breakpoint*. For example, a square wave source actor can predict its next flip time. This information can be used to control the discretization of time. A breakpoint can also be *unexpected*, which mean it is unknown until the time it occurs. For example, an actor that varies its functionality when the input signal crosses a threshold can only report a “missed” breakpoint after an integration step is finished.

One impact of the discontinuities on the ODE solvers is that the history solutions before the breakpoint are useless in approximating the derivative of *x* after the breakpoint. The solver should resolve the new initial conditions and start the solving process as if it is at the starting point.

3.2 Directing The Execution

3.2.1 Controlling Step Sizes

Choosing the right time points to approximate a continuous time system behavior is one of the major tasks of simulation. There are three factors that may impact the choice of the step size.

- *Error control*. For all integration methods, the *local error* at time t_n is defined as a vector norm (say, the 2-norm) of the difference between the actual solution $x(t_n)$ and the approximation x_{t_n} calculated by the integration method, given that the last step is accurate. That is, assuming $x_{t_{n-1}} = x(t_{n-1})$ then
$$E_{t_n} = \left\| x_{t_n} - x(t_n) \right\|. \qquad (22)$$
It can be shown that by carefully choosing the parameters in the integration methods, the local error is approximately of the *p*-th order of the step size, where *p*, an integer closely related to the *k*’s in (9) and (11), is called the *order* of the integration method. I.e. $E_{t_n} \sim O((t_n - t_{n-1})^p)$. Therefore, in order to achieve an accurate solution, the step size should be chosen to be small. But on the other hand, small step sizes means long simulation time. In general, the choice of step size reflects the trade-off between speed and accuracy of a simulation.
- *Convergence*. Theorem 2 shows that for implicit ODE solvers, in order to find the fixed point at t_n, the map $F_I(\;)$ must be a (local) contraction map, and the initial guess must be within the ε ball of the solution. It can be shown that $F_I(\;)$ can be made contractive if the step size is small enough. (The choice of the step size is closely related to the Lipschitz constant). So the general approach

for resolving the fixed point is that if the iterating function $F_I(\ )$ does not converge at one step size *h*, then reduce the step size by half and try again.

- *Breakpoints*. At breakpoints, the derivatives of the signals are not continuous, so the integration formula is not applicable. That means the breakpoints can not be crossed by one integration step. In particular, suppose the current time is *t* and the intended next time point is *t*+*h*. If there is a breakpoint at $t+\delta$, where $\delta < h$, then the next step size should be reduced to $t+\delta$. For a expected breakpoint, the director can adjust the step size accordingly before starting an integration step. However for an unexpected breakpoint, which is reported "missed" after an integration step, the director should be able to discard its last step and restart with a smaller step size to locate the actual breakpoint.

Notice that convergence and accuracy concerns only apply to some ODE solvers. For example, explicit methods do not have the convergence problem, and fixed step size methods do not have the error control feature. On the other hand, breakpoint control is a "generic" feature that is independent on the choice of ODE solvers.

3.2.2 Scheduling

Some continuous time (analog) system simulators work on mathematical representations that are explicitly in the form of (1) or (15). Although they may have a graphical interface to specify the system, the block diagrams are translated into the netlist form and, further, to the mathematical form in the preprocessing stage. We believe that directly working on the signal flow representation is more flexible and extensible. So we do not rely on a preprocessing step to generate mathematical equations. The evaluation of the function *f*() and *g*() in (15) is achieved by schedules and actor firings.

The scheduler partitions a CT system into three clusters: the *state transition cluster*, the *output cluster*, and the *event generation cluster*. In a particular system, these clusters may overlap.

The state transition cluster includes all the actors that are in the signal flow path for evaluating the *f*( ) function in (15). It starts from the source actors and the outputs of the integrators, and ends at the inputs of the integrators. A topological sort of the cluster provides an enumeration of the actors in the order of their firings. This enumeration is called the *state transition schedule*. After the integrators produce tokens representing x_t, one iteration of the state transition schedule gives the tokens representing $\dot{x}_t = f(x_t, u(t), t)$ back to the integrators.

The output cluster consists actors that are involved in the evaluation of the output function *g*() in (15). It is also similarly sorted in topological order. The *output schedule* starts from the source actors and the integrators, and ends at the sink actors.

Event generating actors[1] are continuous time actors that may generate unexpected breakpoints in the middle of an integration step. The event generation cluster includes all the actors in a path that starts from the integrators and sources, and ends at the event generate actors. This cluster is also topologically sorted to produce the *event generation schedule*.

A special situation that must be taken care of is the firing order of a chain of integrators, as shown in Figure 5. For the implicit integration methods, the order of firings determines two distinct kinds of fixed point iterations. If the integrators are fired in the topological order, namely $x_1 \to x_2$ in our example, the iteration is called the *Gauss-Seidel iteration*. That is, x_2 always uses the new guess from x_1 in this iteration for its new guess. On the other hand, if they are fired in the reverse topological order, the iteration is called the *Gauss-Jacobi iteration*, where x_2 uses the guess of x_1 in the last iteration for its

1. The name will be clear when the mixed-signal simulation is discussed in the next chapter.

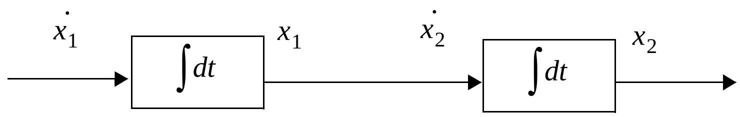


FIGURE 5. A chain of integrators.

new guess. The two iterations both have their pros and cons, which are thoroughly discussed in [20]. Gauss-Seidel iteration is considered faster in the speed of convergence than Gauss-Jacobi. For explicit integration methods, where the new states x_{t_n} are calculated solely from the history inputs up to $\dot{x}_{t_{n-1}}$, the integrators must be fired in their reverse topological order.

3.2.3 Dirac Impulses

The Dirac impulse signal $\delta(t)$ is a special kind of continuous time signal. It satisfies $\delta(t) = 0, \forall t \neq 0$, $\delta(0) = \infty$, and the integration

$$\int_{-\infty}^{\infty} \delta(t)dt = 1\,. \tag{23}$$

Feed an impulse to an integrator, say *A*, will change the state of the integrator x_A abruptly. I.e.

$$x_A(0^+) = x_A(0^-) + 1 \tag{24}$$

where $x_A(0^+)$ is the state after integrating the signal, $x_A(0^-)$ is the state before integrating the signal, they are both at time 0.

General integration method approximate $\delta(t)$ by a narrow square wave, as shown in Figure 6. The

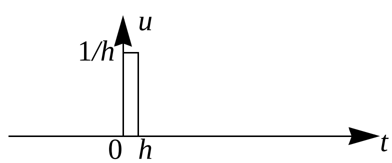


FIGURE 6. Approximation of an impulse.

width of the signal is the current step size *h*, and the height of the signal is 1/*h*. This approximation is usually not satisfactory, since the result is $x_I(0^+) = x_I(0^-)$, but $x_I(h) = x_I(0^-) + 1$. Vlach etc. present a method that overcomes this inaccuracy in [28]. The method, at time 0, uses the backward Euler method (LMS method with k=1, $\alpha_0 = 1$, $\alpha_1 = -1$, $\beta_0 = -1$), and uses the minimum step size h_{min}. The impulse is also approximated as in Figure 6. The result, of course, is $x_I(h_{min}) = x_I(0^-) + 1$. Then, integrate *backward in time* with the same step size, i.e. $h = -h_{min}$, to obtain $x_I(0^+) = x_I(0^-) + 1$. It can be shown that this method can also deal with signals like the derivatives of the impulse, which is impossible for other methods. This method, although it does not advance time, is an ideal method to handle impulse inputs and breakpoints. We call this solver the *impulse backward Euler* (IBE) solver.

3.3 Implementation

The java package `ptolemy.domains.ct.kernel` implements the continuous time semantics based on the kernel and the actor packages of Ptolemy II. The key classes in the CT kernel are: `CTReceiver`, `CTActor`, `CTCompositeActor`, `CTDirector`, `CTScheduler`, and `ODESolver`. The uniformed modeling language (UML) diagrams for the classes are shown in Figure 7 and Figure 8. The classes that are not in the CT kernel packages are drawn with dashed outlines. The constructors for classes `CTSingleSolverDirector`, `CTMultiSolverDirector`, `CTMixedSignalDirector`, `ForwardEulerSolver`, `BackwardEulerSolver`, `TrapezoidalRuleSolver`, `ExplicitRK23Solver`, `ImpulseBESolver` are omitted for space reason.

3.3.1 CTReceiver

Receivers are the token holders in message passing. `CTReceiver` reflects the semantics that time

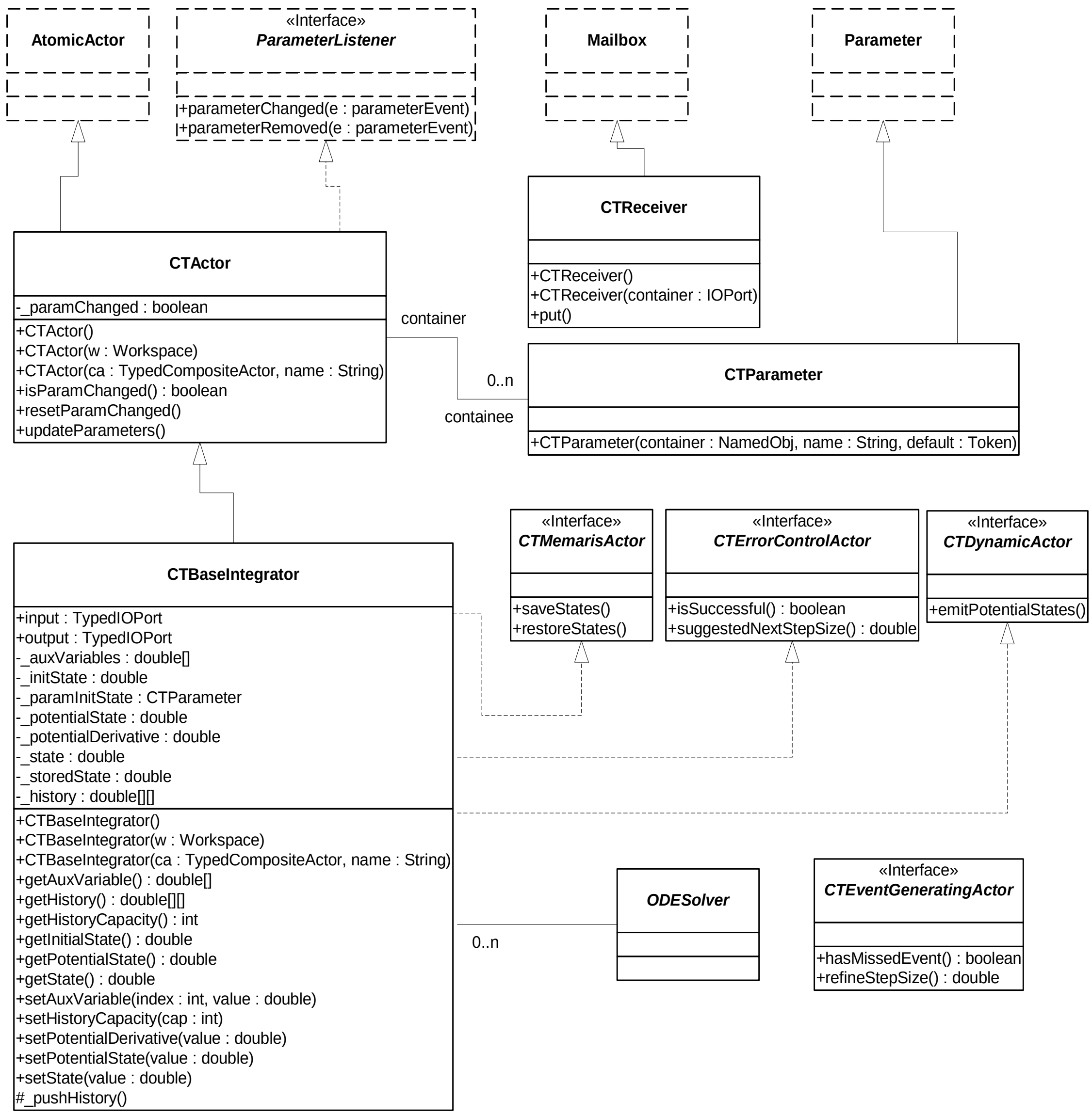


FIGURE 7. UML diagram of CT Actor related classes.

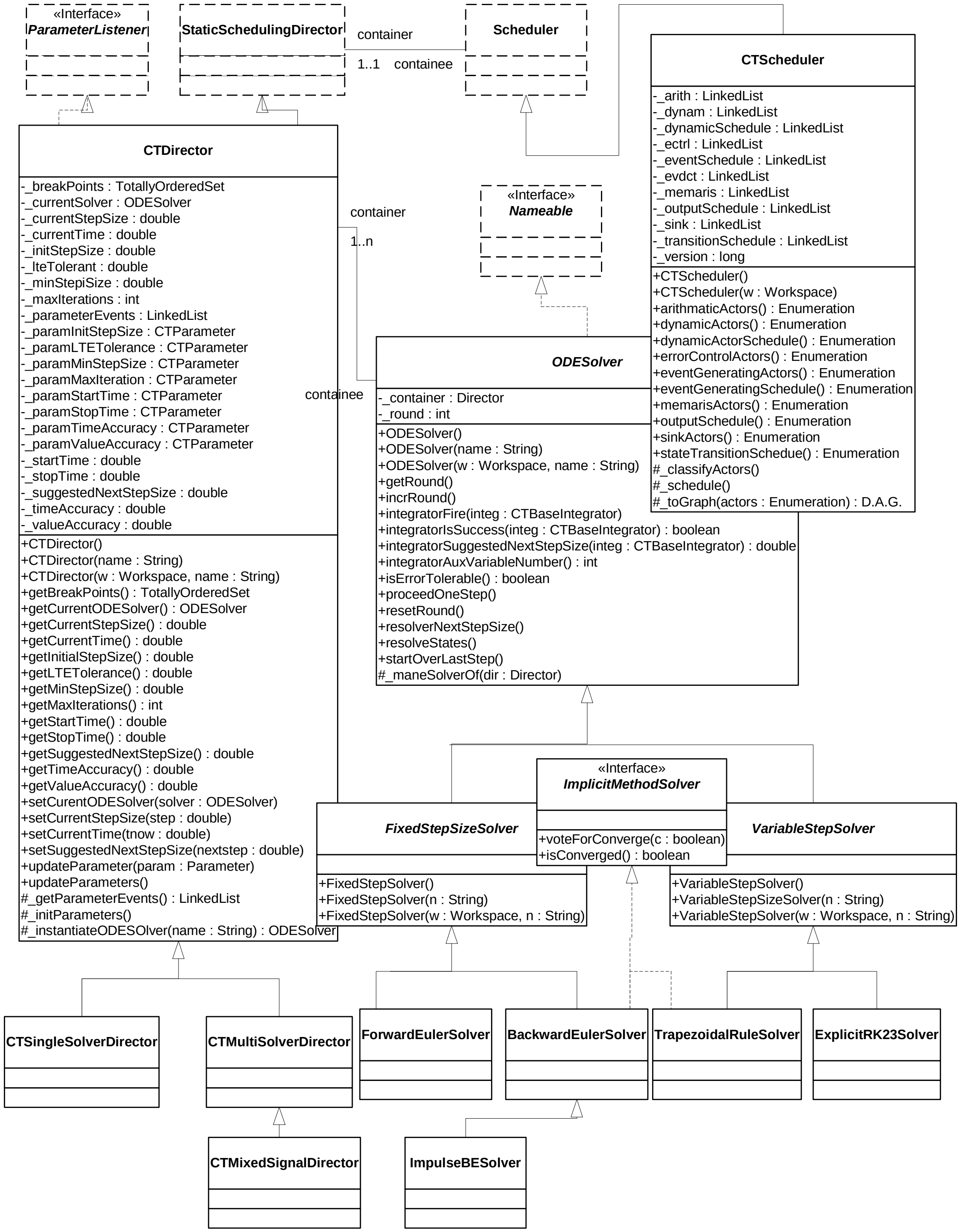


FIGURE 8. UML diagram of CTDirector related classes.

is continuous and any process (actor) must consume and produce tokens when it is fired. So CTReceiver extends ptolemy.actor.Mailbox, which is a receiver with capacity one. To reflect the fact that an integration step may fail due to error control or breakpoints, the CTReceivers must be able to discard the unconsumed token and receive new tokens when the ODE solvers are restarted from a failed step. When a full CTReceiver receives a new token, the old token will be overwritten.

3.3.2 CTActor and CTParameter

CTActor is the base class for the atomic actors in the CT domain. It extends ptolemy.actor.AtomicActor and builds in some default implementations that all the atomic actors in the CT domain may share. One of the major features is to handle parameters. The CT semantics requires that the functionality of an actor keep unchanged in the process of finding the fixed point. Since parameter changes will directly change the functionality of it container, they are only allowed to be changed at some particular points of the execution, namely between the successive iterations. To achieve this requirement, the CTActor class implements the ParameterListener interface, and the CTParameters register their containers as their listeners at the time they are constructed. When a CTParameter is changed at the run time, the CT actor containing it will be notified. The actor will not update the parameter until it is safe to do so. Instead, a flag is set to shown that there is a parameter changing. At the prefire() stage of the next iteration, the parameter will be updated.

In order to help the actor partitioning and scheduling, several interfaces are defined such that CT actors with different functionality may be distinguished. They are:

- CTDynamicActor. Dynamic actors are actors that have integrators. Integrators themselves are dynamic actors, so is an actor that implements (15). In the latter case, *u* is the input and *y* is the output. A unique property of dynamic actors is that they have initial states, and at the beginning of the execution they can emit the initial state response. Actors that implement this interface will be used to break the feedback loops in the scheduling.
- CTErrorControlActor. In order to ensure accuracy, the local error is controlled by some actors. These actors should implement the CTErrorControlActor interface. At the end of each integration step, the error control actors will be asked if their local errors are within tolerance. If not, the step size will be cut by half and the last step will be restarted. If yes, they will be asked for a suggestion for the next step size.
- CTEventGeneratingActor. These are actors that may generate unexpected breakpoints in the middle of an iteration step. They are used for constructing the event generation schedule and controlling unexpected breakpoints, as described in section 3.2.2.
- CTMemarisActor[1]. Some actors have memory; that is the output of the actor depends not only on the inputs but also on some internal variables that may vary. When the CT domain interacts with other domains, like the discrete event domain, it is possible that CT will discard some of its execution, and roll back to an earlier time. In that case, all the actors with memory should go back to their earlier state corresponding to the rollback time. To ensure correct rollbacks, all actors that have internal states should implement this interface to perform saving and restoring states if they are going to be correctly used in CT.

Integrators are special in the CT domain. CTBaseIntegrator implements CTDynamicActor, CTErrorControlActor, and CTMemarisActor, since it has initial state that is used to break the feedback loops; it does error controls in the variables step size methods; and it has memories which is

1. *Memaris* is the Latin word for memory. It is used here to distinguish from actors that model memories.

its state. Integrators behave differently under different ODE solvers. In order to seamlessly change ODE solvers during the simulation, the integrators delegate their fire( ) method and error control methods to the ODE solver that is in charge. `History` is a private array of past states and their derivatives, so methods using history steps can access them.

3.3.3 ODE solvers

`ODESolver` is the base class for all ODE solvers. The basic operation of an ODESolver is `proceedOneStep`(). The method will perform one "successful" integration step without concern for breakpoints. The conceptual execution of this method is shown in Figure 9. Different solvers will implement the methods: `resolveStates()`, `errorControl()`, `startOverLastStep()` and `resolveNextStepSize()` differently. For example, for fixed step size solvers, the `errorControl()` methods always return a "success"; the `startOverLastStep()` methods are empty; and `resolveNextStepSize()` methods always return the current step size. For explicit solvers, the `resolveStates()` methods fire the state transition schedule for a fixed number of times, while for implicit solvers, the methods fire the schedule repeatedly until the fixed point is reached.

3.3.4 CTDirector

`CTDirector` is the base class for directors in the CT domain. It manages shared parameters and

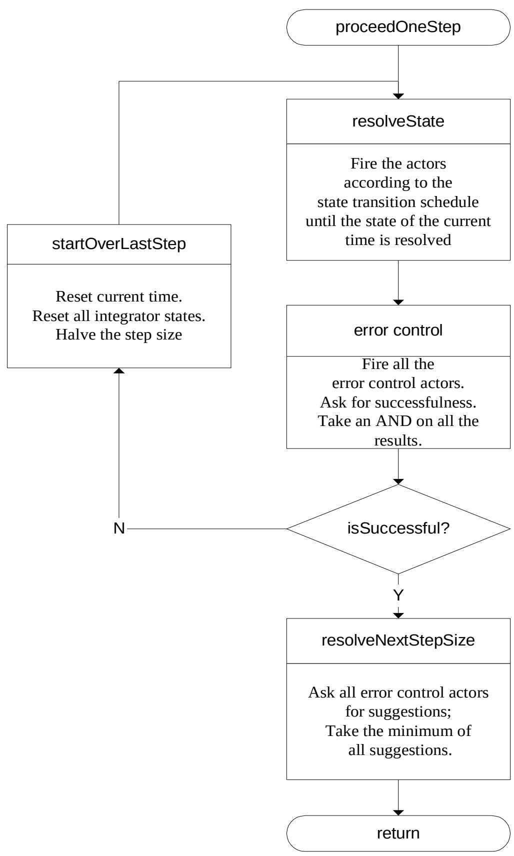


FIGURE 9. The proceedOneStep() method of ODESolver.

the breakpoint table. Tasks like choosing ODE solver and handling the breakpoints are left to individual subclasses. A CT director may have one or more ODE solvers; one of them, called the *current ODE solver,* is the one that takes charge of the current integration step. The reason for separating ODE solvers and directors is that in one execution, the director may use different solvers to deal with different situations; for example, a breakpoint solver may be used to deal with the first few steps after a breakpoint. So making ODESolver a separate class improves code reuse.

CTDirector manages parameters that may be required for subclasses and ODE solvers, including:

- *current time* (`currentTime`). Time is global in CT, meaning that all the actors share the same notion of time. So time is managed in the directors, and actors can reference the time by calling `getCurrentTime()`.
- *start time* (`startTime`). The start time of the simulation. This is only applicable when CT is the top level domain. Otherwise, the CT director should follow the times of its executive director.
- *stop time* (`stopTime`). The stop time of the simulation. This is only applicable when CT is the top level domain, too.
- *current step size* (`currentStepSize`). The step size that is used in the current integration step.
- *suggested next step size* (`suggestedNextStepSize`) For solvers with error control capability, the integrators may be able to predict an estimation for the next step size based on the local error of the current step. This variable is the minimum of all the predictions from the integrators.
- *initial step size* (`initialStepSize`). This is the step size that the user specifies as the desired step size. For fixed step size solvers, this step size will be used in all iterations. For variable step size solvers, this is only a reference.
- *local truncation error tolerance* (`LTETolerance`). The upper bound of the local truncation error. All error control actors will compare the estimated local error to this value. If the local error is greater than this value, then the integration step should be restart.
- *minimum step size* (`minStepSize`). The lower bound for adjusting step sizes. If this step size is used and the errors are still not tolerable, the step is considered failed. And the simulation aborts.
- *minimum time resolution* (`timeResolution`). This controls the comparison of time. Since CT director works on double precision of real numbers, it is sometime impossible to reach or step at a specific time point. If two time points are within this resolution, then they are considered identical.
- *value resolution* (`valueResolution`). This is used in the fixed point iterations. If in two successive iterations the states are within this accuracy, then the fixed point is considered reached.
- *maximum number of iteration per step* (`maxIterations`). This is used to avoid the infinite loops in the fixed point iterations. If the number of iterations exceed this value but the fixed point is still not found, then the fixed point procedure is considered failed. And the simulation aborts.

One of the major tasks of the CT directors is to manage the breakpoints. Recall that a breakpoint can be expected or unexpected, depending on whether it is known beforehand. Expected breakpoints are registered in a `TotallyOrderedSet`, which is called the *breakpoint table*. In the table, the breakpoints are sorted by their expected occurring time. Once a breakpoint is processed, it will be removed from the table. Multiple breakpoints at the same time are treated as one breakpoint. Before one integration step starting from t, if the intended step size is h, the director will query the breakpoint table for the earliest breakpoint, say t_b. If $t < t_b < t + h$, then the current step size is adjusted to be $t_b - t$. If $t_b = t$, which means that this is the first step after the breakpoint, then the breakpoint at t_b will be removed from the table. In this situation, the director may replace the current ODE solver with a breakpoint solver, and it may adjust the step size to the minimum step size. The handling of unexpected breakpoints is key for mixed-signal simulation, and will be discussed in the next chapter.

Three classes are derived from `CTDirector`, `CTSingleSolverDirector`, `CTMultiSolver-Director`, and `CTMixedSignalDirector`. The first two are for “pure” continuous time simulations, and can only handle expected breakpoints. They differ in whether a specific solver is used at the first few steps after the breakpoints. The flow of the fire method of the `CTMultiSolverDirector` is shown in Figure 10. `CTSingleSolverDirector` can be treated as a special case of it that has no `switchODESolver( )` call in the flow. The most complicated one, `CTMixedSignalDirector`, can handle both expected and unexpected breakpoints, and has the ability to interact with event-based models. The details will be discussed in the next chapter. The three directors provide trade-offs between speed and complexity of a simulation. If the application is CT only and all the breakpoints are

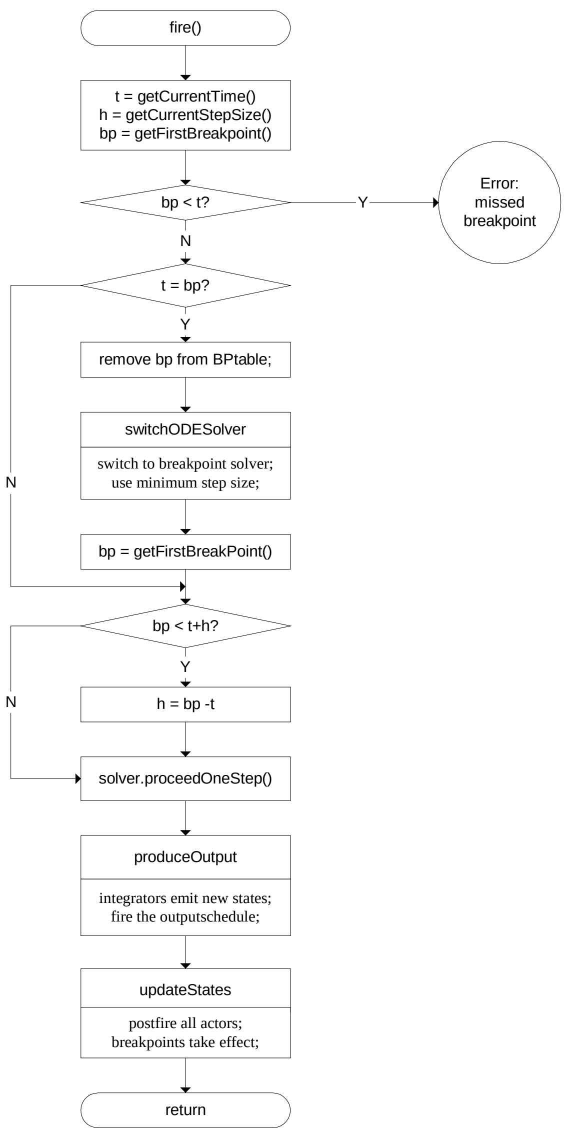


FIGURE 10. fire() method of CTMultiSolverDirector.

expected, then the first two directors are suitable and efficient. In addition, if the default integration method can self start and no impulse signals appear in the simulation, then `CTSingleSolverDirector` is approperate.

3.4 Example

A proof-mass, spring, and damper system is shown in Figure 11. The relation between the input force F_{in}, and the position of the proof-mass *x*, is given by the differential equation:

$$m\ddot{x}(t) + b\dot{x}(t) + kx(t) = F_{in}(t) \qquad (25)$$

where *m* is the mass of the proof-mass, *b* is the damping parameter, and *k* is the spring constant.

The block diagram representation of the system is shown in Figure 12, where the input force is modeled as a square wave. The TclBlend code for constructing of the system in the CT domain is given in Figure 13. There, $\mathrm{G1}=\frac{1}{m}$, $\mathrm{G2}=-\frac{b}{m}$, and $\mathrm{G3}=-\frac{k}{m}$. The parameters of the system[1] are

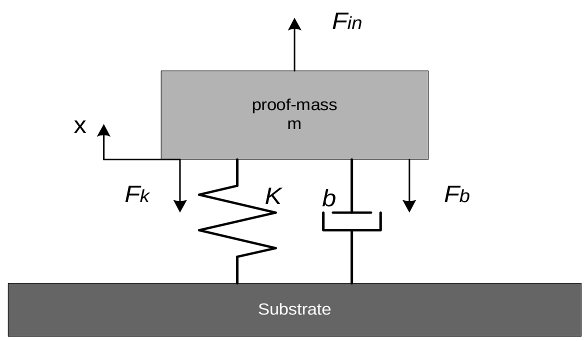


FIGURE 11. A dumped spring-mass system.

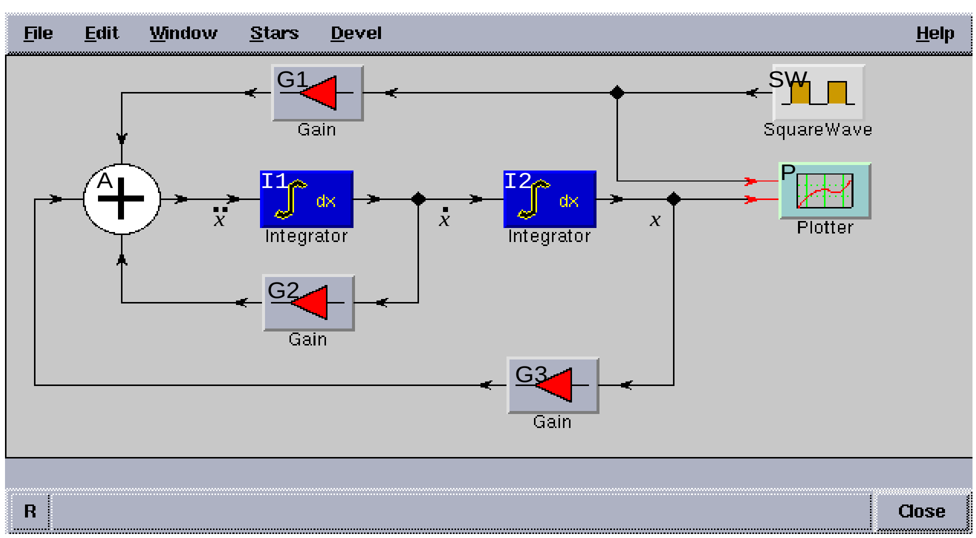


FIGURE 12. Block diagram for the spring-mass system.

1. Provided by Coyote Systems Inc.

$m = 500$, $k = 0.2$, $b = 5$. The input of the system has frequency 0.25 and magnitude ± 1.

The system is simulated from time 0 to 1.0 using the following ODE solvers:

- Single solver: fixed step size Forward Euler (FE). `initiaStepSize`=0.005
- Single solver: fixed step size Backward Euler (BE). `initialStepSize`=0.005
- Multi-solver: RK23 as default solver and BE as breakpoint solver (RK23). `initialStepSize` = 0.005, `minStepSize` = 1e-6
- Multi-solver: Variable step size Trapezoidal Rule as default solver and IBE as breakpoint solver

```
set sys [java::new ptolemy.actor.TypedCompositeActor]
$sys setName System
set man [java::new ptolemy.actor.Manager]
$sys setManager $man
set dir [java::new ptolemy.domains.ct.kernel.CTMultiSolverDirector DIR]
$sys setDirector $dir
#construct actors
set sqwv [java::new ptolemy.domains.ct.lib.CTSquareWave $sys SQWV]
set add1 [java::new ptolemy.domains.ct.lib.CTAdd $sys Add1]
set intgl1 [java::new ptolemy.domains.ct.lib.CTIntegrator $sys
Integrator1]
set intgl2 [java::new ptolemy.domains.ct.lib.CTIntegrator $sys
Integrator2]
set gain1 [java::new ptolemy.domains.ct.lib.CTGain $sys Gain1]
set gain2 [java::new ptolemy.domains.ct.lib.CTGain $sys Gain2]
set gain3 [java::new ptolemy.domains.ct.lib.CTGain $sys Gain3]
set plot [java::new ptolemy.domains.ct.lib.CTPlot $sys Plot]
#get ports
set sqwvout [$sqwv getPort output]
set add1in [$add1 getPort input]
set add1out [$add1 getPort output]
set intgl1in [$intgl1 getPort input]
set intgl1out [$intgl1 getPort output]
set intgl2in [$intgl2 getPort input]
set intgl2out [$intgl2 getPort output]
set gain1in [$gain1 getPort input]
set gain1out [$gain1 getPort output]
set gain2in [$gain2 getPort input]
set gain2out [$gain2 getPort output]
set gain3in [$gain3 getPort input]
set gain3out [$gain3 getPort output]
set plotin [$plot getPort input]
#create relations
set r1 [$sys connect $sqwvout $gain1in R1]
set r2 [$sys connect $gain1out $add1in R2]
set r3 [$sys connect $add1out $intgl1in R3]
set r4 [$sys connect $intgl1out $intgl2in R4]
set r5 [$sys connect $intgl2out $plotin R5]
set r6 [$sys connect $gain2out $add1in R6]
set r7 [$sys connect $gain3out $add1in R7]
$gain2in link $r4
$gain3in link $r5
$plotin link $r1
```

FIGURE 13. TclBlend code for constructing the spring-mass system in Ptolemy II.

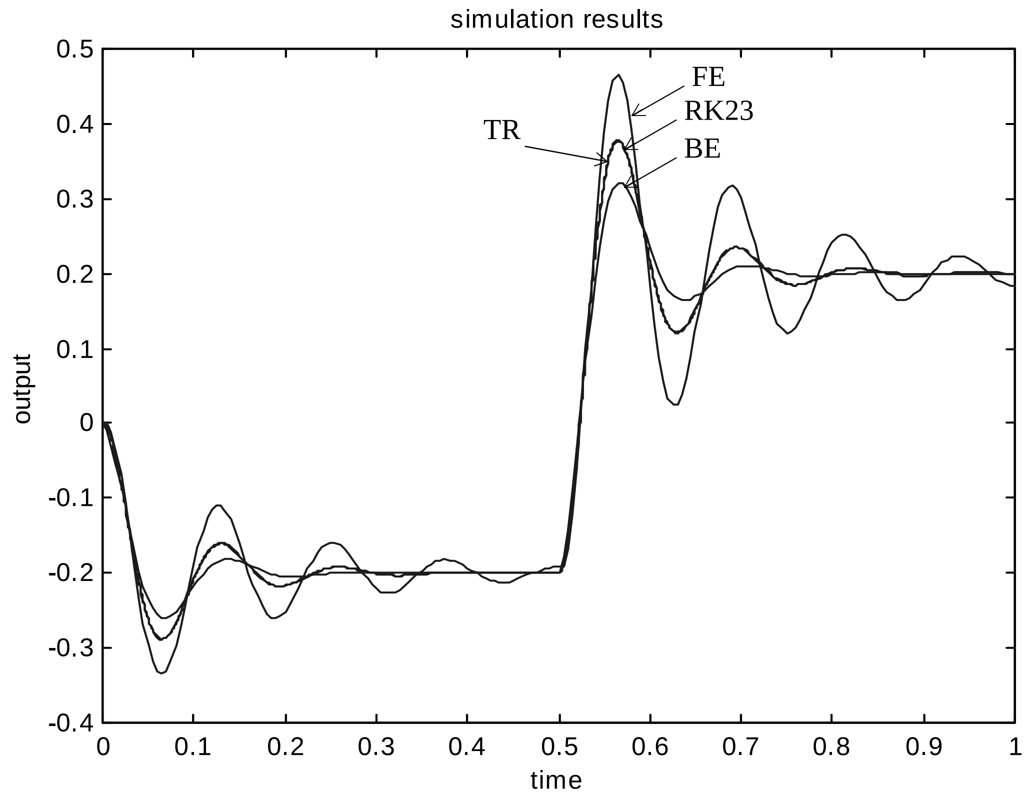


FIGURE 14. The simulation results of the spring-mass system.

(TR). `initialStepSize` = 0.005, `minStepSize` = 1e-6

Other parameters in CTDirector are:

```
maxIterations = 20; LTETolerant = 1e-4;
valueAccuracy = 1e-6; timeAccuracy = 1e-10.
```

The simulation results under different integration methods are shown in Figure 14. From the results we can see that FE method trends to overshoot, and BE trends to undershoot. The slightly large step sizes makes the error significant. TR and RK23 have similar results, and are more accurate than the fixed step size methods (for the not carefully chosen step sizes).

4 Mixed-signal Simulation

Mixed-signal simulation, in this report, refers to the simulation of systems that are specified partly in ordinary differential equations and partly in the discrete event models. In the discrete event (DE) model, the set of events are located discretely on the time axis. An *event* consists of a value and a time stamp. The components in a system respond to input events and produce output events (instantaneously or in the future) that may trigger other parts of the system.

In this chapter, we are going to analysis the mixed-signals in the tagged signal model and derive correct and efficient mixed-signal simulation techniques.

4.1 Discrete Event Model

We briefly review some semantic properties of the DE model that will be used later in this chapter. For a complete analysis of the discrete event model, please see [16].

4.1.1 Semantics

Following the tagged signal model in section 3.1.1, for a discrete event signal *s*, the set of distinct tags $T(s)$ is a subset of *T* such that $T(s)$ is *order-isomorphic* to a subset of the integers. That is, the tags are *ordered* and *countable*. These tags are called the time stamps. For ease of discussion, we use τ with possible subscripts to denoted the time stamps of the discrete events, and continue using t with possible subscripts for the time points of CT signals.

Causality is an important concept in the DE model. The formal definition of causality uses the Cantor metric on the process level [16], which is beyond the scope of this report. We roughly explain the concepts by using the firing functions of actors. Let $\Phi = \{\phi: S^{N_i} \times T \to S^{N_o}\}$ be the set of all firing functions an actor has in a simulation, and *M*, a positive integer, be the number of firings of the actor in the simulation. I.e. $\forall m \in \{1, 2, \ldots, M\}$, $\phi_m \in \Phi$, and $\phi_1, \phi_2, \ldots \phi_M$ are the sequence of firing functions executed in the simulation. The relation between the input signal $\mathbf{s}^I = [s^I_1, s^I_2, \ldots s^I_k, \ldots s^I_{N_i}] \in S^{N_i}$ and the output signal $\mathbf{s}^O \in S^{N_o}$ is given by procedure (26). In (26), The first three statements initialize the procedure. The output signal is set to an empty result. The while loop processes the pending events for each firing. Within the loop, τ^I_m is the smallest time stamp of the pending input events. The *m*-th firing function ϕ_m operates on the input signals at time τ^I_m, and produces the output s'. τ^O_m is the smallest time stamp of the events in s'. The processed events are removed from the input signals by the set subtraction, and the output s' is appended to the output signal by the set union operation.

$$
\begin{aligned}
&\mathbf{s}' = [s'_1, \ldots s'_l, \ldots s'_{N_o}] \in S^{N_o} \\
&\mathbf{s}^O = \Lambda_{N_o} \\
&m = 1
\end{aligned}
\tag{26}
$$

$$
\begin{aligned}
&\text{while}(m \le M)\{\\
&\quad \tau_m^I = \min_k (\min(T(s_k^I)))\\
&\quad \mathbf{s}' = \phi_m(\mathbf{s}^I(\tau_m^I), \tau_m^I)\\
&\quad \tau_m^O = \min_l (\min(T(s'_l)))\\
&\quad \mathbf{s}^I = \mathbf{s}^I - \mathbf{s}^I(\tau_m^I)\\
&\quad \mathbf{s}^O = \mathbf{s}^O \cup \mathbf{s}'\\
&\quad m = m + 1\\
&\}
\end{aligned}
$$

where Λ_{N_o} is the N_o-tuple of empty signals.

A DE actor is called *causal* if $\forall m \in M$,

$$\tau_m^O \ge \tau_m^I . \tag{27}$$

That is, for each firing, the minimum time stamp of the output events is not earlier than the time stamp of the earliest input events. If $\tau_m^O = \tau_m^I$ for some m, then we say the actor can have *zero delay*. A DE actor is called *strictly causal* if $\forall m \in M$,

$$\tau_m^O > \tau_m^I . \tag{28}$$

That is, for each firing, the minimum time stamps of the output events are strictly greater than the time stamps of the earliest input events. A DE actor is *delta causal* if $\exists \Delta > 0$, such that $\forall m \in M$,

$$\tau_m^O \ge \tau_m^I + \Delta . \tag{29}$$

4.1.2 Simulating DE Systems

A typical discrete event simulator operates by maintaining an *event queue*, in which the events are sorted by time stamp. During the simulation, the output events from all actors will be fed into the queue. At each iteration of the execution, the events with the smallest time stamp are removed from the queue, and the actors that respond to the events are fired. The time stamp of the event that is just removed from the queue is defined as the *current time* of the system. An *iteration* of the simulation is defined as processing all the simultaneous events at the current time. It has been shown in [16] that, for DE systems that are built by delta causal actors, this simulation technique yields the correct result. A looser condition is that all the actors must be causal, and at least one delta causal actor is required in a feedback loop.

How to handle simultaneous events (events with the same time stamp) is a key issue in designing a discrete event simulator. In Ptolemy II (as well as Ptolemy 0.x [13]), the actors in a DE system are topologically sorted, and a priority is assigned to each arc. The topological sort is based on the annotation of the actors in the graph indicating whether the actor can have zero delay. When such zero delay is possible, the topological sort views this as a precedence constraint. For example, for the system shown in Figure15, if C is a zero delay actor, then the firing order is A $\to$ C $\to$ B, and B will see two simultaneous events when it fires.

A slightly subtlety arises in discrete event simulations with sources of events. Since the source actors has no input, there is no event in the event queue that can trigger its firing. This problem is solved in Ptolemy II using *pure events*. A pure event is an event that has no value. Whenever the

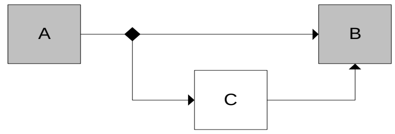


FIGURE 15. A DE system with a zero delay block.

source actor is fired, besides of emitting signal events, it puts a pure event into the event queue with itself as the destination and its next firing time as the time stamp. When this pure event is at the top of the event queue, the source actor can be refired. Of course, this requires the actor to be fired at least once at the beginning of the simulation. The refiring mechanism is also key for mixed domain simulations.

4.2 Mixed-signal System Representation

Ptolemy II mixes different models of computation by the container-containee relationship. As shown previously in Figure 2, an opaque composite actor in one domain can implement another domain internally. This mechanism controls the complexity of a design by information hiding, such that from the outer domain point of view, all the actors under its control have the same type.

CT and DE have distinct types of signals. For CT systems, the signals on all the arcs are continuous time waveforms, while for DE systems, the signals are discrete events. When putting together different domains, the signals at the boundaries must be correctly converted. These conversions are performed by *event generators* and *event interpreters*.

For example, Figure 16 shows a DE system that contains a CT subsystem C. The arcs that pass continuous waveforms are drawn in thicker lines than the arcs passing discrete events. Actors U and V are event interpreters, which have discrete event inputs and waveform outputs. Actor W is an event generator, which converts waveforms to discrete events. A reverse situation is shown Figure 17, where a DE subsystem is contained in the CompositeActor D.

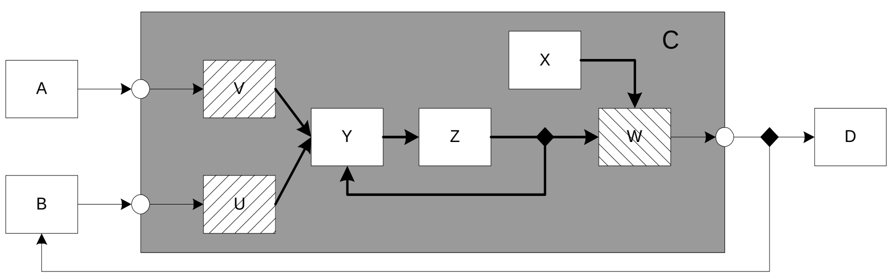


FIGURE 16. A CT subsystem inside a DE system.

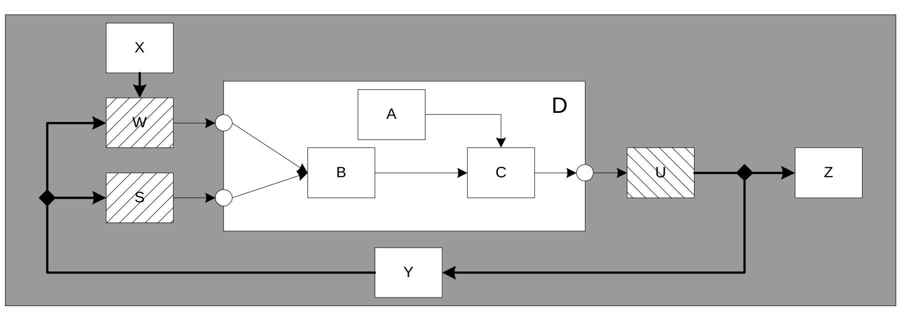


FIGURE 17. A DE subsystem inside a CT system.

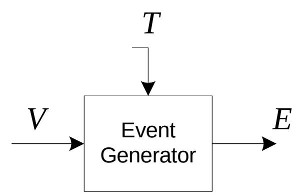


FIGURE 18. A generic event generator.

4.2.1 Event Generators

Event generators are actors that convert continuous waveforms to discrete events. Since a discrete event is defined by a value and a time stamp, the generator must be able to capture both of them.

Following the system formulation (15) of a continuous time system, an event $e(\tau, v)$ is said to *be triggered by the condition*

$$q(x(t), u(t), t) = 0, \tag{30}$$

if $\exists\delta > 0$, such that $\forall\varepsilon$ such that $0 < \varepsilon < \delta$, $q(x(\tau - \varepsilon), u(\tau - \varepsilon), \tau - \varepsilon) \neq 0$, but $q(x(\tau), u(\tau), \tau) = 0$. Condition (30) is called the *trigger condition*. The values of the event is defined by

$$v = r(x(\tau), u(\tau), \tau). \tag{31}$$

In other words, for any fixed input waveform $u(t)$, condition (30) defines a closed set $\Gamma \subseteq \Re^n$, and the events occur at the first time that the state trajectory enters Γ from the outside. Notice that this definition assures that the events are located *discretely* on the time axis. If there is a segment of trajectory $x[t_1, t_2]$ that satisfies (30), then the event is defined to occur at t_1.

An event generator is fully specified by the function pair (q, r). Since the functions q() and r() can be built in the CT domain by a chain of blocks from the outputs of integrators and sources, the event generators can be reduced to a two-input-one-output actor as shown in Figure 18. The input T is the trigger signal, and the input V is the value signal. The generic function of an event generator can be described in the pseudo code as in Figure 19.

```
enabled = true;
for each firing {
   if (T == 0 and enabled) {
      output E(now, V);
      enabled = false;
   } else {
      enabled = true;
   }
}
```

FIGURE 19. Pseudo code for discrete event generators.

In the code, `now` is the current time of the system, and `enabled` is an internal state that rules out the continuity of events. The actor has no output if the trigger condition is not satisfied or it is not enabled.

Notice that since the current time is maintained by CT directors and all the actors can access it without an input signal, some event generators can be simplified if the trigger condition q() depends only on time. These actors are called *time triggered event generators*. Actor `S` in Figure 16 is a time triggered event detector. One of the most useful time triggered event generators is the periodic sampler which has one input V, and a parameter sample period t_s, such that it outputs an event at $\tau = t_0 + nt_s$, where t_0 is the starting time of the simulation, and n is a natural number. If $q()$ depends on the state trajectory, then the event generator is called *level triggered*.

4.2.2 Event Interpreters

Event Interpreters are actors that convert discrete events into a continuous waveform. The critical task for event interpreters is to provide a *default* value at the time points where no events occur. Event interpreters can be application specific. Usually, users may interpret events in the following two forms.

- *Zero-order hold*. The output of a zero-order hold actor is a piecewise constant function. If no events are received after the starting time, the output is zero. Formally, if the sequence of input events is $\{e_1(\tau_1, v_1), e_2(\tau_2, v_2), \ldots, e_M(\tau_M, v_M)\}$, then the output signal $w(t)$ is

$$\begin{aligned} w(t) &= 0, \quad \text{for } t_0 \le t < \tau_1; \\ w(t) &= v_i, \quad \text{for } \tau_i \le t < \tau_{i+1}, i \in \{1, \ldots, M-1\}; \\ w(t) &= v_M, \quad \text{for } \tau_M \le t. \end{aligned} \tag{32}$$

  This is the most commonly used event interpreter, and it is consistent with D/A converters.
- *Impulses*. Some discrete events, like switching capacitors, are better modeled as impulse functions. In this case, if the sequence of input events is $\{e_1(\tau_1, v_1), e_2(\tau_2, v_2), \ldots, e_M(\tau_M, v_M)\}$, then the output signal $w(t)$ is

$$w(t) = \sum_{i=1}^{M} v_i \cdot \delta(t - \tau_i), \tag{33}$$

  where $\delta(\ )$ is the Dirac delta function defined in (23).

Strictly speaking, event generators and event interpreters belongs to neither the CT domain nor the DE domain. Implementing them in the CT domain simplifies the overall software design. There are multiple event-based domains, but none of them handle continuous-time signals.

4.3 Mixed-signal System Execution.

Many requirements for mixed-signal simulation have already been met by the CT directors discussed in the last chapter. For example, the event interpreters can be treated as sources, and the CT system can be scheduled in the same way; the impulse interpretation of events can be handled by the breakpoint IBE solver discussed in section 3.2.3. However, we still need more techniques to handle event generators, and more importantly to coordinate the executions of the two domains.

4.3.1 Detecting Events

The difficulty of generating an event is to locate the event time. If the time is found, to find the value of the event is just a function evaluation. The breakpoint mechanism in the CT directors provides a convenient way to locate an event time.

1. *Time triggered event generators*

Time triggered event generators can know the time at which their next event happens beforehand. This time is an expected breakpoint for the CT directors. Recall that the CT directors will find the behavior of the system at all the breakpoints in the breakpoint table. So the time triggered event generators can just register a breakpoint at its next event time, say t_k. When the current time reaches t_k, it can compute the event value and register the next event time.

2. level triggered event generators

Suppose the function $q(x(t), u(t), t)$ is built by a chain of actors starting from the integrators and source actors. Let $z(t)= q(x(t), u(t), t)$ be the signal that is fed into the trigger input *T* of a level triggered event generator. We want to find $t \in [t_0,t_f]$ such that

$$z(t) = 0. \tag{34}$$

In general, (34) cannot be solved analytically. We will rely on numerical methods to find an approximation. In continuous time simulation, time is discretize into a discrete set *Tc*, where the time points are selected based on the accuracy, convergence, and expected breakpoint concerns. It is unlikely that these time points happen to satisfy (34). However, the state trajectory is continuous when there are no breakpoints. This continuity assures that if two consecutive integration time points t_i and t_{i+1} satisfy

$$z(t_i) \cdot z(t_{i+1}) < 0, \tag{35}$$

then there must exist $\tau \in (t_i, t_{i+1})$ such that $z(\tau) = 0$. From the CT simulation point of view, τ is an unexpected breakpoint. Only when the integration step from t_i to t_{i+1} is finished can the event generator report a "missed" event.

Notice that (35) is only a sufficient condition for detecting an event. It is also possible that although $z(t_i) \cdot z(t_{i+1}) > 0$, there still exists $\tau \in (t_i, t_{i+1})$ that satisfies (35). Figure 20 shows the two cases of level triggered events. The event in (a) is called a *zero crossing event*, which can be detected by (35); the event in (b) is called a *zero touching event*, which can't be detected.

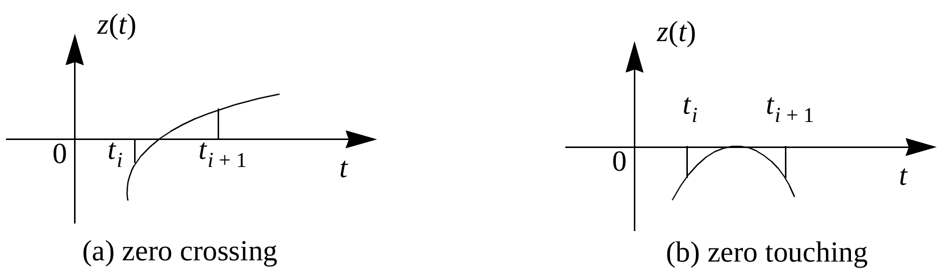


FIGURE 20. Two types of level triggered events.

Appendix A presents some methods to iteratively find τ given (35) based on [19] and [25]. Among them, the Illinois method is considered the best one, and is implemented.

4.3.2 Time Synchronization.

Information hiding requires that a composite actor with internal semantics should obey the semantics of the outer domain at the boundaries. CT and DE are both timed domains, so the tags of signals are both time. When they are combined together, they must share the same notion of the global time. The global time is maintained by the outer domain. The current time of a domain is accessible from the `getCurrentTime`() method. A composite actor with an inside domain may have its own local time. But this local time must be correctly tuned such that the signals at the boundaries follows the semantics of the outer domain. The next two sections study the two possible cases when combining CT and DE domains.

4.3.3 CT inside DE: Rollback

For ease of discussion, we denote by ***Q*** the event queue in the DE domain, and by $e_A(\tau, v)$ a discrete event for actor A at time τ with value *v*. When the event has no value, we write $v = \perp$, so $e_A(\tau, \perp)$ is a pure event. Subscript *A* will be omitted if the destination actor is not important.

When a CT composite actor is embedded in a DE domain, it must behave like a DE actor. That is, it responds to discrete events and produces discrete events. In particular, it must be causal, in the sense that the time stamps of the output events is not earlier than the time stamps of the input events. To satisfy this causality property, the following claim must hold.

Claim 1: If a CT composite actor C with current time t is contained in a DE composite actor D with current time τ, then whenever C is fired, it must be true that:

$$t \geq \tau . \tag{36}$$

Proof: An actor C in the DE domain can be fired at time τ iff there is an event $e_C(\tau, v)$ at the top of $\boldsymbol{Q}$. I.e. all the events that are earlier than τ have been processed. In addition, when a CT actor is executing from t, it may generate events at anytime after t. So suppose C is fired at τ and $t < \tau$, it will continue executing from t (it can't jump to time τ since the semantics requires the local time to be continuous). Let $d = \tau - t$, and $d > 0$. It is possible that C generates an event $e\left(t + \frac{d}{2}, v'\right)$. Since $t + \frac{d}{2} < \tau$, C is not causal. Thus, in order for C to behave like a causal DE actor, (36) must be satisfied.

◆

Claim 1 shows that the CT subsystem must run ahead of the global time. Inequality (36) implies two possibilities:

1. $t = \tau$. This is ideal. C can continue execution and the input event $e_C(\tau, v)$ is converted to a source signal by the corresponding event interpreters.

2) $t > \tau$. This says that C has gone too far the last time, and skipped the input event $e_C(\tau, v)$. I.e. the (last) execution from τ to t was wrong. To correct this error, C must *roll back* to an early state where it can be guaranteed not to miss events. The following claim shows how far should this rollback be performed.

Claim 2: Let $e_C(\tau_1, v_1)$ and $e_C(\tau_2, v_2)$ be two consecutive events for a CT subsystem C, where $\tau_1 \leq \tau_2$. If at the time C is fired, its local time is $t > \tau_2$, then it at most needs to roll back to time τ_1.

Proof: Since the events in $\boldsymbol{Q}$ are ordered by their time stamp, and only the events with the smallest time stamp will be removed from $\boldsymbol{Q}$ to fire its destination actor, when C receives event $e_C(\tau_2, v_2)$, no event with time stamp less than τ_2 can be received after that. That is C can roll back to any time less than or equal to τ_2. Since $\tau_1 \leq \tau_2$, C at most needs to roll back to time τ_1.

◆

Claim 2 implies that C should remember the state at the time of the last input event. This is the state is a "τ^+ state," meaning that the time of the state is the event time τ, and the effect of the event has been taken care of. The IBE solver is exactly suitable for this situation, because it considers the effect of the input signal, and does not advance time.

The ahead-of-time execution of the CT subsystem brings another problem that an event generated may not be a "real" event, especially for level triggered event generators. If there is a skipped input event that causes the CT subsystem to rollback, the potential output event may not happen anymore. So when the CT subsystem C detects an output event $e(t_e, v)$, it should locally cache the event and require a refire at time $\tau = t_e$ (by emitting a pure event $e_C(t_e, \perp)$). Only when the global time has reached t_e can the CT subsystem be sure that $e(t_e, v)$ does actually happen, and it can be emitted. A side effect of this approach is that a CT subsystem always emits events at the current time of the DE system, in other words, if pure events are also treated as input events, then *CT subsystems have zero delay.*

Rollback is a time consuming operation. It implies that certain segments of the state trajectory have to be computed twice. Since the simulation of CT systems is very expensive, we should minimize the possibility of rollbacks. So the question "*how far should the CT subsystem run?*" must be answered carefully. Ideally, the CT subsystem should run to the time of its next input event. But this is not applicable, because the DE system can not provide the next input event time until the event is actually happening. Practically, there are several possible answers:

- *Run until the output event is found.* This is the most aggressive answer. It blindly applies the input/output semantics to an actor. This is unsatisfactory not only because an output event may not actually happen, but also because if no event is detected, the execution control will never return to the outer domain.
- *Run one minimum step*. This is the most conservative answer. It tries to avoid rollback by making the smallest possible progress. But, this approach puts too many pure events into the DE event queue, and looses the advantage of speeding up the CT simulation by adjusting step sizes. In addition, we still cannot guarantee that rollback is not needed. It is always possible that two successive input events have time difference less than the minimum step size, or even that they are simultaneous.
- *Run until the time of the next event in the event queue*. This approach is better than the previous two, since it will always return the execution control to the outer domain, and CT can use larger step sizes. Notice that the next event in the queue may not be the next event for the CT subsystem, so this approach cannot avoid rollback.

In our implementation, all the directors support the query of the next iteration time. The `getNextIterationTime()` method in the DE directors will return the time stamp of the next event currently in the queue. Notice that this time is strictly greater than the current time, but is not the actual next iteration time because some events that are currently processing may produce output events that are earlier than the potential next iteration time. In summary, one firing of a CT subsystem stops if one of the followings is true:

1. A potential output event is generated;
2. The CT current time reaches the next iteration time of the outer domain.

4.3.4 DE inside CT: Breakpoints

The situation is much simpler when a DE subsystem is contained in a CT system, once the event generation problem is solved. The mechanism of breakpoint handling in the CT domain allows the DE subsystem to register its next output event as a breakpoint. Since time is advanced monotonically in CT, and the event generators can only generate events that are at the "current time," the DE subsystem will receive inputs events monotonically in time. In addition, a composition of causal DE actors is still causal (discard the zero delay loops), so the time stamp of the output events is always greater than or equal to the CT current time. That is, the DE subsystem only produces expected breakpoints.

Note that the DE subsystem should not be involved in the process of the fixed point iteration of CT. As stated in section 3.1.2, the fixed point solution is an instantaneous behavior. The configuration of the system should be kept unchanged. The firing of a DE subsystem may change its states. So the DE subsystem can only be fired between two successive integration steps of the CT system.

4.3.5 CTMixedSignalDirector

CTMixedSignalDirector is the director that handles the execution of a CT composite actor when it is interacting with other domains. It implements all the features discussed above to achieve a correct and efficient mixed-signal simulation. It extends the functionality of CTMultiSolverDirector in the following ways:

- initialize(). If the CT composite actor is embedded in another domain, in the initialize phase of the whole simulation, the `CTMixedSignalDirector` will ask for a zero delay refire from its executive director. This will guarantee that the CT subsystem is running ahead of global time.
- prefire(). While the prefire() method of the multi-solver director simply returns true, the prefire() method of the mixed-signal director manages possible rollback. As shown in Figure 21, when the prefire() method is called, it checks if it is embedded in another domain. If so, it compares its local current time t with the current time τ of its executive director. If $t > \tau$, it must roll back. In order to correctly roll back to a previous "known good" state, all the actors that have state (parameters and internal variables) should implement the `CTMemarisActor` interface. The known good state is saved by calling `saveStates()` on all actors with memory, when the director's local current time is equal to the global current time and the input event has been processed. The method `restoreStates`() is called when rollback is performed. The current time is set back to the "*known good time*," the time of the know good state. After rollback, the director can re-execute the system up to the time of the outside current time.

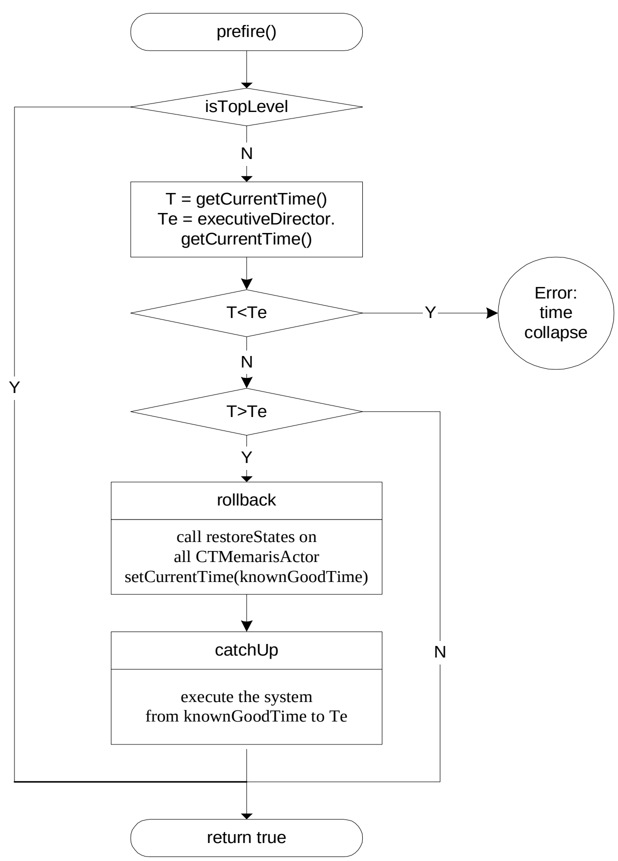


FIGURE 21. The prefire() method of CTMixedSignalDirector.

- fire(). As shown in Figure 22, the fire() method of the director executes in two phases, the event processing phase and the execution phase. In the event processing phase, all the event generators emit their potential event at the current time; all the event interpreters consume input events if there are any. Notice that time is not advanced in the event processing phase. The director will request a zero delay refire from its executive director, if it is not a top level director. If the outer domain is the DE domain, this request will be a pure event that is placed after all other events with the same time stamp. This pure event will be processed after the DE director has processes all other simultaneous events. The consequence is that when the CT composite actor is triggered by this pure event, it will have a better estimation about the next iteration time. In fact, if there is only one CT composite actor in a DE domain, then the CT composite actor will never need to roll back. In the execution phase, the CT system will run until the "fire end time" if it is not at the top level. Otherwise, it will just run one step. The fire end time is by default the next iteration time of the

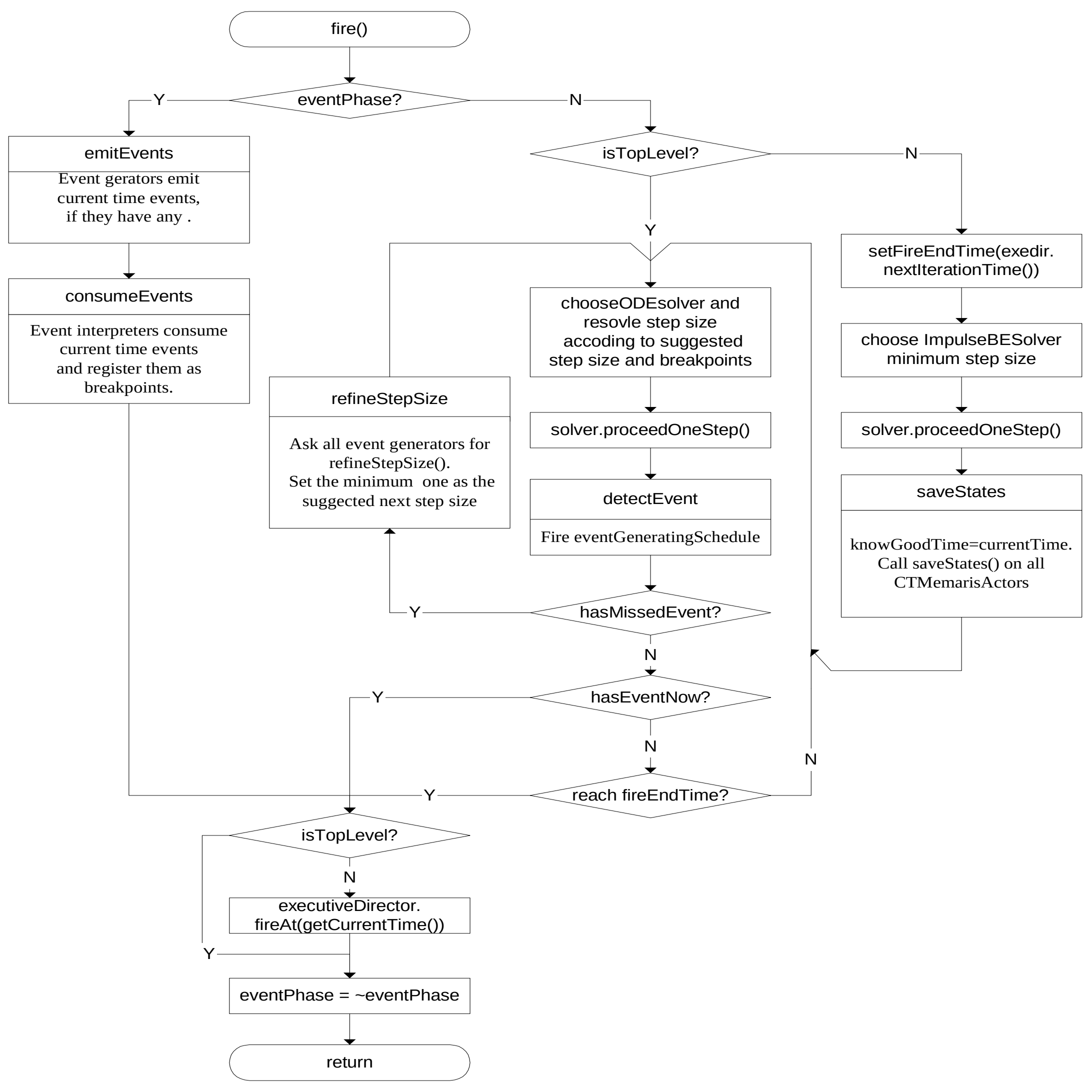


FIGURE 22. THe fire() method for CTMixedSignalDirector.

executive director, but it can be refined if an event is generated during the execution.

CTMixedSignalDirector is the most powerful director in the CT domain. It not only can be used for interactions with other domain, but also can be used to handle unexpected breakpoints. So when users construct a pure CT system with actors that may cause unexpected breakpoint, the CTMixedSignalDirector should be used.

5 Case Study

We explore a mixed signal simulation of a micro accelerometer with digital force feedback. Micro accelerometers are MEMS devices that use beams, gaps, and electrostatics to measure acceleration. Beams and anchors, separated by gaps, form parallel plate capacitors. When the device is accelerated in the sensing direction, the displacement of the beams causes changes in gap sizes, which further causes change in the capacitance. By measuring the change of capacitance (using the Winston capacitor bridge), the acceleration can be obtained accurately.

Feedback can be applied to the beams by charging the capacitors. The benefits of feedback are [7]:

- Reduce the sensitivity to process variations.
- Eliminate mechanical resonances, increase sensor bandwidth
- High selectivity and dynamic range.
- Reduce sensor Brownian motion noise.

Sigma-delta modulation [11], also called the pulse density modulation or the bang-bang control, is a digital feedback technique, which provides the A/D conversion functionality "for free." A design of a micro accelerometer with digital feedback is described in [18].

As shown in Figure 23, the second order CT subsystem is used to model a beam. The voltage on the beam-gap capacitor is sampled every T seconds (much faster than the required output of the digital signal), then filtered by a lead compensator (FIR filter), and fed to an one-bit quantizer. The outputs of the quantizer are converted to force and fed back to the beams. The outputs are also counted and averaged every N*T seconds to produce the digital output. In our example, the external acceleration is a Sin wave.

"Sampler" is a time triggered event generator, which samples the input signal every T seconds and produces events. "ZeroOrderHold" is an event interpreter that translate the input discrete events into piecewise constant CT signals. The simulation result is shown in Figure 24. In the CTPlot, Data0 is the position of the beam; Data1 is the input Sin signal; Data2 is the feedback control force. In the DEPlot, Data0 is the quantization result for every 0.02 second, and Data1 shows the digital output of the device, which is the moving average of Data0 over every 50 samples.

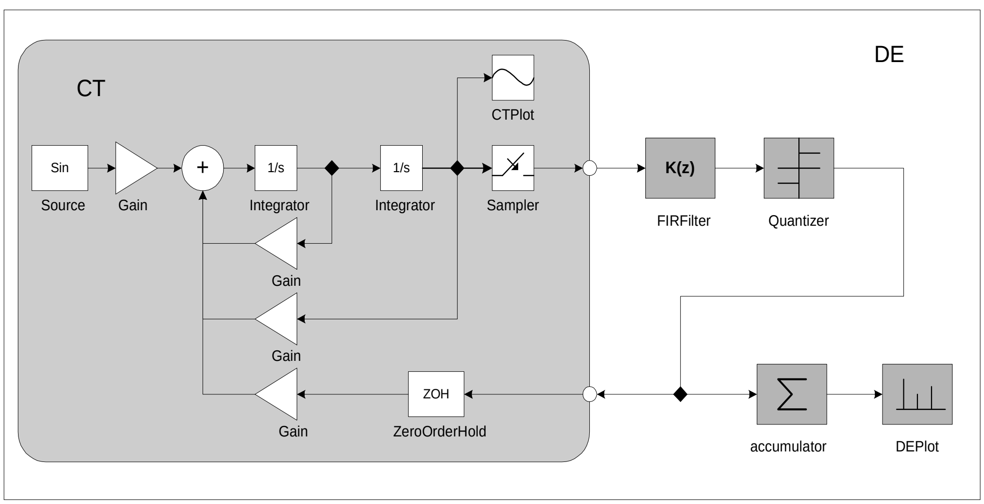
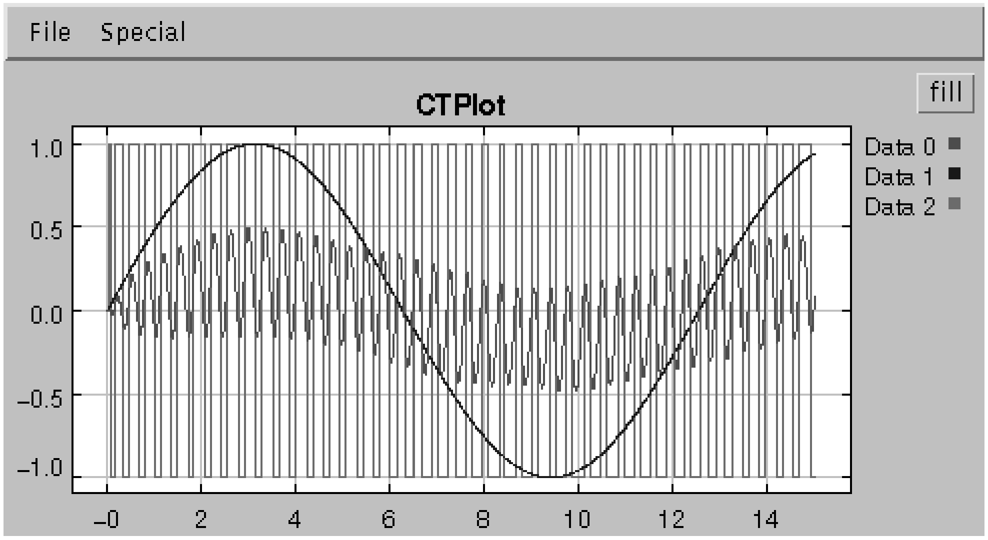
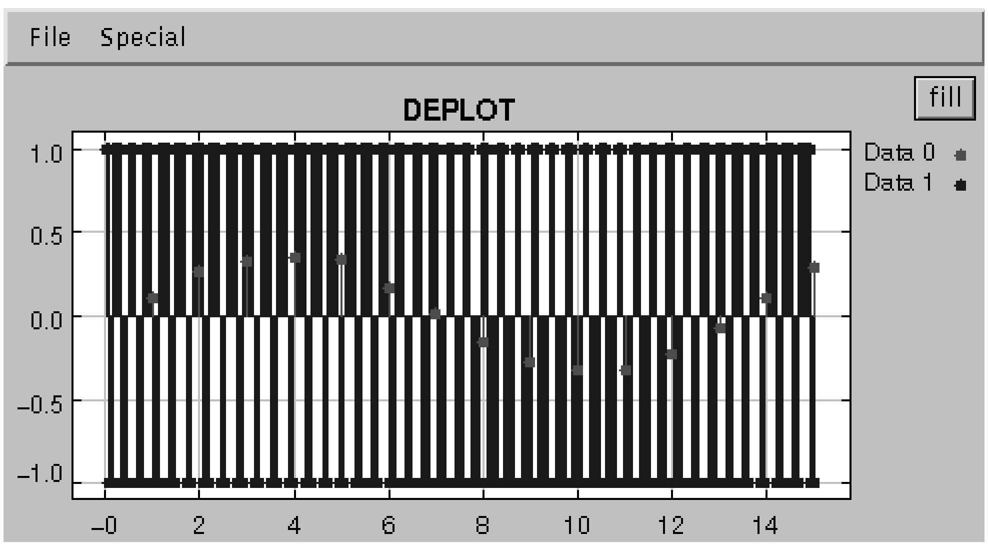


FIGURE 23. The block diagram for the digital feedback micro accelerometer.



(a) Plot of the continuous signals



(b) Plot of the discrete signals

FIGURE 24. The simulation result of the digital feedback micro accelerometer.

6 Conclusion and Future Work

This report presents the techniques of continuous time and mixed signal simulation in Ptolemy II. Continuous time systems that are modeled as ordinary differential equations can be represented in the Ptolemy II CT domain using integrators with feedback loops. The numerical integration methods are implemented using the Ptolemy II execution model and the proper schedules. Breakpoints are treated as an integral part of the CT semantics, and the directors are designed to handle them correctly.

Mixed-signal simulation of continuous time and discrete event model is studied using the tagged signal model. Since the signal types in the two domains are distinct, special actors, like event generators and event interpreters, are needed to convert them. Using the breakpoint mechanism in the CT domain, the event generation problem can be easily solved. The execution coordination of the CT and DE domains are discussed in detail. When CT subsystems are embedded in a DE system, the semantics requires that the CT subsystems must run ahead of time and be able to rollback. A special treatment in the CTMixedSignalDirector can help preventing the CT subsystem from running too far, and thus can minimize the possibility of rollback.

The techniques presented in this report, especially the event detection techniques, the breakpoint mechanism, and the variable step size integration methods are helpful to support other possible heterogeneous simulations. For example, the discrete time (DT) model is a special case of the DE model where the events only happen at equally separated time points. The integration of CT and DT models is widely used to model periodical sampling systems. The simulation of this kind of system should be easy to achieve since CT part can know exactly how far it should run ahead, and rollback is never necessary. Finite state machine (FSM) is a popular model for specifying sequential control logics. The integration of CT and FSM models yields the so called “hybrid system” model. Our event detection techniques can generate trigger events from CT that cause state transitions in the FSM part, but we need more study on how the FSM can change the CT system configuration by its “actions”. The integration of the simulation of CT, DT, and FSM will be future work.

7 References

[1] H. Anton, “Calculus with Analytic Geometry,” 2nd ed. Jone Wiley & Sons, 1984.

[2] P. Antsaklis, W. Kohn, A. Nerode, and S. Sastry (Eds.), “Hybrid Systems II,” Springer Verlag, Lecture Notes on Computer Science 999, 1995.

[3] P. J. Ashenden, “The Designer’s Guide to VHDL,” Morgan Kaufmann Publishers, 1996.

[4] E. L. Acuna, J. P. Dervenis, A. J. Pagones, F. L Yang, and R. A. Saleh, “Simulation Techniques for Mixed Analog/Digital Circuits,” IEEE Journal of Solid-State Circuits, Vol. 25, No. 2, April. 1990. pp. 353-363.

[5] F. Balarin, M. Chiodo, etc., “Hardware-Software Co-Design of Embedded Systems, the POLIS Approach,” Kluwer Academic Publisher, 1997.

[6] J. Banks, J.S. Carson II, and B. L. Nelson, “Discrete-Event System Simulation,” Prentice Hall, 1996.

[7] B. E. Boser, “Electrostatic Force-Feedback,” talk slides, http://knwloon.eecs.berkeley.edu/~boser/pdf/feedback.pdf

[8] V. Bryant, “Metric Spaces,” Cambridge University Press, 1985.

[9] J. T. Buck, S. Ha, E.A. Lee, and D. G. Messerschmitt, “Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems,” Int. Journal of Computer Simulation, special issue on “Simulation Software Development,” Vol. 4, pp. 155-182, April, 1994.

[10] F. M. Callier and C. A. Desoer, “Linear System Theory,” Springer-Verlag, 1991.

[11] James C. Candy, “A Use of Limit Cycle Oscillations to Obtain Robust Analog-to-Digital Converters,” IEEE Trans. on Communications, Vol. COM-22, No. 3, March 1974, pp298-305.

[12] B. Carlson and V. Gupta, “Hybrid cc with Interval Constraints,” Lecture Notes in Computer Science (1386), Hybrid Systems: Computation and Control, Springer 1998, pp. 80-95.

[13] W.-T. Chang, S. Ha, and E. A. Lee, “Heterogeneous Simulation-- Mixing Discrete-Event Models with Dataflow,” invited paper, Journal on VLSI Signal Processing, Vol. 13, No. 1, Jan. 1997.

[14] IEEE DASC 1076.1 Working Group, “VHDL-A Design Objective Document, version 2.3,” http://www.vhdl.org/analog/ftp_files/requirements/DOD_v2.3.txt

[15] S. Kowalewski, M. Fritz, H. Graf, J. Preubig, S. Simon, O. Stursberg, and H. Treseler, “A Case Study in Tool-Aided Analysis of Discretely Controlled Continuous Systems: the Two Tanks Problem,” 5th Int. Workshop on Hybrid Systems, Notre Dame, 1997, Hybrid Systems V, Lecture Notes in Computer Science, Springer, 1998.

[16] E. A. Lee, “Modeling Concurrent Real-time Processes Using Discrete Events,” UCB/ERL Memorandum M98/7, March 4th 1998.

[17] E. A. Lee and Alberto Sangiovanni-Vencentelli, “A Framework for Comparing Models of Computation,” ERL Memorandum UCB/ERL M97/11, University of California, Berkeley, CA 94720, Jan. 1997.

[18] M. A. Lemkin, “Micro Accelerometer Design with Digital Feedback Control,” Ph.D. dissertation, University of California, Berkeley, Fall 1997.

[19] C. Moler, “Are We There Yet? Zero Crossing and Event Handling for Differential Equations,” Matlab News & Notes: Simulink2 Special Edition, The Mathworks Inc., 1997, pp. 16-17.

[20] A. R. Newton and A. L. Sangiovanni-Vincentelli, “Relaxation-Based Electrical Simulation,” IEEE Trans. on Electron Devices, Vol. ed-30, No. 9, Sept. 1983.

[21] C. L. Phillips and H. T. Nagle, "Digital Control System Analysis and Design," 3rd ed. Prentice Hall, 1995.

[22] W. H. Press, S.A. Teukolsky, W. T. Vetterling, and B. P. Flannery, "Numerical Recipes in C: the Art of Scientific Computing," Cambridge University Press, 1992.

[23] The Ptolemy group, "Ptolemy II Design Document," http://ptolemy.eecs.berkeley.edu/ptolemyII

[24] R. A. Saleh and A. R. Newton, "Mixed-Mode Simulation," Kluwer Academic Publishers, 1990.

[25] L. F. Shampine, R. C. Allen Jr., and S. Pruess, "Fundamentals of Numerical Computing," John Wiley & Sons, Inc., 1997

[26] L. F. Shampine and M. W. Reichelt, "The Matlab ODE Suite," SIAM J. Sci. Compute. vol. 18, No. 1, pp. 1-22, Jan. 1997.

[27] D. E. Thomas, "The Verilog Hardware Description Language," Kluwer Academic Publishers, 1998.

[28] J.Vlach and A. Opal, "Modern CAD Methods for Analysis of Switched Networks," IEEE Trans. on Circuits and Systems-I: Fundamental Theory and Applications, Vol. 44, No. 8, Aug. 1997.

Appendix A: Numerical Root Finding Techniques

Part of the materials in this section are compiled from [25] and [19]. The general statement of the problem is:

Problem 1. To find $x \in \Re$, such that

$$f(x) = 0 \tag{37}$$

where *f*( ) is a function that is continuous and "smooth".

Definition: *Multiplicity of the root*: If $\exists g(x)$, such that

$$f(x) = (x - \alpha)^m g(x), \tag{38}$$

and $g(\alpha) \neq 0$, then α is a *m* multiple root of *f*( ).

If we examine the derivative of *f*( ) near α, we can find the difference of an even multiplicity root and an odd multiplicity root, as shown in Figure 24

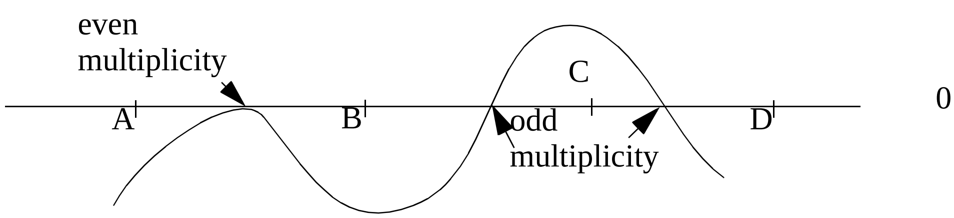


FIGURE 25. Roots of even and odd multiplicity.

It can be seen that a zero with even multiplicity is very hard to detect, especially on a finite precision digital computer. So when facing the root finding problems, sometimes we can only say "as good as it gets."

So in practice, the problem is reformulated as:

Problem 2. Given a closed interval [B, C] with *f*(B)*f*(C)< 0, find the root $M \in [B, C]$, such that |*f*(M)|<ε, for some small ε>0.

Notice that if *f*(B)*f*(C)>0, there may be:

- no zeros in [B,C]
- odd number of even zeros, like the interval [A, B] in Figure 24.
- even number of odd zeros, like the interval [B, D] in Figure 24.

The basic idea of solving Problem 2 is to shrink the interval and keep the *f*(B)*f*(C)< 0 property until some point M is found such that |*f*(M)|<ε. Here are some generally used methods.

A1.Bisection.

This is the most naive and popular method. The method is shown below.

```
if (f(B)*f(C)<0) {
    M = B + (B - C)/2.0
    if (abs(f(M))<eps) {
        return M;
    } else {
        if (f(M)f(B)<0) {
            C = M;
```

```
                redo;
            } else {
                B = M;
                redo;
            }
        }
    }
```

Although the algorithm looks like a blind search, it is very robust and the convergence is guaranteed as long as the function is continuous. The problem is that this method may converge very slowly.

A2.Newton's Method and the Secant Method

These methods do not need an interval of opposite signs. They use a first order (linear) approximation of the function f near 0, as shown in (39), and try to recursively solve the linear equations (40) using (41)

$$f(x) \approx f(x_i) + (x - x_i)f'(x_i) \tag{39}$$

$$f(x_i) + (x - x_i)f'(x_i) = 0 \tag{40}$$

$$x_{i+1} = x_i + \frac{f(x_i)}{f'(x_i)}. \tag{41}$$

Since it is inconvenient to calculate $f'(x_i)$, it can be approximated by linearly extrapolating the old points, x_i and x_{i-1}, to get:

$$x_{i+1} = x_i - f(x_i) \cdot \frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})}. \tag{42}$$

This is called the *secant* method. The advantage of this class of methods is that it converges fast. The downside is that if f is "flat," i.e. $f(x_i)$ and $f(x_{i-1})$ are very close to each other, then the extrapolation becomes very coarse. This method is not easy to scale to the multidimensional case, where f is a vector function.

A3.Regular Falsi (False Position)

This is the method that is used in `fzero.m` in the Matlab. The method is similar to the bisection method. But, instead of finding the midpoint of B and C, it uses the linear interpolation of the values at B and C. As shown in Figure 25, for a convex function, this method will use one "old" value again and again, and it converges slowly.

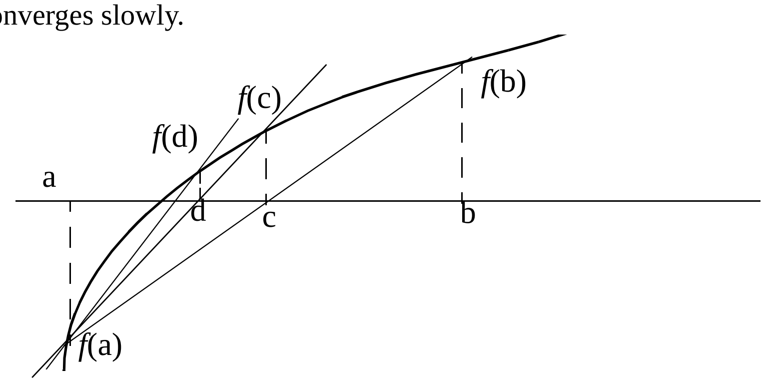


FIGURE 26. An illustration of the false position method.

A4.Illinois Algorithm

This method combines the regular falsi and the bisection method. As shown in Figure 26, instead of bisection in *x*, it bisection on *f*(*x*) when *x* is reused in the regular falsi method.

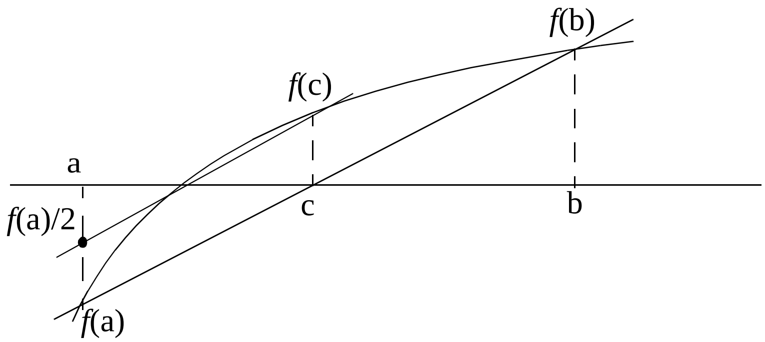


FIGURE 27. An illustration of the Illinois algorithm.

“This method shares the reliability that the bisection algorithm obtains by staying in an ever shrinking interval, but avoids the slow convergence that plagues the method of false position. And it can be vectorized.”[19]