Operational Semantics of Hybrid Systems

Haiyang Zheng

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2007-68
http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-68.html

May 18, 2007

Operational Semantics of Hybrid Systems

by

Haiyang Zheng

B.S. (University of Science and Technology of China) 1999
M.S. (University of California, Berkeley) 2006

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Engineering - Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Edward A. Lee, Chair
Professor Alberto L. Sangiovanni-Vincentelli
Professor David M. Auslander

Spring 2007

The dissertation of Haiyang Zheng is approved.

Chair	Date
	Date
	Date

University of California, Berkeley

Spring 2007

Operational Semantics of Hybrid Systems

by

Haiyang Zheng

Abstract

Operational Semantics of Hybrid Systems

by

Haiyang Zheng

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Edward A. Lee, Chair

Hybrid systems are heterogeneous systems that include continuous-time (CT) subsystems interacting with discrete-event (DE) subsystems. They are effective models for physical systems interacting with software or experiencing discrete mode changes.

This dissertation discusses an interpretation of hybrid systems as executable programs written in a programming language with a hybrid system semantics. The semantic properties of such a programming language affect our ability to understand, execute, and analyze a hybrid system model. This dissertation focuses on a few semantic issues that come in defining such a programming language, such as the interpretation of discontinuities in CT signals and simultaneous discrete events in DE signals, liveness property, and the consequences of numerical ODE solver techniques.

The interactions between CT and DE subsystems and between DE subsystems themselves are captured by discontinuities in continuous-time signals and simultaneous discrete events in discrete-event signals. In order to precisely represent them in compute execution results, a two-dimension domain, called "super-dense time," is used as the domain for defining signals. This domain allows a signal to have multiple values at the same time point while keeping the values ordered.

CT and DE subsystems are modeled as actors, which are functions that map a set of signals to another set of signals. In this way, a hybrid system model is just a network of actors interacting via signals. We can always transform a network of actors into a composite actor with feedback, where the function of the composition actor is the composition of functions of the component actors. The

least fixed point solution to the function of the composite actor, which is a set of signals, gives the denotational semantics of the hybrid system model.

The operational semantics takes the denotational semantics as a mathematical foundation and defines a set of rules for evaluating actors such that the least fixed point solution can be constructed. Rather than constructing the whole signals, the operational semantics only computes a discrete subset of the signals called a discrete representation of the signals. The constructive procedure is formalized with the Abstract State Machine semantics, where a hybrid system is treated as a state transition system and the rules specify how state transformations are performed.

This operational semantics supports heterogeneous and hierarchical composition of different models of computation, such as CT, DE, finite state machines, and synchronous languages, and modular execution of the composition as a whole. This ability makes it easy to jointly model and design software controlled systems.

The operational semantics proposed in this dissertation has been implemented in HyVisual, which is a software tool for modeling and simulating hybrid systems. HyVisual is part of the Ptolemy II software framework, which is available in open-source form at http://ptolemy.org.

Professor Edward A. Lee
Dissertation Committee Chair

To my parents and extended family.

Contents

List of Figures

List of Tables

Acknowledgements

I am extremely grateful to my advisor, Professor Edward A. Lee. He led me into the research area of hybrid systems and embedded systems. His consistent support, inspiration, and guidance make this dissertation finally become a reality. At the most difficult times of my life, he supported and encouraged me like a family member. I am deeply indebted to him.

I would like to thank Professor Alberto Sangiovanni-Vincentelli and Professor David M. Auslander for serving on my qualifying exam and dissertation committees, and for providing valuable comments to my research.

I would like to thank all the colleges who contributed to this dissertation. The Zeno chapter is a collaborative work with Aaron D. Ames. His passion on Zeno hybrid systems greatly motivated the work. The causality interface section describes a collaborative work with Ye Zhou. The original idea of this work was developed in a joint class project with her. Many ideas of this dissertation result from numerous discussions with Stephen Neuendorffer, Xiaojun Liu, Adam Cataldo, Eleftherios Matsikoudis, Jie Liu, and Jonathan Sprinkle. It is great pleasure to work with Elaine Cheong, Yang Zhao, Yuhong Xiaong, Thomas Huining Feng, Gang Zhou, and Man-kit Leung. I would also like to thank Christopher Brooks, Mary Stewart, Mary Margaret Sprinkle, and Tracey Richards for their excellent support.

No words can express how grateful I am to my parents, Dao Zheng and Xu Wang, my grandparents, Aiming Wang and Zhihui Sun, my younger brother, Qinwen Zheng, and the extended family. Their unconditional love and support make me what I am today. Weining's love makes my life wonderful. Emily's laughter makes my life full of joy. Hanjing's care and support of the past many years cannot be forgotten. This dissertation is dedicated to them.

Chapter 1

Introduction

Hybrid systems are heterogeneous systems that include continuous-time (CT) subsystems interacting with discrete-event (DE) subsystems. They are effective models for physical systems interacting with software or experiencing discrete mode changes. Typically, the CT dynamics is modeled by differential equations, while the DE dynamics is modeled by finite state machines or DE systems. Transitions between states and discrete events in DE systems are used to describe discrete mode changes of dynamic systems and actions taken by software subsystems.

Most of the major contributions in hybrid systems have been in the construction of system theories, theories of control, and analysis and verification tools (see for example [34, 48, 56, 71, 81, 72]). A few software tools have been built to support such analytical methods, such as Charon [2, 3], CheckMate [85], d/dt [11], HyTech [49], Kronos [32], Uppaal [57], and a toolkit for level-set methods [74]. In addition, some software tools provide simulation of hybrid systems, including Charon [2], Hysdel [92], HyVisual [63, 21], Modelica [91], Scicos [35], Shift [33], HyBrSim [76], and Simulink/Stateflow (from the MathWorks) [30]. An excellent analysis and comparison of these tools is given by Carloni, et al. [26].

This dissertation focuses on the modeling and simulation tools, but takes the perspective that hybrid systems are *executable* programs written in a programming language with a hybrid system semantics. The theme of this dissertation is to develop an operational semantics for interpreting such a programming language. The emphasis is one of correctness, preciseness, modularity, and predictable and understandable behavior, rather than one of accurate approximation of unachievable

behavior. Of the above tools, Shift and Modelica probably come closest to reflecting this philosophy, since they are consistently presented as programming languages more than as simulation tools.

The view of hybrid systems as executable computational artifacts was stimulated by the DARPA-funded MoBIES project (model-based integration of embedded software), which undertook the challenging task of establishing an interchange format for hybrid systems. The goal was to facilitate exchange of models and techniques between tools. The effort was led by the key proponents of model-integrated computing [87], the developers of Charon, CheckMate, and HyVisual, and intensive users of Simulink/Stateflow. The result of this work was a formalism called HSIF (hybrid system interchange format) [90]. The proposals for the next generation of interchange format can be found in [25, 80].

One of the key objectives of HSIF, that of model exchange among diverse tools, was at odds with another of its key objectives, that of defining an executable and complete hybrid systems semantics. The diverse tools represented by the HSIF community, in fact, have significant differences in their semantics, often reflecting their differing objectives (e.g. verification vs. simulation).

In this dissertation, we set aside the concern for interchange of models among tools, but focus instead on defining a *clean* and *complete* hybrid systems semantics. One objective is to completely define behaviors, including subtle corner cases, in the spirit of giving a complete semantics for a programming language. The other objective of this operational semantics is to support *heterogeneity* and *modularity*.

Hybrid systems are semantically heterogeneous, combining continuous dynamics, periodic timed actions, and asynchronous event reactions. Modeling and design of such heterogeneous systems is challenging. A number of researchers have defined concurrent models of computation (MoCs) that support modeling, specification, and design of such systems [23, 61, 43, 54, 51]. Typical and most commonly used MoCs for modeling hybrid systems include synchronous/reactive (SR) systems [16, 15], discrete-event (DE) systems [58, 94, 39, 27], and continuous-time (CT) dynamics [75, 41, 91, 62].

A variety of approaches have been tried for dealing with the intrinsic heterogeneity. Ptolemy Classic [23] introduced the concept, showing useful combinations of asynchronous models based on variants of dataflow and timed discrete-event models. The concept was picked up for hardware design in SystemC (version 2.0 and higher) [86], on which some researchers have specifically built heterogeneous design frameworks [51]. Metropolis [43] introduced communication refinement as a

mechanism for specializing a general MoC in domain-specific ways, and also introduced "quantity managers," which provide a unified approach to resource management in heterogeneous systems.

Several authors advocate unified MoCs as a binding agent for heterogeneous models [13, 44, 20]. Heterogeneous designs are expressed in terms of a common semantics. Some software systems, such as Simulink from the MathWorks, take the approach of supporting a general MoC (continuous-time systems in the case of Simulink) within which more specialized behaviors (like periodic discrete-time) can be simulated. The specialized behaviors amount to a *design style* or *design pattern* within a single unified semantics. Conformance to design styles within this unified semantics can result in models from which effective embedded software can be synthesized, using for example Real-Time Workshop or DSpace.

Our approach is different in that the binding agent is an abstract semantics [60]. By itself, it is not sufficiently complete to specify system designs. Its role is exclusively as a binding agent between diverse concrete MoCs, each of which is expressive enough to define system behavior (each in a different way).

We propose a particularly useful way to combine the SR, DE, and CT systems, providing a disciplined and rigorous mixture. In particular, subsystems can be modeled using any of the three semantics, and these subsystem models can be combined hierarchically to form a whole. This approach leverages the idea of abstract semantics to provide a coherent and rigorous meaning for the heterogeneous system, and the principles of synchronous/reactive languages to improve the modularity of conventional DE and CT semantics. These improvements facilitate arbitrary heterogeneous combination of the three distinct modeling styles and modular execution of the subsystems as a whole.

The operational semantics proposed in this dissertation has been implemented in a software package called HyVisual [21, 63], which is a subset of the Ptolemy II software framework [59, 22]. The HyVisual software package is available in open-source form (BSD-style license) at http://ptolemy.org.

1.1 Overview of the Dissertation

The structure of the dissertation is as follows. In this chapter, we will study two examples to illustrate the necessity of a precise and unambiguous representations of discontinuities of continuous-time

(CT) signals and simultaneous events of discrete-event (DE) signals, and point out the limitations and difficulties of the current common practice.

Chapter 2 provides a *super-dense time* [73] as a solution and gives precise definitions of CT and DE signals using the Tagged Signal Model [61].

Chapter 3 studies *actors*, where interactions of signals are conducted, and defines a set of typical actors used in hybrid systems. A few practical mechanisms are given for generating discontinuities in CT signals and simultaneous events in DE signals.

Chapter 5 studies some properties of compositions of actors, including the existence, uniqueness, and liveness properties of behaviors. An interface theory is developed to abstractly represent *causality* of actors and allow algebraic compositions of causality interfaces for composite actors.

Chapter 5 proposes an operational semantics based on the denotational semantics given in Chapter 4. This operational semantics describes in details the incremental construction of discrete representations of signals and the evaluation of signal values by leveraging the synchronous languages principles and abstract actor semantics. This chapter formalizes the operational semantics with the Abstract State Machine.

Chapter 6 gives a technique for simulating Zeno hybrid systems beyond their Zeno time points. This technique gives a systematic way to complete the system dynamics by introducing *post-Zeno* states, and provides a practical simulation strategy for constructing approximations of Zeno signals. This technique allows executions to *jump over* Zeno time points and proceed with the dynamics defined in the post-Zeno states. In this way, simulation of a Zeno hybrid system model can go beyond its Zeno time point and therefore reveal the complete dynamics of the system being modeled.

The last chapter summarizes the main results and contributions of this dissertation and discusses some future work.

1.2 Sticky Masses Model

We start by considering a fairly typical hybrid system example shown in Figure 1.1 that we can use to frame the discussion. The model is deliberately small and simple, making it easier to discuss semantic issues without the distracting complexity of a more "real-world" example. The figure shows the visual syntax of HyVisual [28], which is implemented within the Ptolemy II software framework

[59]. The reader should not be misled by the visual syntax. While visual syntaxes are commonly used for models that approximate real systems, they can also be used as a programming language syntax, in which case the model *is* the real system (the program), in the same sense that the text of a C program is the program. We nonetheless call a visual program like that in Figure 1.1 a "model" because calling it a "program" would confuse too many readers who assume that programs must have textual syntaxes.

The model in Figure 1.1 is of a physical system consisting of two masses on springs that oscillate.[1] When the masses collide, they stick together with an exponentially decaying stickiness. When the differential force of the springs exceeds the stickiness, the masses come apart. The three-dimensional rendition of the physical system shown in Figure 1.2 is a snapshot of an animation created using the Ptolemy II graphics infrastructure [40]. The top-level of the hierarchy in Figure 1.1 shows a continuous-time model, where boxes represent actors and connections between them represent continuous-time signals. The `Masses` block encapsulates the spring-masses model. The other three blocks are plotters.

Figure 1.3 shows the next level of the hierarchy, which contains a finite-state machine with an (unimportant) initial state and two states representing the two modes of operation. Since states of this state machine represent modes of operation, we use the terms "state" and "mode" interchangeably for them. In the `Separate` mode, the masses are separately oscillating, and in the `Together` mode, they are stuck together. The behaviors in the `Separate` mode and the `Together` mode are specified in Figure 1.4 and Figure 1.5 respectively. Each mode is given by a signal-flow block diagram representing the ordinary differential equations that model the dynamics. These diagrams are called *refinements* of their corresponding modes.

The traces of an execution are shown in Figure 1.6, where it can be seen that the masses start with separated positions, come together and collide, oscillate together for a short time, come apart, then again collide and come apart. The three plots, produced by the three plotter blocks at the top level in Figure 1.1, represent the positions, velocities, and accelerations of the two masses as a function of time. The `Masses` block in Figure 1.1 produces as outputs the positions of the two masses (*p1* and *p2*), their velocities (*v1* and *v2*), and their accelerations (*a1* and *a2*).

The state machine diagram in Figure 1.3 shows the mode logic. The state machine starts in the `Init` state, which has a single outgoing transition with guard expression "`true`."[2] This

[1] This model was studied by Liu [66] and was inspired by microelectromechanical accelerometers [65].

[2] In the HyVisual syntax, each mode transition is annotated with two lines of text, where the first line is the

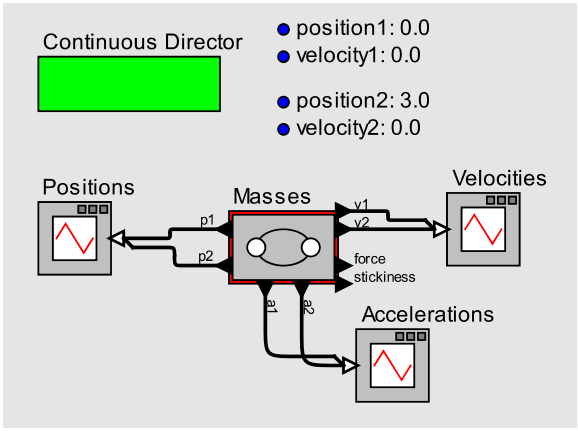

Figure 1.1. A hybrid system of two masses on springs.

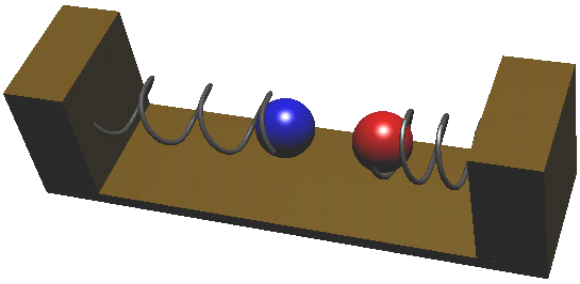

Figure 1.2. The three-dimensional rendition of the physical system shown in Figure 1.1. This is a snapshot of an animation created using the Ptolemy II graphics infrastructure [40].

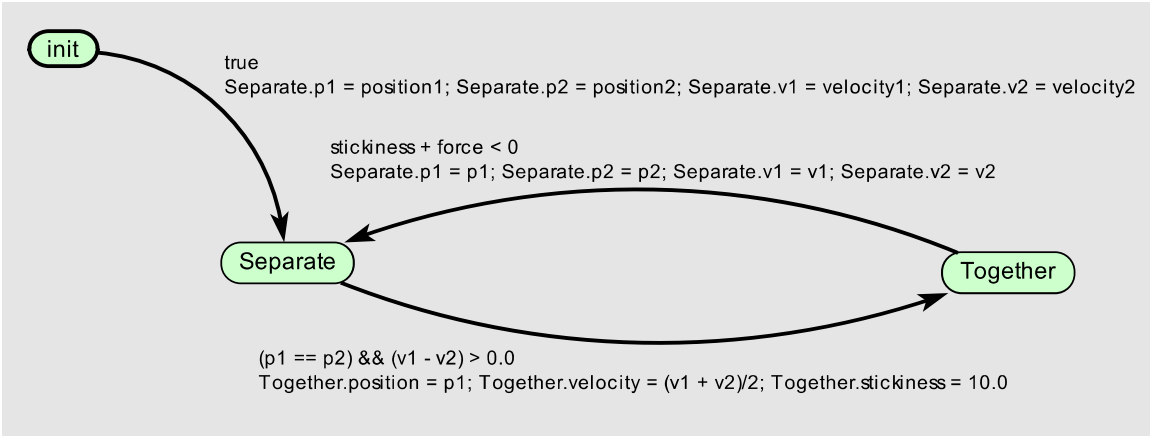

Figure 1.3. The mode controller of the hybrid system in Figure 1.1.

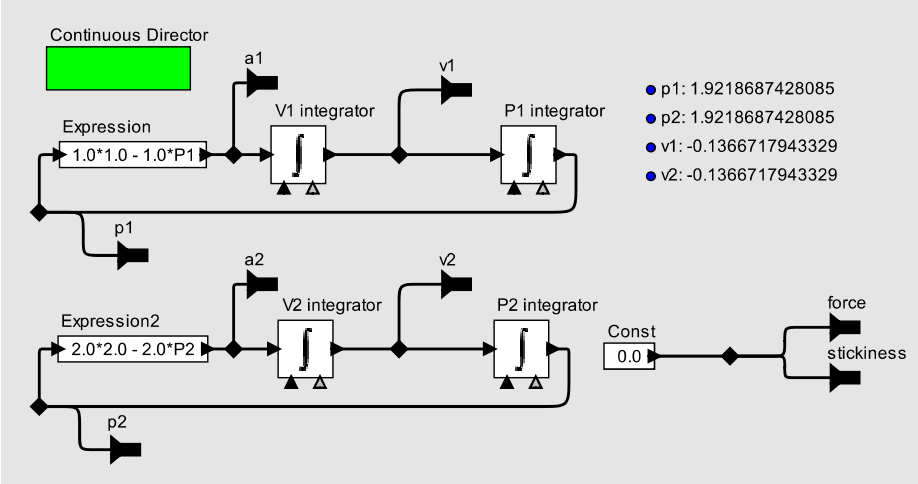

Figure 1.4. The refinement of the `Separate` mode in Figure 1.3.

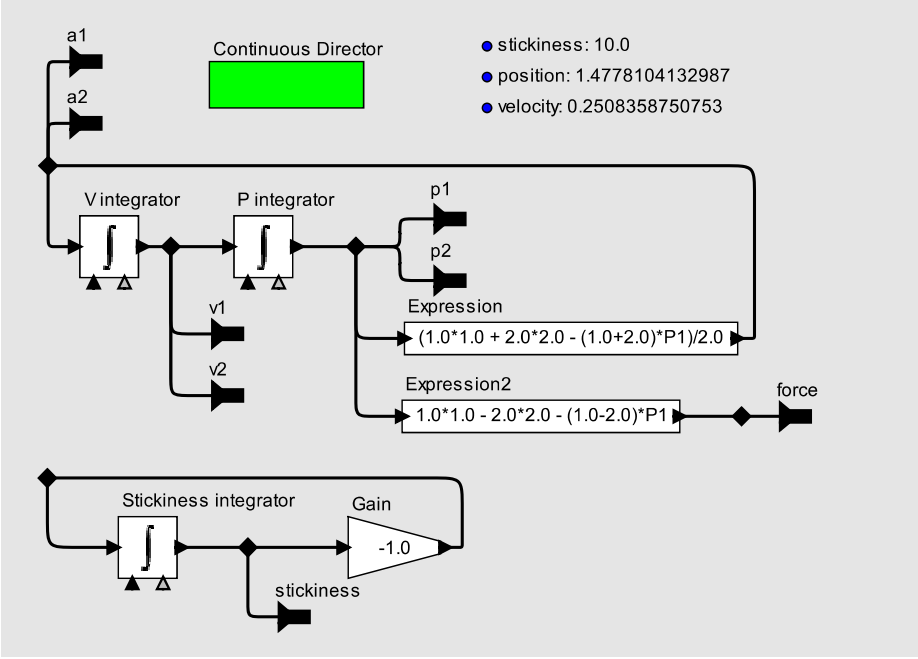

Figure 1.5. The refinement of the `Together` mode in Figure 1.3.

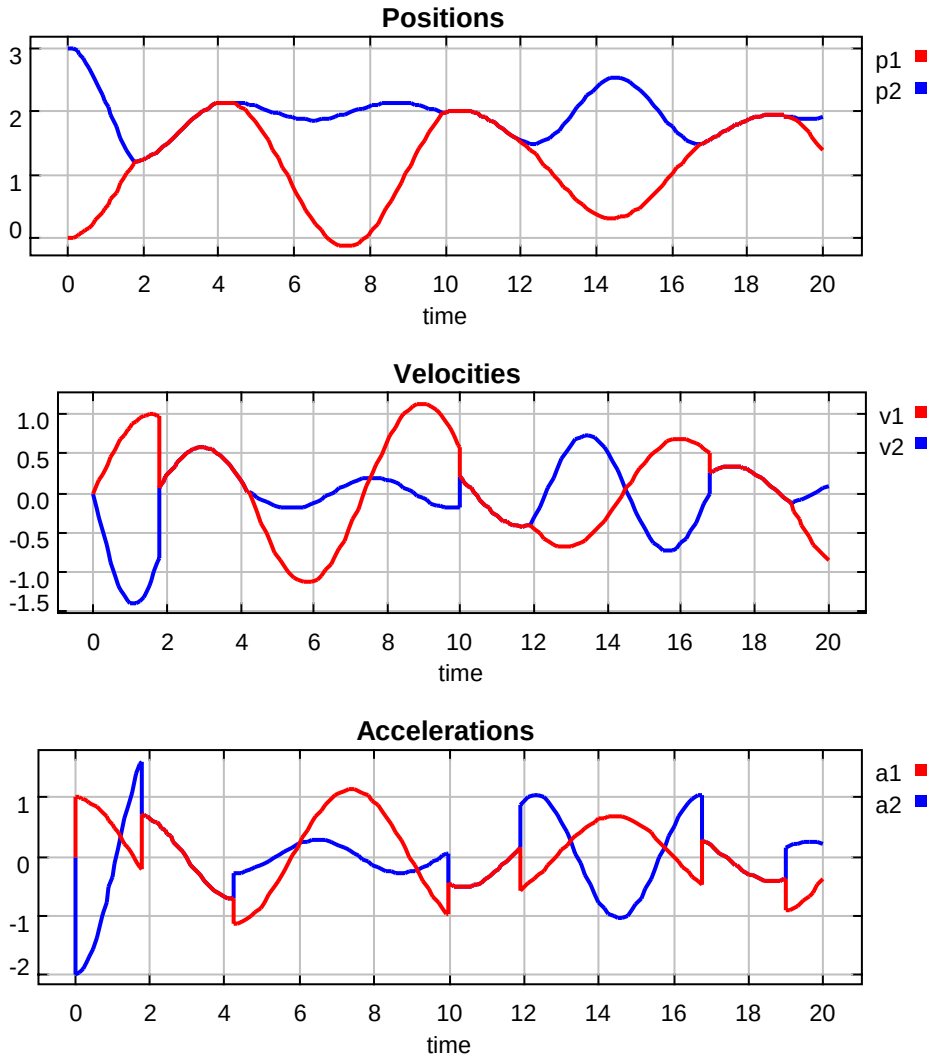

Figure 1.6. The plots resulting from executing the hybrid systems model in Figure 1.1.

guard expression evaluates to true, so the transition is taken immediately, and the action expression (immediately below the guard expression) is executed. This action expression initializes the positions and velocities in the destination mode, `Separate`.

The state machine remains in the `Separate` mode until the guard on its outgoing transition becomes true. The guard expression is "`(p1 == p2) && (v1 - v2) > 0`," which becomes true when the two masses collide. At that point, the state machine transitions to the `Together` mode. The action (shown in the figure immediately below the guard) sets the position and velocity of the (now joined) masses in the destination mode, and also initializes the stickiness. The velocity in the destination mode is set to "`(v1 + v2)/2`," which, assuming the two masses are the same, implements the law of conservation of momentum.

The state machine will remain in the `Together` mode until the guard on its outgoing transition becomes true. That guard expression is "`stickiness + force < 0`," which becomes true when the force pulling the masses apart exceeds the stickiness. The action on the transition again initializes the positions and velocities of the masses in the destination mode, `Separate`.

In HyVisual, when a guard expression becomes true, the transition must be taken immediately. This is consistent with the physics being modeled in this spring-masses example. Many hybrid system formalisms, however, define a guard expression on a transition as an enabler. Rather than requiring that the transition be taken, it simply permits the transition to be taken. In a simulator, however, this typically results in the transition to be taken at an arbitrary time after the guard becomes true. In simulation, the time at which the transition is taken is typically dependent on the step-size control algorithm of the ODE (ordinary differential equation) solver. For this example, that behavior would be inappropriate. Such hybrid system formalisms associate with each state an *invariant*, which like a guard is a predicate. When the invariant becomes false, a transition out of the state must be taken. In such a formalism, the spring-masses example would be expressed by a combination of invariants and guard expressions that would achieve the same effect.

The system is depicted schematically in Figure 1.7. The physics of this problem is quite simple if we assume idealized springs. Let $p_1(t)$ denote the right edge of the left mass at time t, and $p_2(t)$ denote the left edge of the right mass at time t, as shown in Figure 1.7. Let n_1 and n_2 denote the neutral positions of the two masses, i.e. when the springs are neither extended nor compressed, so the force is zero. For an ideal spring, the force at time t on the mass is proportional to $n_1 - p_1(t)$

`guard`, a predicate that determines when the transition is taken, and the second line is the `action`, a set of statements executed when the transition is taken.

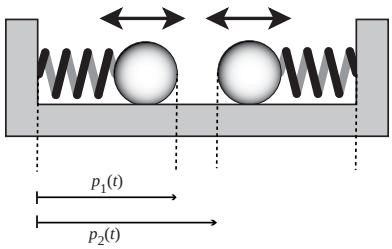

Figure 1.7. A schematic illustration of the system that is modeled in Figure 1.1.

(for the left mass) and $n_2 - p_2(t)$ (for the right mass). The force is positive to the right and negative to the left.

Let the spring constants be k_1 and k_2, respectively. Then the force on the left spring is $k_1(n_1 - p_1(t))$, and the force on the right spring is $k_2(n_2 - p_2(t))$. Let the masses be m_1 and m_2 respectively. Now we can use Newton's law, which relates force, mass, and acceleration, $f = ma$. The acceleration is the second derivative of the position with respect to time, which we write $\ddot{p_1}(t)$ and $\ddot{p_2}(t)$ respectively. Thus, as long as the masses are separate, their dynamics are given by

$$\begin{aligned} \ddot{p_1}(t) &= k_1(n_1 - p_1(t))/m_1 \\ \ddot{p_2}(t) &= k_2(n_2 - p_2(t))/m_2. \end{aligned}$$

If we integrate both sides twice, we get

$$\begin{aligned} p_1(t) &= \int_{t_0}^{t} \left(\int_{t_0}^{\alpha} \frac{k_1}{m_1}(n_1 - p_1(\tau))d\tau + v_1(t_0) \right) d\alpha + p_1(t_0) \\ p_2(t) &= \int_{t_0}^{t} \left(\int_{t_0}^{\alpha} \frac{k_2}{m_2}(n_2 - p_2(\tau))d\tau + v_2(t_0) \right) d\alpha + p_2(t_0) \end{aligned}$$

The two equations above define the dynamics of the refinement in Figure 1.4, where it is assumed that $k_1 = 1$, $m_1 = 1$, $n_1 = 1$, and $k_2 = 2$, $m_2 = 1$, $n_2 = 2$. The initial values $p_1(t_0)$, $p_2(t_0)$, $v_1(t_0)$ and $v_2(t_0)$ are the initial states of the integrators in the figures, which are set by the actions upon entering the mode.

With the masses stuck together, they behave as a single object with mass $m_1 + m_2$ and positions $p_1(t) = p_2(t)$. This single object is pulled in opposite directions by two springs. Let

$$p(t) = p_1(t) = p_2(t),$$

the dynamics are then given by

$$\ddot{p}(t) = \frac{k_1 n_1 + k_2 n_2 - (k_1 + k_2)p(t)}{m_1 + m_2}.$$

Again we can integrate both sides twice to get the relation represented by the refinement of the `Together` mode shown in Figure 1.5.

The initial velocities of two masses are given by the following equations that implement the law of conservation of momentum, where v_1 and v_2 are the velocities before the collision, and v_1' and v_2' are the velocities after the collision, k is the restitution coefficient of collision.

$$\begin{aligned} v_1' &= \frac{(m_1 - km_2)v_1 + m_2(1+k)v_2}{m_1 + m_2}, \\ v_2' &= \frac{m_1(1+k)v_1 + (m_2 - km_1)v_2}{m_1 + m_2}. \end{aligned}$$

In the above example, this collision is perfectly inelastic with the restitution coefficient $k = 0$. With $m_1 = m_2$, we have

$$\dot{p}(t) = \frac{\dot{p_1}(t) + \dot{p_2}(t)}{2},$$

where $\dot{p}(t)$ is the common initial velocity of both masses.

1.2.1 Discussion of the Sticky Masses Example

The most notable feature of our example model, and the one which distinguishes it most from other "programs," is the *continuous evolution* of its "variables." In the visual syntax of HyVisual, the lines connecting blocks (sometimes called "wires" in analogy with circuit diagrams) represent variables of the program. In a corresponding textual syntax, these variables would be given names and referred to by name. In a visual syntax, however, there is usually no need to name them, since their users can simply connect to them. Whereas in a textual syntax "scoping rules" would limit the visibility of such variables, in a visual syntax like HyVisual, visibility is limited by the constraints on wiring in the diagram, for example that the wires cannot cross levels of the hierarchy. In HyVisual, to make variables visible across levels of the hierarchy, we use named "ports." In Figure 1.4 and Figure 1.5, the ports labeled `p1`, `p2`, `v1`, `v2`, `a1`, `a2`, `force`, and `stickiness` are the inside view of the same ports with the same names in Figure 1.1. These ports represent the continuously evolving variables representing position, velocity, and acceleration of the masses, plus the force pulling them apart and the stickiness holding them together.[3]

The continuous evolution of the values of such variables, of course, is what presents the greatest challenge to a programming language designer, since continuous evolution of variables is outside the

[3]We use the term "continuously evolving" for signals whose values evolve continuously rather than in discrete steps. We do not require continuously evolving signals to be continuous. We will make this more precise below.

domain of discourse of today's computers. Thus, while a denotational semantics for a hybrid systems language might embrace continuous evolution of the variable values, an operational semantics can only define values at *discrete* points in time. It is the relationship between such a denotational semantics and operational semantics that is one of the principal topics of this dissertation.

One solution: Uniform Sampling

One solution to this conundrum is to simply disallow continuous evolution. We can invoke sampling theory to assert that any continuously evolving signal (with finite bandwidth) can be sampled uniformly at a sufficiently high rate without loss of information. Indeed, some of the tools mentioned above (notably Hysdel [92] and Shift [33]) operate only on models that have been discretized by sampling by the programmer. This greatly simplifies the programming language semantics, since now the semantics of the model easily matches well-known techniques for synchronous concurrent programming languages such as the synchronous/reactive languages [16]. The problem is that even an example as simple as our spring masses violates the finite bandwidth assumption. As shown in Figure 1.6, the velocities and accelerations both have discontinuities that imply infinite bandwidth. In hybrid system modeling, these discontinuities are the principle subject of study, so a failure to properly represent them is a serious omission.

Another Solution: Non-Uniform Sampling

We can do better than uniform sampling with non-uniform sampling, where we include the points of discontinuity in the samples. However, this is not quite enough. Non-uniform sampling, by itself, is not sufficient to unambiguously represent discontinuities.

Continuous signals exhibit an intrinsic robustness under discretization. Mathematically, the continuously evolving variables of Figure 1.6 are typically represented as functions of the form

$$x\colon T \to \mathcal{R}^n,$$

where T (called the *time line*) is a connected subset of the reals, $\mathcal{R}$, and $\mathcal{R}^n$ is a normed vector space consisting of n-tuples of real numbers with some norm. This function is continuous at $t \in T$ if for all $\epsilon > 0$, there exists a $\delta > 0$ such that for all τ in the open neighborhood $(t - \delta, t + \delta) \subset \mathcal{R}$

$$|x(t) - x(\tau)| < \epsilon.$$

This means that if we examine the value of the signal at a point in time, and if the signal is continuous at that point in time, then small errors in the time at which we examine it result in small errors in the value.

In a computational setting, signal values may have data types significantly different from $\mathcal{R}^n$, in which case, if the set of data values form a topological space, then the topological form of continuity provides similar robustness.

However, signals in hybrid systems are not typically continuous at all points in time. Specifically, let $D \subset T$ be a discrete subset[4] of T. A signal is piecewise continuous if it is continuous at all points in $T \backslash D$, where D is some discrete subset of T, and where the backslash represents set subtraction. However, this leaves open the question in an operational semantics about how to represent the signal at or near points in D.

A typical approach in mathematical modeling of hybrid systems is to define signals to be *continuous on the right* at points in D. A function $x\colon T \to \mathcal{R}^n$ is continuous on the right at $t \in T$ if for all $\epsilon > 0$, there exists a $\delta > 0$ such that for all τ in the interval $[t, t + \delta)$

$$|x(t) - x(\tau)| < \epsilon.$$

This makes explicit the non-robustness of piecewise continuous signals. It is straightforward to generalize this to topological spaces rather than normed vector spaces, so that the same argument may be applied to other data types than $\mathcal{R}^n$.

An operational semantics must somehow represent that a signal value infinitesimally before some $t \in D$ is significantly different from the value at t. Unfortunately, no discretized rendition can properly represent this.

To make this concrete, assume that we seek an operational semantics for an execution of a hybrid system on a computer. This semantics can represent continuously evolving signals only on a discrete subset of real-valued times. Let $D' \subset T$ be the discrete subset of the reals where it will explicitly represent signal values. We can require that the points of discontinuity D be in this set, or $D \subset D'$. However, how can we choose D' to represent the discontinuity? Suppose $t \in D$. Then, since D' is discrete,[5] there is a $t' \in D'$ where $t' < t$ and there is no $\tau \in D'$ such that $t' < \tau < t$. We say that t' *immediately precedes* t. Since $t' < t$, there is a non-zero interval between the samples

[4] A discrete subset is a subset for which there exists an order embedding to the integers [58]. Note that "discrete" is a stronger condition than "countable."

[5] The existence of an order embedding to the integers is essential to this argument [58]. Countable sets would not be sufficient.

that span the discontinuity. Given only the discrete samples, therefore, the discontinuous signal is fundamentally indistinguishable from a continuous signal that simply changes sufficiently rapidly. This is not splitting hairs. It means that an operational semantics based on discrete samples cannot unambiguously represent discontinuities.

In addition to semantic difficulties, this ambiguity creates practical problems for numerical ODE solvers. Variable step solvers typically adjust the spacing between sample points to be smaller where signals are varying rapidly and larger where they are varying more smoothly. With this ambiguity, such solvers must be made explicitly aware of the discontinuities or they will be forced to reduce step sizes down to resolution tolerances before giving up and deciding that the variability represents a discontinuity.

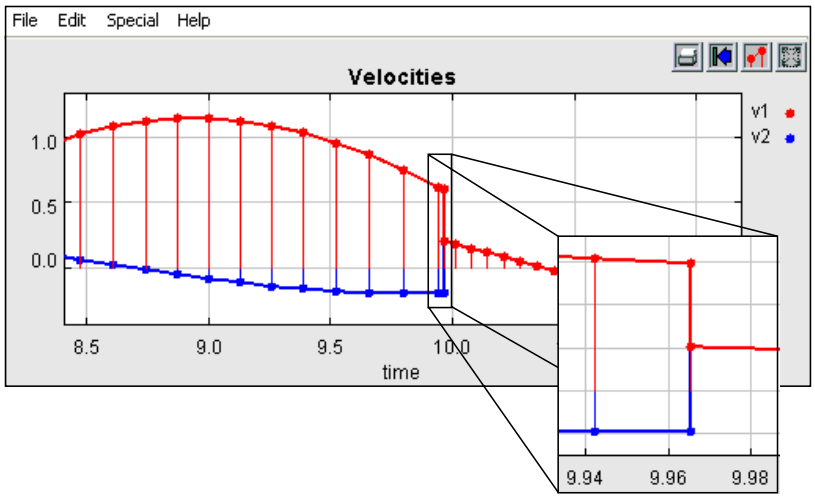

Figure 1.8. A portion of the plot of velocities in Figure 1.6, showing multiple values at one time.

The key problem here is the form of the function

$$x\colon T \to \mathcal{R}^n.$$

Whereas this form works well in a mathematics that embraces the continuum of $\mathcal{R}$, it fails in the formal framework of computing, where continuums are not directly manageable. Figure 1.8 shows a portion of the velocities plot from Figure 1.6 where at time approximately 9.965 the masses collide. The plot shows a dot for each computed value of the velocities, showing the discretization that is not evident in Figure 1.6. At time 9.965, the two velocity signals have more than one value. They have both the value just prior to the collision and the value just after the collision. Having two values at one time is semantically unambiguously distinct from having two distinct values closely spaced

in time. But it requires augmenting the mathematical model for signals. We will come back to this later.

1.3 Newton's Cradle Model

Now we consider another slightly complicated hybrid system example, which models the Newton's cradle, an apparatus with three (or more) balls hanging from strings as shown in Figure 1.9. We name the balls 1 to 3 from left to right. This example is inspired by a one dimensional version in [75].

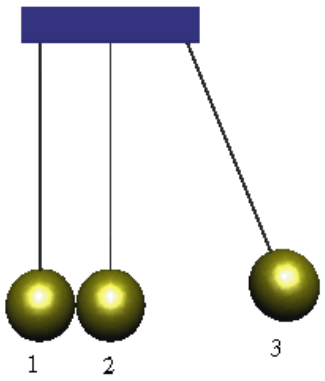

Figure 1.9. The Newton's cradle example.

When these balls move in the same plane, they may collide with their adjacent neighbors. We make two assumptions to make our analysis easier.

Assumptions

1. A collision between balls happens *instantaneously*.
2. Two and only two balls are involved when a collision happens.

The first assumption states that a collision takes *zero* time and will be treated as a *discrete event*. The reason for the second assumption is that there is no general analytic formula to resolve the velocities after collision for collisions involving more than two bodies. Another way to explain this assumption is that a collision happens instantaneously and it is so fast that no other bodies are involved. In order to study the *emergent* behavior resulting from multi-body collision, a common practice is to iterate and examine possible collisions between any pair of bodies until no more

collisions can be found. Obviously, this method may result in different behaviors depending on the examination order of the ball pairs. Nevertheless, we choose this strategy to model the Newton's cradle system. We put aside the question how close this modeling strategy complies with reality but instead focus on how to define a programming language that can construct models that we are interested in.

Figure 1.10 shows a hybrid system model for the Newton's cradle system shown in Figure 1.9. At the top-level of hierarchy in Figure 1.10, the model is a continuous-time model, where blocks represent actors and connections between them represent continuous-time signals. The `ModalModel` block encapsulates the dynamics of three balls and their interactions. The `positions` and `velocities` blocks are plotters. The `Graphic Animation` block generates the animation of ball movement as shown in Figure 1.9. The list of parameters on the left specifies the geometry sizes of the balls and their initial states. The initial states include the initial angle away from the equilibrium position, specified by the parameter `initialTheta`, and initial velocity, specified by the parameter `initialTheta_dot`. In this example, the first ball starts at an angle of "`-PI/8`" with a "`0.0`" velocity and the other balls are at rest.

The `BallClass` block is a *class block* that defines the dynamics of a ball moving as a pendulum under gravitational force as shown in the schematic graph in Figure 1.11. This dynamics is given by the signal-flow block diagram representing a second-order differential equation shown in Figure 1.11. Note that we assume a simple and ideal dynamics here without considering damping, masses of strings, etc.

Figure 1.12 shows the next level of the hierarchy of the `ModalModel` block, where a finite state machine (FSM) contains an (unimportant) `init` state and `run` state. Comparing to the sticky masses example in the previous section, where there are two modes of operation, `Separate` and `Together`, this FSM has only one operation mode specified by the `run` state. However, the `run` state is associated with two outgoing transitions that come back to itself. This makes the dynamics of this model a little bit more complicated but more interesting.

Before we examine the details of transitions, we first look inside the `run` state. A refinement is associated with the `run` state as shown in figure 1.13, which contains three instances of the `BallClass` defined in Figure 1.11. Each instance is configured with a set of parameters listed on the left side. Only instances of a class block but not the class block itself are involved in execution.

Now let us get back to the finite state machine shown in Figure 1.12. The transition from the

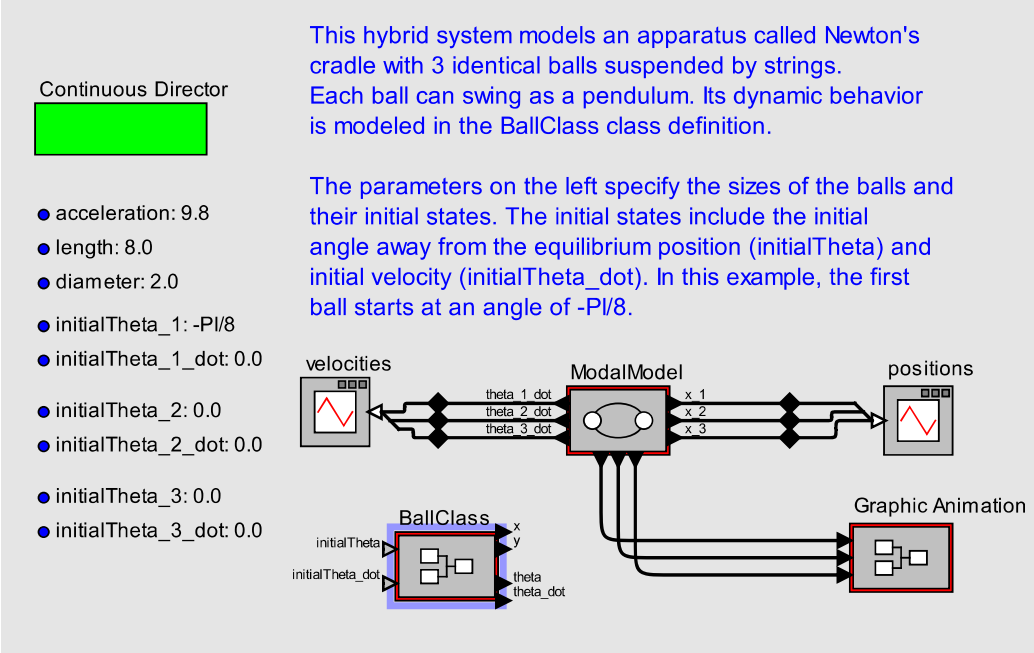

Figure 1.10. The Newton's cradle example as a hybrid system.

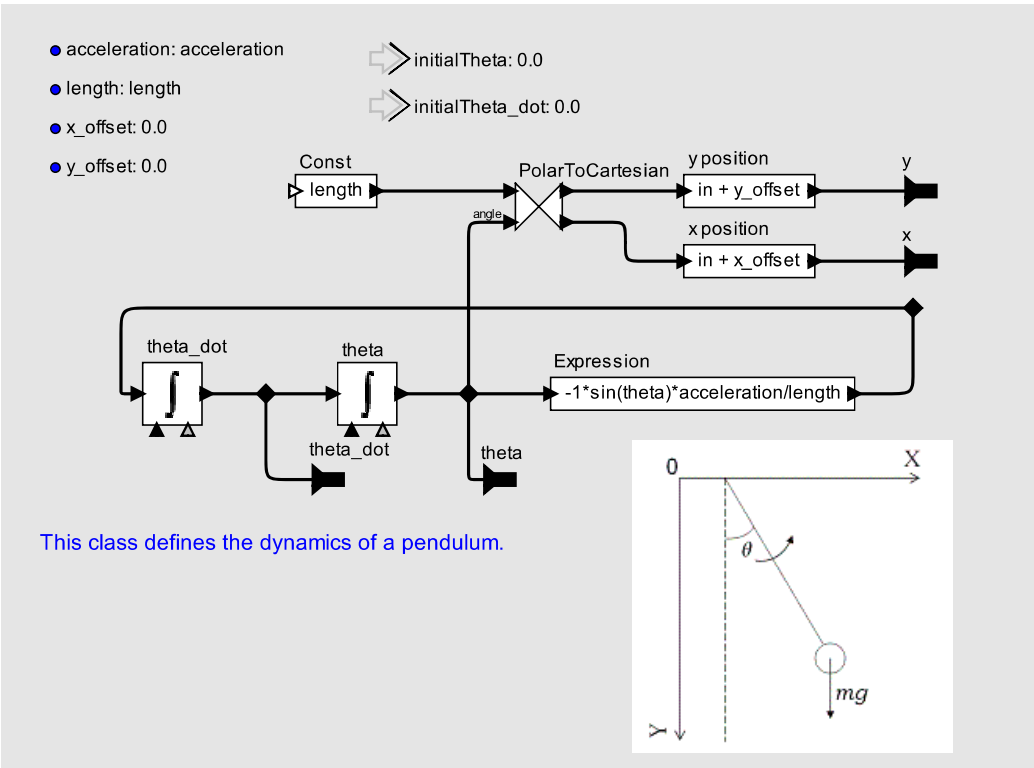

Figure 1.11. The inside details of the `BallClass` definition in Figure 1.10.

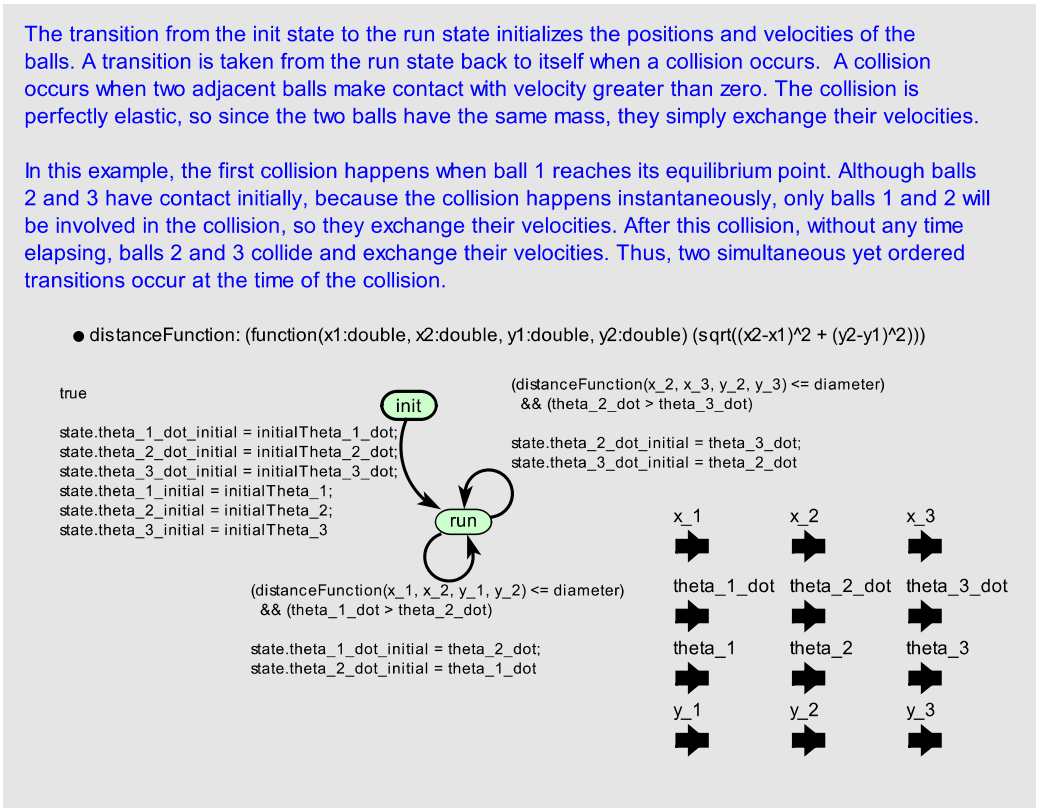

Figure 1.12. The refinements of the modal model of the hybrid system in Figure 1.10.

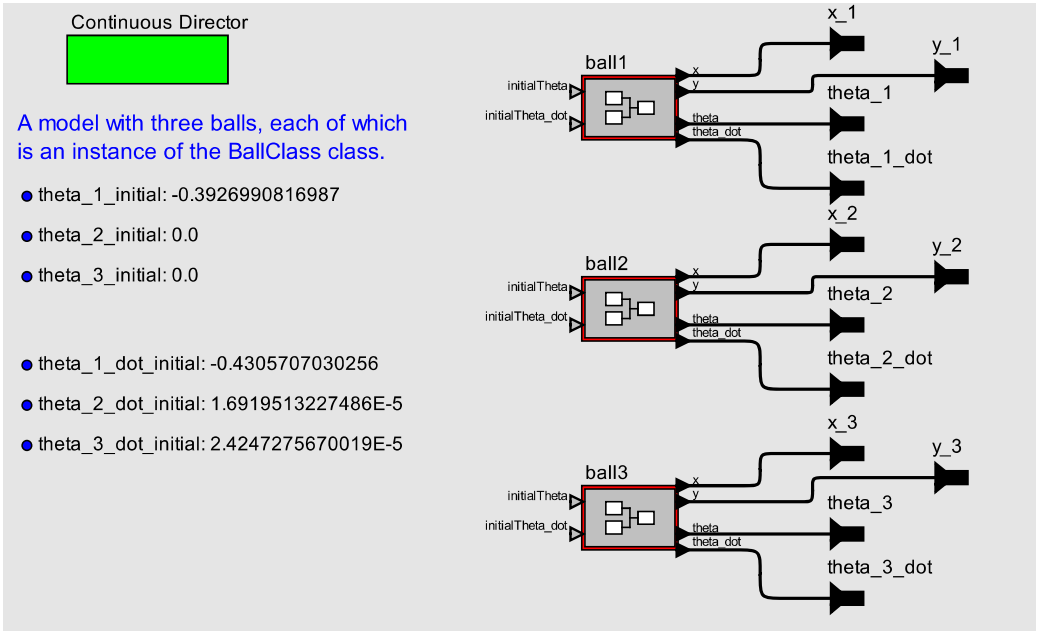

Figure 1.13. The refinements of `run` state of the modal model in Figure 1.12. The three balls are the instances of the `BallClass` defined in Figure 1.11.

`init` state to the `run` state initializes the positions and velocities of the balls. A transition is taken from the `run` state back to itself when a collision occurs. A collision occurs when two adjacent balls make contact and their relative velocity is greater than zero. When the distance between two balls is less then the diameter of balls, there is contact. The distance is measured by a *function* called `distanceFunction`, defined as a *parameter* in Figure 1.12:

$$function(x_1 : double, x_2 : double, y_1 : double, y_2 : double)$$
$$(sqrt((x_2 - x_1)^2 + (y_2 - y_1)^2))).$$

Recall that the equations for calculating the velocities of two balls after a collision are the following, where k is the restitution coefficient of collision, v_1 and v_2 are the velocities before the collision, and v_1' and v_2' are the velocities after the collision,

$$\begin{aligned} v_1' &= \frac{(m_1 - k * m_2) * v_1 + m_2 * (1 + k) * v_2}{m_1 + m_2}, \\ v_2' &= \frac{m_1 * (1 + k) * v_1 + (m_2 - k * m_1) * v_2}{m_1 + m_2}. \end{aligned}$$

If the collision is perfectly elastic, meaning the restitution coefficient $k = 1$, then we have

$$\begin{aligned} v_1' &= v_2, \\ v_2' &= v_1, \end{aligned}$$

meaning that two balls simply exchange their velocities.

If the collision is perfectly inelastic, meaning that the restitution coefficient $k = 0$, then we have

$$v_1' = v_2' = \frac{v_1 + v_2}{2},$$

meaning that both balls move together at the same velocity.

In the model shown in Figure 1.12, we assume the collisions are perfectly elastic, i.e., $k = 1$.

Now let us study the dynamics of transitions. The initial conditions are that the first ball is moved away from its equilibrium position with an angle of -PI/8 and the other two balls are at rest. Therefore, the first collision happens between ball 1 and 2 when ball 1 reaches its equilibrium point. This corresponds to the *lower* transition in Figure 1.12. Note that although balls 2 and 3 have contact initially, their relative velocity is zero, so there is no collision between them. According to the 2nd assumption, only balls 1 and 2 will be involved in the collision and they exchange their velocities.

After this collision, ball 2 gains some non-zero velocity, which makes the relative velocity between ball 2 and ball 3 non-zero. Therefore, without any time elapsing, balls 2 and 3 collide and exchange their velocities. This corresponds to the *right-upper* transition in Figure 1.12.

In summary, two *simultaneous* yet *ordered* transitions occur at the time of the collision.

After the above two simultaneous collisions, ball 1 and ball 2 are at rest and ball 3 starts swinging. When ball 3 comes back and hits the ball 2, the analysis of the collision dynamics is similar to what we had discussed before. Essentially, ball 3 first collides with ball 2, then ball 2 collides with ball 3, and then ball 1 starts swinging with ball 2 and 3 at rest. Again, there are two simultaneous collisions.

Note that when ball 1 swings to the left most position, the whole model basically regains its initial states. Because there is no energy loss during swinging and collisions, the above dynamics will keep repeating for ever.

The traces of balls for an execution are shown in Figure 1.14 and Figure 1.15, where Figure 1.14 shows positions of balls in the horizontal (x) dimension and the Figure 1.15 shows the angular velocities.

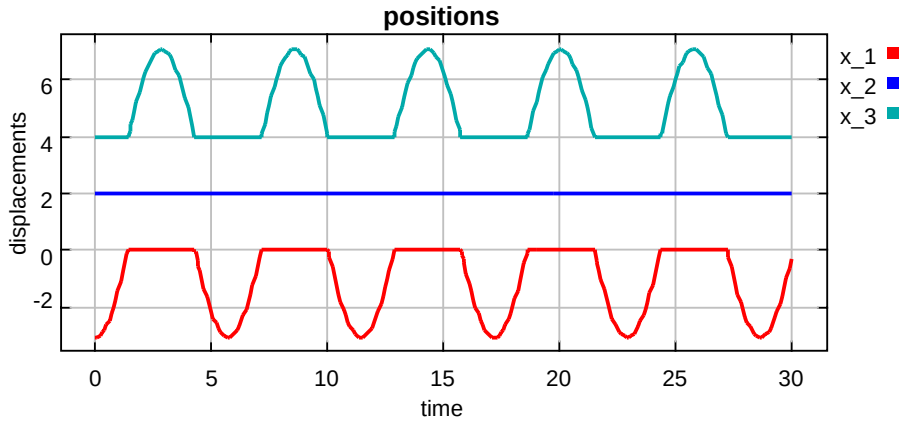

Figure 1.14. The position plots resulting from executing the hybrid systems model in Figure 1.10 with perfect elastic collisions.

From the position figure, it can be seen that ball 1 starts with separated positions, comes to and collides with the ball 2, and then stops moving. However, ball 2 does not move at all and ball 3 instead starts moving. When ball 3 swings back, it collides with ball 2 and stops moving. Then ball 1 starts moving again. This whole process repeats while ball 2 does not move at all.

The plots in Figure 1.15 show how the angular velocities of individual balls change. It is easy to

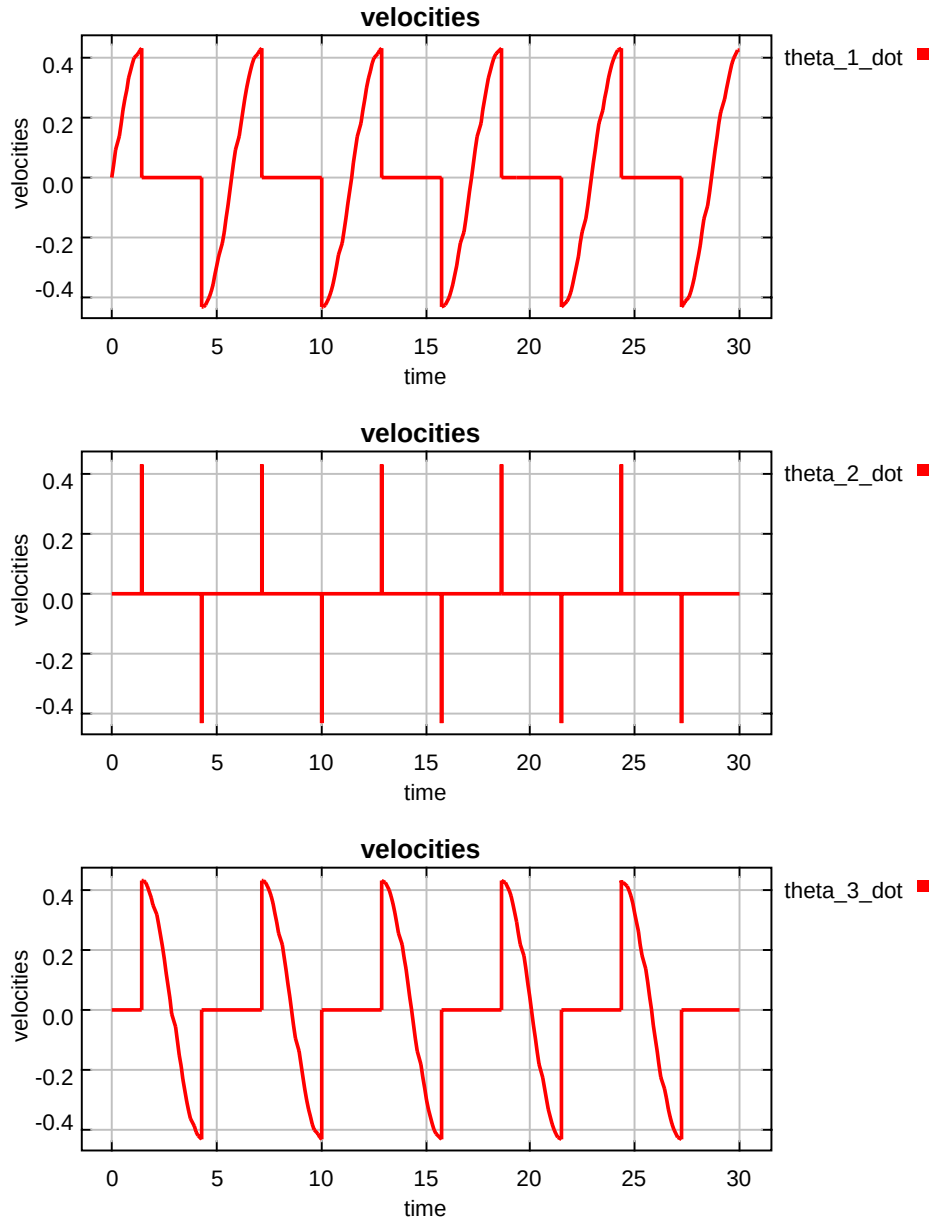

Figure 1.15. The velocities plots resulting from executing the hybrid systems model in Figure 1.10 with perfect elastic collisions.

see that ball 1 and ball 3 swing alternately as time passes by. There are some glitches in the velocity plot for ball 2. We examine these glitches in detail next.

Figure 1.16 shows a zoomed portion of the velocity plot of ball 2 in Figure 1.15 at time approximately 1.433. This is the time when ball 1 collides with ball 2. The plot shows a dot for each computed value of the velocities, showing the discretization that is not evident in Figure 1.15.

The velocity of ball 2 is 0.0 just prior the collision. The non-zero value of the velocity corresponds the effect of the collision between ball 1 and 2 as discussed before. After ball 2 collides with ball 3, the velocity drops to 0.0 again. In summary, there are three dots (with two 0.0 values stacked together) in the plot as the computed values of the velocities, which reflect the dynamics of the collisions. Again, having more than one value at one time point is semantically unambiguously distinct from having distinct values closely spaced in time. We need a mathematical framework to realize this.

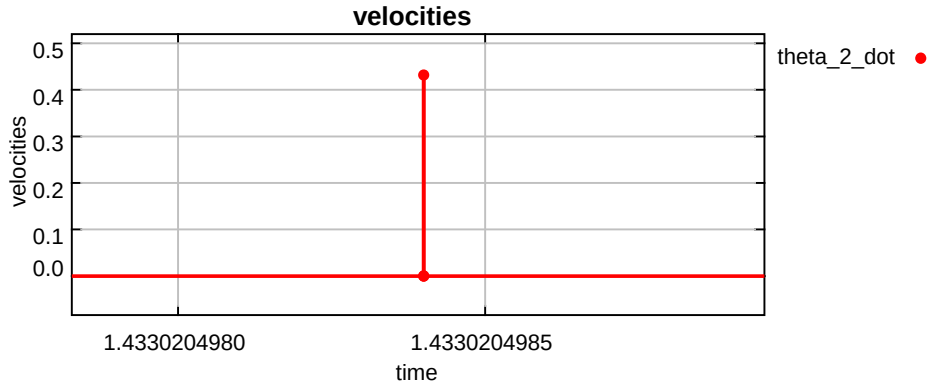

Figure 1.16. The zoomed velocity plot of ball 2 at time around 1.433.

Now let us go one step further to study the cause of having multiple values at the same time for a signal. By revisiting the Newton's cradle model, we see that the three values of velocities of ball 2 are caused by two *consecutive* collisions at the same time. This makes explicit the interactions between discrete dynamics and continuous dynamics. Now the question becomes how to model discrete dynamics.

We call the signals modeling discrete dynamics "discrete-event signals" to distinguish the continuous-time signals modeling continuous evolutions of dynamics. The key characteristics of a discrete-event signal is that it does not have values at all time points. When a discrete-event

signal has a value at a time, we call that value a *discrete event*.[6] A discrete-event signal is good at modeling sporadic mode changes such as the collisions.

In our model, to model consecutive collisions at the same time, we require a discrete-event signal to contain two discrete events at the same time to model the collision dynamics. Discrete events at the same time are called *simultaneous* events.

Simultaneous events are not uncommon in practice and usually result from the *synchrony* assumption. Here are some examples: batch arrivals at a queue; sequence of software executions which are abstracted as instantaneous; transient states (states have zero duration) in finite state machines.

For a discrete-event signal to have multiple discrete events at the same time is similar to a continuous-time signal having multiple values at the same time. We call this in general a *discontinuity* at a time.

To exaggerate the existence of simultaneous discrete events, we make one simple modification to the Newton's cradle model shown in Figure 1.10 and make the dynamics even more interesting.

In particular, we make the collisions between balls *perfectly inelastic* rather than perfectly elastic, meaning the collision restitution $k = 0$. According the equations of collisions, we get the initial velocities of balls after collision as follows:

$$v_1' = v_2' = \frac{v_1 + v_2}{2},$$

where v_1 and v_2 are the velocities before the collision, and v_1' and v_2' are the velocities after the collision. This equation essentially means that both balls move at the same speed.

The plots in Figure 1.17 show the traces of one execution with the modification to collision restitution. The upper figure shows the positions, the middle figure shows the velocities, and the lower figure shows a zoomed portion of the velocity plot at the time where collisions happen. It is easy to notice that after these collisions, all balls move together with the same speed.

Note that there are a lot of dots in the lower figure near time 1.433 indicating that it takes many computation steps to find the common velocity for all balls to move together. Each computation step corresponds to the handling of one collision. Table (1.1) shows the sequence of computation steps. The collisions with odd indexes happen between ball 1 and 2 and those with even indexes (except 0, which shows the initial velocities) happen between ball 2 and 3. Take collision `#1` as an example, following the collision equations, ball 1 and 2 get $v/2$ after collision. Then the collision `#2`

[6] We will make these concepts more precise with formal definitions in the next chapter.

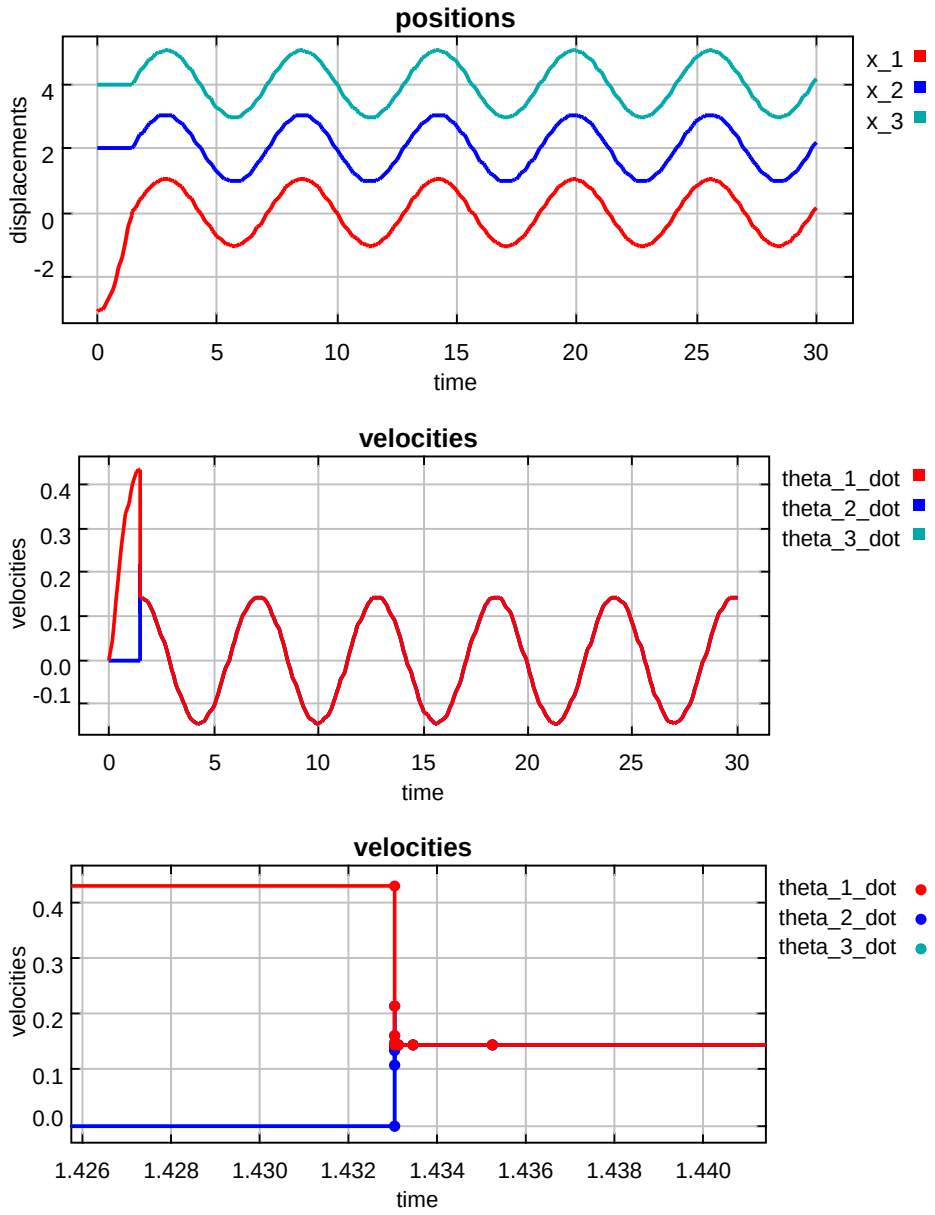

Figure 1.17. The position and velocity plots resulting from executing the hybrid systems model in Figure 1.10 with perfect inelastic collisions.

makes both ball 2 and ball 3 have a velocity of $v/4$. As more collisions happen, the kinetic energy (which keeps decreasing due to perfect inelastic collisions) and momentum get equally distributed to all three balls. Eventually, all balls gain the same velocity, $v/3$.

Table 1.1. An infinite sequence of collisions leads to a steady state.

$\#ofcollisions$	v_1	v_2	v_3
0	v	0	0
1	$v/2$	$v/2$	0
2	$v/2$	$v/4$	$v/4$
3	$3v/8$	$3v/8$	$v/4$
4	$3v/8$	$5v/16$	$5v/16$
5	$11v/32$	$11v/32$	$5v/16$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
∞	$v/3$	$v/3$	$v/3$

If the above transition dynamics is modeled as a discrete-event signal, then each collision corresponds to a discrete event, and all these events happen exactly at the same time. In this case, the discrete-event signal will have an *infinite* number of discrete events at a single time point, and this corresponds to a Zeno phenomenon. We will give further discussion on this issue later.

Now we give a mathematical definition of discrete-event signals. Again, if we define them as the following commonly used form of function,

$$x\colon T \to \mathcal{R}^n,$$

it will not be sufficient to capture the simultaneous discrete events. We need a new mathematical definition for both continuous-time and discrete-event signals. We do this in the next chapter.

Chapter 2

Signals

In this chapter, I will use the tagged signal model [61] as the basic mathematical framework to formally define signals and their discrete representations for computing purpose.

The tagged signal model provides a formal framework for considering and comparing actor-oriented models of computation. It is similar in objectives to the coalgebraic formalism of abstract behavior types in [9], interaction categories [1], and interaction semantics [88]. As with all three of these, the tagged signal model seeks to model a variety of interaction styles between concurrent components.

2.1 Signals

In the tagged signal model [61], a fundamental concept is the *tags*. The tags form a *partially ordered set* (poset) $\mathcal{T}$ with an order relation $\leq$. The tag set $\mathcal{T}$ defines the mathematical structure of signals. For example, $\mathcal{T}$ might represent causality properties, time, or activation orders.

Let $\mathcal{V}$ be the value set (the data type of the signal, such as $\mathcal{R}^n$ for signals whose values are n-tuples of reals).

Definition 1 (Event) An **event** is a pair (t, v), where $t \in \mathcal{T}$ and $v \in \mathcal{V}$. The set of events is $\mathcal{E} = \mathcal{T} \times \mathcal{V}$.

We define two operators, `tag` and `value`, for an event $e \in \mathcal{T}$, where `tag`(e) gives the tag of the

event and $\texttt{value}(e)$ gives the event value. For example, given an event $e = (t, v)$, $\texttt{tag}(e) = t$ and $\texttt{value}(e) = v$.

In the original tagged signal model, a *signal* is defined as a subset of $\mathcal{E}$. In most cases, we only study functional signals. A *functional signal* s is a partial function from $\mathcal{T}$ to $\mathcal{V}$,

$$s \colon \mathcal{T} \rightharpoonup \mathcal{V}. \tag{2.1}$$

Let $s(t) \in \mathcal{V}$ denote the value of signal s at tag t.

We will only consider functional signals in this dissertation. So from now on, without special comments, when we say "signals," we mean "functional signals."

In this dissertation, we do not directly use the original definition of signal from the tagged signal model. Instead, we deploy the definition given by Liu in [68, 69], which constraints the domain of signals in a subtle but useful way. In particular, a signal is a partial function defined on a **down set** of $\mathcal{T}$, defined next.

Definition 2 (Down Set) Let $(\mathcal{T}, \leq)$ be a poset. A subset $T \subseteq \mathcal{T}$ is a down set if

$$\forall t' \in T \text{ and } t \in \mathcal{T}, t \leq t' \Rightarrow t \in T.$$

Let $\mathcal{D}(\mathcal{T})$ be the set of all down sets of a partially-ordered set $\mathcal{T}$. The following properties come from [68].

1. $(\mathcal{D}(\mathcal{T}), \subseteq)$ is a *complete partial order* (CPO) with the least element as an empty set $\emptyset$.
2. $(\mathcal{D}(\mathcal{T}), \subseteq)$ is a *complete lattice*.

The down set T where a signal s is defined is called the *preimage* of s, written as $\text{dom}(s)$. A signal is called *complete* if $\text{dom}(s) = \mathcal{T}$.

Let $\mathcal{S}$ denote the set of all signals with tag set $\mathcal{T}$ and value set $\mathcal{V}$. $\mathcal{S}$ is a poset under the **prefix order** [68], defined next.

Definition 3 (Prefix Order) For any $s_1, s_2 \in \mathcal{S}$, s_1 is a prefix of s_2, denoted by $s_1 \sqsubseteq s_2$, if and only if $\text{dom}(s_1) \subseteq \text{dom}(s_2)$, and $\forall t \in \text{dom}(s_1), s_1(t) = s_2(t)$.

The prefix order on signals is a natural generalization of the prefix order on strings or sequences, and the extension order on partial functions [89].

As a special case, we say $s = s'$ if $\text{dom}(s) = \text{dom}(s')$ and $s(t) = s'(t), \forall t \in \text{dom}(s)$.

The set $(\mathcal{S}, \sqsubseteq)$ is also a CPO [68]. The least element of $\mathcal{S}$ is $s_\perp$ called the *empty signal*, where $\text{dom}(s_\perp) = \emptyset$.

Any pair of signals $s_1, s_2 \in \mathcal{S}$ has a greatest lower bound $\bigwedge\{s_1, s_2\} \in \mathcal{S}$. This greatest lower bound is the common prefix, which may be the empty signal if the two signals have nothing in common. In fact, any non-empty subset $\mathcal{S}' \subseteq \mathcal{S}$ has a greatest lower bound, which makes $\mathcal{S}$ a *complete lattice* in addition to a CPO.

Next we give definitions of signals for timed models of computation by associating the tag set and value set concrete structures. First, we associate the tag set with time semantics.

2.1.1 Tag Set as Global Time

In timed models of computation, the tag set represents time. In this dissertation, we only study those models of computation with a global time. Therefore, we require the tag set to be a *totally ordered set*.

As we explained earlier in Chapter 1, the set of non-negative real numbers is insufficient to model discontinuities of signals, which essentially capture the interactions of timed models of computation. In order to unambiguously describe discontinuities of signals (multiple value changes at the same time and simultaneous discrete events), we choose the following tag set,

$$\mathcal{T} = \mathcal{R}_0 \times \mathcal{N}, \tag{2.2}$$

where $\mathcal{N}$ is the set of non-negative integers. The set $\mathcal{R}_0 \subset \mathcal{R}$ still represents the time line while $\mathcal{N}$ is used for indexing signal values at the same time point [62]. This mathematical structure of the tag set is similar to the *super-dense time* in [73] and the *multitime* in [93].

An alternative way to allow a signal to have multiple values at a single time point is to use a *vector* to store all values.[1] We did not choose that option for a couple of reasons. First, using a vector will introduce a new data structure to the value set of signals. Second, in order to represent discrete-event signals, *absence* (indicating no discrete events) has to been *explicitly* represented with a special token in a vector. This is exactly what we try to avoid. The reason will be more obvious when we formally define DE signals. Third, the size of the vector may be unknown beforehand

[1] This suggestion was given by Stephen Edwards from Columbia University and Pieter Mosterman from the MathWorks separately.

because of the hard-to-predict emergent behaviors resulting from complicated interactions, which creates more complexity for implementation. Last, we want to keep a total order relation for all events in a signal, which is hard to achieve if using vectors to maintain the simultaneous events. All these issues do not appear with the super-dense time as the signal domain, especially when this domain is associated with a total order.

The tag set $\mathcal{T}$ is a totally ordered set with a *lexicographic* order relation $\leq$, where $\forall\ t_1 = (r_1, n_1)$, $t_2 = (r_2, n_2) \in \mathcal{T}$,

$$t_1 \leq t_2 \iff r_1 < r_2 \text{ or } (r_1 = r_2 \text{ and } n_1 \leq n_2).$$

A graphic representation of the tag set $\mathcal{T}$ is shown in Fig. 2.1.

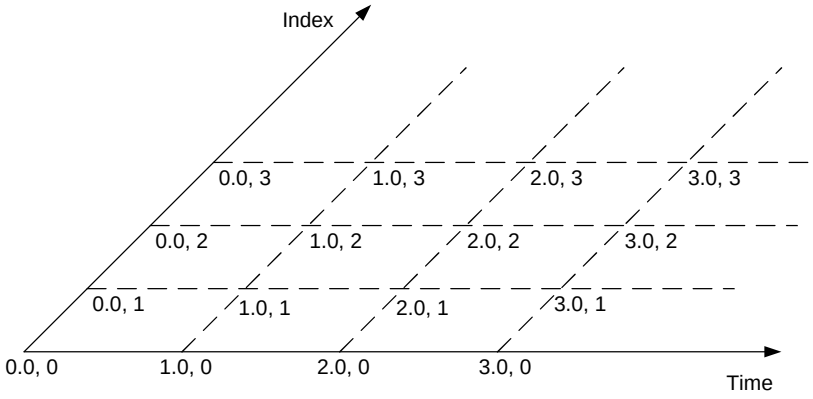

Figure 2.1. The tag set $\mathcal{T}$ is a totally ordered set.

The structure $(\mathcal{T}, \leq)$ is a CPO and the least element is $(0, 0)$, denoted as $t_\perp$. The tag set is also a complete lattice.

Operators on the Tag Set

Given a tag $t = (r, n) \in \mathcal{R}_0 \times \mathcal{N}$, $r \in \mathcal{R}_0$ is called the *time* part of t and $n \in \mathcal{N}$ is called the *index* part. We define two functions, $\texttt{time} : \mathcal{T} \to \mathcal{R}_0$ and $\texttt{index} : \mathcal{T} \to \mathcal{N}$, where

$$\forall\, t = (r, n) \in \mathcal{T}, \quad \texttt{time}(t) = r, \quad \texttt{index}(t) = n.$$

Sometimes we are interested in the functions defined on the time parts of a set of tags only. Therefore, we define an operator that takes a subset of $\mathcal{T}$ and returns the time parts of all the tags in that set.

Let $\mathcal{P}(X)$ denote the power set of set X. Define $\texttt{times} : \mathcal{P}(\mathcal{T}) \to \mathcal{P}(\mathcal{R}_0)$, where

$$\forall \mathcal{P} \in \mathcal{P}(\mathcal{T}), \texttt{times}(\mathcal{P}) = \{\texttt{time}(t) \mid t \in \mathcal{P}\}.$$

Define an operator $+ : \mathcal{T} \times \mathcal{T} \to \mathcal{T}$, with pointwise operation as follows,

$$\forall\ t_1 = (r_1, n_1),\ t_2 = (r_2, n_2) \in \mathcal{T}, \quad t_1 + t_2 = (r_1 + r_2, n_1 + n_2).$$

Sometimes, we need to directly operate on the time part or the index part of a tag, so we define two more operators $+_r : \mathcal{T} \times \mathcal{R} \to \mathcal{T}$ and $+_n : \mathcal{T} \times \mathcal{N} \to \mathcal{T}$ as follows,

$$t +_r r' = (r + r',\ n) \text{ and } t +_n n' = (r,\ n + n').$$

Because the tag set is a totally ordered set with a least element $t_\perp$, we define an operator $\texttt{min} : \mathcal{P}(\mathcal{T}) \to \mathcal{T}$. This operator takes a subset from $\mathcal{T}$ and gives the *minimum* tag in that set. Formally,

$$\forall\, \mathcal{P} \in \mathcal{P}(\mathcal{T}), \quad \texttt{min}(\mathcal{P}) = \bigwedge \mathcal{P}.$$

Table (2.1) summarizes the operators that we have defined over the tag set.

Table 2.1. Summary of the operators defined on the tag set.

Operator	Domain and Codomain
time	$\mathcal{T} \to \mathcal{R}_0$
index	$\mathcal{T} \to \mathcal{N}$
times	$\mathcal{P}(\mathcal{T}) \to \mathcal{P}(\mathcal{R}_0)$
min	$\mathcal{P}(\mathcal{T}) \to \mathcal{T}$
$+$	$\mathcal{T} \times \mathcal{T} \to \mathcal{T}$
$+_r$	$\mathcal{T} \times \mathcal{R} \to \mathcal{T}$
$+_n$	$\mathcal{T} \times \mathcal{N} \to \mathcal{T}$

Down Set of the Tag Set

Since the tag set $\mathcal{T}$ is a CPO and has a least element $t_\perp = (0, 0)$, a down set of the tag set is always left-closed with $t_\perp$ as the left endpoint. A down set of the tag set can be either right-closed or right-open. A down set is called *closed* if it is closed on both sides and denoted as $[t_\perp, t]$. Otherwise, it is called *half-closed* and denoted as $[t_\perp, t)$.

Example 1 Let $t_1 = (1, 0), t_2 = (1, 9),$ and $t_3 = (1, 10)$. Then we have the follows:

- $[t_\perp, t_1)$ is well defined, and includes all the tags whose time parts are less than 1. This set has no maximum tag.

- $[t_\perp, t_1]$ is well defined, and the maximum tag is t_1.
- $[t_\perp, t_3) = [t_\perp, t_2]$ is well defined, and the maximum tag is t_2.

Consecutive Tags

Two tags are said to be *consecutive* if there is no tag in between them. For the tag set $\mathcal{T}$, two consecutive tags have the same time part and their index parts differ by 1. For example, the tags $t_2 = (1, 9)$ and $t_3 = (1, 10)$ in Example (1) are consecutive tags. The tag with a smaller index is said to *immediately precede* the other tag.

On the other hand, for any two tags in the tag set $\mathcal{T}$ with different time parts, no matter how small the difference is, there always exists an infinite number of tags in between these two tags.

2.1.2 Value Set

A continuous-time signal has a value for each tag in its preimage, while a discrete-event signal does not have a value at most tags. We introduce a new value ε called the `absence` value, and form a new value set $\mathcal{V}_\varepsilon = \mathcal{V} \cup \{\varepsilon\}$. In such a way, both continuous-time signals and discrete-event signals can be defined as partial functions from $\mathcal{T}$ to $\mathcal{V}_\varepsilon$.

We call the elements in the original value set $\mathcal{V}$ `presence` values to distinguish them from the absence value. Given a signal s, if $s(t) = \varepsilon$, it means that the signal does not have a "presence" value at t. Without ambiguity, we simply say "the signal does not have a value."

We introduce one more value, an `unknown` value denoted by $\perp$. This value together with the $\mathcal{V}_\varepsilon$ form a new set, $\mathcal{V}_\perp = \mathcal{V}_\varepsilon \cup \{\perp\}$. In contrast, we call the values in the value set $\mathcal{V}_\varepsilon$ `known` values.

If a signal s has a value $\perp$ at tag t, it means that the signal value at t has not been defined at that tag. Note that this is different from saying "the signal does not have a value."

The value set $\mathcal{V}_\perp$ is a poset under an **information order**, $\preceq$, defined below.

Definition 4 (Information Order) For any $v_1, v_2 \in \mathcal{V}_\perp$, $v_1 \preceq v_2 \iff v_1 = \perp$ or $v_1 = v_2$.

The above definition implies that $\forall\, v \in \mathcal{V}_\perp$, $\perp \preceq v$ and $v \preceq v$, and all the other pairs of values are incomparable. The Hasse diagram of the value set $\mathcal{V}_\perp$ is shown in Figure 2.2.

It is easy to tell that the value set $\mathcal{V}_\perp$ has the following properties.

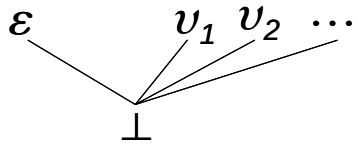

Figure 2.2. The value set $\mathcal{V}_\perp$ is a flat CPO.

1. $\mathcal{V}_\perp$ is a CPO.

2. $\mathcal{V}_\perp$ is a complete lattice.

3. $\mathcal{V}_\perp$ is a flat CPO with the longest chain of length 2.

Tuple-Value Set

The information order $\preceq$ can be easily extended to tuples of values.

Let $\mathbf{v}, \mathbf{v}' \in \mathcal{V}_\perp^n$, where $\mathbf{v} = (v_1, v_2, \cdots, v_n)$ and $\mathbf{v}' = (v_1', v_2', \cdots, v_n')$,

$$\mathbf{v} \preceq \mathbf{v}' \iff v_1 \preceq v_1', v_2 \preceq v_2', \cdots, v_n \preceq v_n'.$$

The set $(\mathcal{V}_\perp^n, \preceq)$ is a complete lattice and a CPO with the bottom element as $(\underbrace{\perp, \perp, \cdots, \perp}_{n})$. The longest chain in $\mathcal{V}_\perp^n$ has a length $n+1$.

Example 2 Let $\mathcal{V}_\perp = \{\perp, \varepsilon, 0\}$. The following figure shows the Hasse diagram of $\mathcal{V}_\perp^2$.

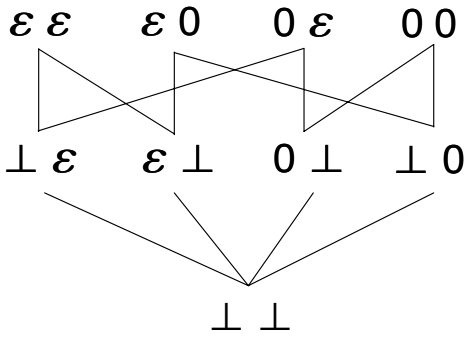

Figure 2.3. The set $\mathcal{V}_\perp^2$, where $\mathcal{V}_\perp = \{\perp, \varepsilon, 0\}$, is a CPO.

2.1.3 Timed Signals

Now we give the formal definition of signals for timed models of computation.

Definition 5 (Timed Signal) A timed signal s is a function from a down set of $\mathcal{T}$ to $\mathcal{V}_\varepsilon$, where $\mathcal{T} = \mathcal{R}_0 \times \mathcal{N}$.

From now on, we omit "timed" but simply say signals. The value $\perp$ cannot be a value for a signal. To make reading a little bit easier, we use $s(r, n)$ instead of $s((r, n))$ to denote the value of signal s at tag $t = (r, n)$.

Let the set $\mathscr{R}_s = \texttt{times}(\text{dom}(s))$ be the time parts of all tags where the signal s is defined. It is easy to tell that the set $\mathscr{R}_s$ is a down set of $\mathcal{R}_0$.

It is important to notice that the tag set $\mathcal{T} = \mathcal{R}_0 \times \mathcal{N}$ not only allows a signal to have multiple different values at a single time point, but also assigns an order to these values, indexed by the elements from the second dimension $\mathcal{N}$.

2.2 Non-Zeno Constraints

In this section, we give two constraints for signals to avoid Zeno conditions. There are two kinds of Zeno conditions, namely the *chattering Zeno* and the *genuine Zeno* conditions [6]. The first constraint, called the **finite change condition**, is to eliminate the chattering Zeno condition, and the second constraint, called **discrete discontinuous time points** is to eliminate the genuine Zeno condition.

2.2.1 Finite Change Condition

Definition (5) requires a signal to have an infinite number of values at each time point, which are indexed by the ordinary order of the index parts of their corresponding tags. Therefore, a signal may have several *different* values at a single time point. We require a signal to satisfy the finite change condition defined below.

Definition 6 (Finite Change Condition) Let s be a signal. Then s is said to satisfy the finite change condition if $\forall r \in \mathscr{R}_s$, there exists an $m \in \mathcal{N}$ (m may depend on r) such that

$$\forall\, n > m,\ s(r, n) = s(r, m). \tag{2.3}$$

Intuitively, this finite change condition requires the values of signal s to stabilize or converge before some finite index at each time point.

Example 3 The following signal s with $\text{dom}(s) = \mathcal{T}$ satisfies the finite change condition at any time point.

$$s(r, n) = \begin{cases} 1 & \text{if} \quad r = 0 \text{ and } n = 0 \\ -1 & \text{if} \quad r = 0 \text{ and } n = 1 \\ -0.5 & \text{if} \quad r = 0 \text{ and } n = 2 \\ 1 & \text{otherwise} \end{cases} .$$

Example 4 The following signal s with $\text{dom}(s) = \mathcal{T}$ does not satisfy the finite change condition at time 1.

$$s(r, n) = \begin{cases} 1 & \text{if} \quad r = 1 \text{ and } n = 2k, k \in \mathcal{N} \\ -1 & \text{if} \quad r = 1 \text{ and } n = 2k + 1, k \in \mathcal{N} \\ 1 & \text{otherwise} \end{cases} .$$

Example 5 The following signal s with $\text{dom}(s) = \mathcal{T}$ satisfies the finite change condition.

$$s(r, n) = \begin{cases} 1 & \text{if} \quad 0 \leq r < 1 \\ -1 & \text{if} \quad r \geq 1 \end{cases} .$$

This finite change condition prevents *chattering Zeno* conditions, where the values of a signal do not stabilize after an finite number of changes at a particular time. The time when a chattering Zeno condition happens is called a *Zeno time point*. Example (4) is such a signal where its values at time 1 keep alternating between 1 and -1. In Chapter 1, when the Newton's cradle model is configured with perfectly inelastic collision settings, the velocity signals of all three balls shown in Figure 1.17 are also examples of chattering Zeno signals.

If we constrain that an operational semantics evaluates signal values in the order specified by the tags and does not advance time until the signal values at a time point converge, the chattering Zeno condition would prevent that operational semantics from progressing time beyond that Zeno time point. In Chapter 6, we will give further discussions of how to simulate Zeno hybrid systems in details. At the current stage, we assume that all signals we study satisfy the finite change condition.

This finite change condition is stronger than requiring the signal to have a finite number of different values at a single time point. For example, the signal in Example (4) has at most 2 different values at each time point but it still has a chattering Zeno condition.

If a signal s satisfies the finite change condition at time r, there exists a *least* integer m satisfying Equation (2.3). We call m the **final index** of signal s at r.

If $m > 0$ at time r, then s has more than one *different* value at r. If $m = 0$ at time r, then s has the same value along all tags at time r. Without ambiguity, we simply say that "s has one value at time r."

At different time points, the final index may vary. For example, in Example (3), the final index is 3 at time 0.0 and 0 everywhere else. While in Example (4), the final index is 0 everywhere except at time 1.0, where the final index does not exist.

Definition 7 (Final Index Function) For a signal s satisfying the finite change condition, define a final index function

$$\mathcal{M}_s : \mathscr{R}_s \to \mathcal{N}, \quad \mathcal{M}_s(r) = m,$$

where m is the final index of signal s at r.

The signal value $s(r, 0)$ is the **initial value** of signal s at time r. The signal value $s(r, \mathcal{M}_s(r))$ is called the **final value** of the signal s at time r.

If $\mathscr{R}_s \neq \emptyset$, let $r_f = \bigvee \mathscr{R}_s$, due to the finite change constraint, r_f always exists and $r_f \in \mathscr{R}_s$. We call r_f the **final time** and $(r_f, \mathcal{M}_s(r_f))$ the **final tag** of the domain $\mathrm{dom}(s)$.

2.2.2 Discrete Discontinuous Time Points

Definition 8 (Discontinuity of Signals) A signal s has a discontinuity at time r if $\mathcal{M}_s(r) \neq 0$.

According to Definition (8), the signal in Example (5) does not have discontinuities. In fact, a discontinuity only happens across two consecutive tags. In particular, let r be the time when the discontinuity happens, then $\exists n \geq 0 \in \mathcal{N}$, such that $s(r, n) \neq s(r, n + 1)$. This constraint of where a discontinuity can happen makes it easy and clean to find a discrete representation of signals.

Before we give the discrete representation of signals, we will talk about the other constraint that prevents the genuine Zeno condition.

We first give the definition of **discrete set** from [58].

Definition 9 (Order Embedding) Given two posets $(A, \leq)$ and $(B, \preceq)$, an order embedding is a

function $f : A \to B$ such that for all $a, a' \in A$,

$$a \leq a' \Leftrightarrow f(a) \preceq f(a').$$

Definition 10 (Discrete Set) A discrete set is a poset for which there exists an order embedding to the natural numbers $\mathcal{N}$.

Example 6 The following examples are a few simple discrete sets.

- The empty set $\emptyset$ is a trivial discrete set.
- Any totally ordered finite set is a discrete set.
- Any singleton set is a discrete set.

This "discrete" condition is stronger that the "countable" condition. The following examples illustrate their difference.

- The sets $\{1 - \frac{1}{2^k} \mid k \in \mathcal{N}\}$ and $\{\frac{1}{2^k} \mid k \in \mathcal{N}\}$ are discrete sets and they are also countable.
- The sets $\{2\} \cup \{1 - \frac{1}{2^k} \mid k \in \mathcal{N}\}$ and $\{0\} \cup \{\frac{1}{2^k} \mid k \in \mathcal{N}\}$ are countable sets but not discrete.

Now we define **discontinuous time points** of a signal.

Definition 11 (Discontinuous Time Points) The discontinuous time points of a signal s, D_s, is a subset of $\mathscr{R}_s$, which contains all time points where the signal s has discontinuities.

$$D_s = \{r \in \mathscr{R}_s \mid \mathcal{M}_s(r) > 0\}.$$

Given a signal, the set of its discontinuous time points is unique.

Our purpose is to find a discrete representation of signals. Therefore, we require the set of discontinuous time points of a signal to be discrete. From Example (6), we know that if D_s is finite, it is discrete. On the other hand, if D_s is infinite, then we require $\bigvee D_s \notin \mathscr{R}_s$.

This condition prevents *genuine Zeno* conditions [6], where the values of a signal changes infinitely many times at distinct time points within a finite time interval. A classic example is the bouncing ball model, shown in Figure 6.1. The Zeno time point in this case is the limit of the discontinuous time points, $\bigvee D_s$.

Genuine Zeno signals also cause a challenge for designing an operational semantics to evaluate the signal values at and after the Zeno time point. In particular, if we constrain the operational

semantics to evaluate these events according to the order of their tags, then the tags whose time parts are equal to or greater than the Zeno time point cannot be reached, consequently the signal values defined on these tags cannot be evaluated. Therefore, for genuine Zeno signals, we restrict our discussion on the tags that happen before the Zeno time point. In Chapter 6, we will give a solution to get around this problem. At this time, we simply assume that all signals that we are studying satisfy both non-Zeno conditions.

All the following example signals except Example (10) have a discrete set of discontinuous time points. Example (9), Example (10), and Example (11) are all genuine Zeno signals with a Zeno time point at 1.0. Only Example (7), Example (8), and Example (11) satisfy both non-Zeno constraints mentioned above.

Example 7 (Discrete Clock) For the following signal s_1 with $\mathrm{dom}(s_1) = \mathcal{T}$, $D_{s_1} = \mathcal{N}$ is a discrete set. $\bigvee D_{s_1} = \infty$, $\mathscr{R}_{s_1} = \mathcal{R}_0$, therefore $\bigvee D_{s_1} \notin \mathscr{R}_{s_1}$.

$$s_1(r,n) = \begin{cases} 1 & \text{if} \quad r \in \mathcal{N} \text{ and } n = 1 \\ \varepsilon & \text{otherwise} \end{cases}.$$

Example 8 (Discrete Signal) For the following signal s_2 with $\mathrm{dom}(s_2) = \mathcal{T}$, $D_{s_2} = \{0, 1, 2\}$ is finite and it is a discrete set.

$$s_2(r,n) = \begin{cases} 1 & \text{if} \quad r \in \{0,1,2\} \\ \varepsilon & \text{otherwise} \end{cases}.$$

Example 9 (Zeno Signal) For the following signal s_3 with $\mathrm{dom}(s_3) = \mathcal{T}$, $D_{s_3} = \{1 - \frac{1}{2^k} \mid k \in \mathcal{N}\}$ is a discrete set. $\bigvee D_{s_3} = 1.0$, $\mathscr{R}_{s_3} = \mathcal{R}_0$, therefore $\bigvee D_{s_3} \in \mathscr{R}_{s_3}$.

$$s_3(r,n) = \begin{cases} 1 & \text{if} \quad r \in \{1 - \frac{1}{2^k} \mid k \in \mathcal{N}\} \text{ and } n = 1 \\ \varepsilon & \text{otherwise} \end{cases}.$$

Example 10 (Not Discrete Signal) For the following signal s_4 with $\mathrm{dom}(s_4) = \mathcal{T}$, $D_{s_4} = \{2\} \cup \{1 - \frac{1}{2^k} \mid k \in \mathcal{N}\}$ is not a discrete set.

$$s_4(r,n) = \begin{cases} 1 & \text{if} \quad r \in \{1 - \frac{1}{2^k} \mid k \in \mathcal{N}\} \text{ and } n = 1 \\ 1 & \text{if} \quad r = 2 \text{ and } n = 1 \\ \varepsilon & \text{otherwise} \end{cases}.$$

Example 11 (Discrete Signal) For the following signal s_5 with $\mathrm{dom}(s_5) = [(0,0),(1,0))$, $D_{s_5} = \{1 - \frac{1}{2^k} \mid k \in \mathcal{N}\}$ is a discrete set. $\bigvee D_{s_5} = 1.0$, $\mathscr{R}_{s_5} = [0.0, 1.0)$, therefore $\bigvee D_{s_5} \notin \mathscr{R}_{s_5}$.

$$s_5(r,n) = \begin{cases} 1 & \text{if} \quad r \in \{1 - \frac{1}{2^k} \mid k \in \mathcal{N}\} \text{ and } n = 1 \\ \varepsilon & \text{otherwise} \end{cases}.$$

2.3 Classifying Signals

In this subsection, we will give a few more definitions to classify signals. In particular, we will distinguish continuous-time signals and discrete-event signals.

First we need two auxiliary functions defined below.

Definition 12 (Initial Value Function) Define an initial value function $s_i : \mathscr{R}_s \to \mathcal{V}$ for signal s as $\forall\ r \in \mathscr{R}_s,\ s_i(r) = s(r,0)$.

Definition 13 (Final Value Function) Define a final value function $s_f : \mathscr{R}_s \to \mathcal{V}$ for signal s as $\forall\ r \in \mathscr{R}_s,\ s_f(r) = s(r, \mathcal{M}_s(r))$.

It is easy to see that for a signal s, if $D_s = \emptyset$, then $\forall r \in \mathscr{R}_s, s_i(r) = s_f(r)$. We say that the initial value function and final value function are equal.

2.3.1 Continuity of Initial and Final Value Functions

There are several equivalent ways to define a continuous function because of equivalent definitions for a topological structure. Let $(X;\ \mathbb{T})$ be a topological space, where X is a set and $\mathbb{T}$ is the topology on that set.

Definition 14 (Continuous Function Via Open Sets) Let $(X;\ \mathbb{T})$ and $(X';\ \mathbb{T}')$ be topological spaces. A function $f : X \to X'$ is continuous if for any $V \subseteq X'$ is open, $f^{-1}(V) \subseteq X$ that is open.

Definition 15 (Continuous Function Via Neighborhoods) Let $(X;\ \mathbb{T})$ and $(X';\ \mathbb{T}')$ be topological spaces. A function $f : X \to X'$ is continuous at $x \in X$ if for any neighborhood V of $f(x)$, there is a neighborhood $U \subseteq X$ of x such that $f(U) \subseteq V$. If f is continuous at every $x \in X$, then we say f is continuous.

Intuitively, the above two definitions of continuous functions say that a function f is continuous at x if $f(y)$ is "near" $f(x)$ whenever y is "near" x.

Definition 16 (Constant Function) Let $f : X \to Y$ be a function on the sets X and Y, f is a constant function if $\forall a, b \in X, f(a) = f(b)$.

Proposition 1 *Let $f : \mathcal{R} \to \mathcal{V}$ be a function on the topological spaces $(\mathcal{R}, \mathbb{T}_{\mathcal{R}})$ and $(\mathcal{V}, \mathbb{T}_d)$, where $\mathcal{V}$ is an arbitrary set, $\mathbb{T}_{\mathcal{R}}$ is the standard topology on $\mathcal{R}$, and $\mathbb{T}_d$ is the discrete topology. If f is a continuous function, then f is a constant function.*

Proof: Suppose f is not a constant function. Then $f(\mathcal{R})$ contains at least two elements. For $u \in f(\mathcal{R})$, let $Y = \{u\} \subseteq \mathcal{V}$. Y is an open set because of the discrete topology $\mathbb{T}_d$. Let $X = f^{-1}(Y) \subseteq \mathcal{R}$. According to Definition (14) X must be an open set. An open set in $\mathcal{R}$ is an open interval of one of the following four kinds: $\mathcal{R}$, (a, b), $(-\infty, c)$, and (d, ∞), where $a, b, c, d, \in \mathcal{R}, a < b$. However, note that X cannot be $\mathcal{R}$ because otherwise f is a constant function.

First assume $X = (a, b)$. Then, let $u' = f(a)$, then $u' \in f(\mathcal{R})$. Let $Y' = \{u'\} \subseteq f(\mathcal{R})$. Y' is an open set because of the discrete topology $\mathbb{T}_d$. Denote $X' = f^{-1}(Y') \subseteq \mathcal{R}$. Obviously, the intersection $X \cap X'$ must be an empty set. Furthermore, X' must be an interval of either $(c, a]$ or $(-\infty, a]$, which is right closed at a. This contradicts definition (14), which says that X' must be an open set. Therefore, the assumption that $X = (a, b)$ does not hold.

Similarly, we can show that the X can be neither $(-\infty, c)$ nor (d, ∞).

Therefore, the assumption that function f is not a constant function does not hold. The function f must be a constant function. □

Remark 1 In Proposition (1), the codomain $\mathcal{V}$ can be an arbitrary set. If we restrict the codomain to $\mathcal{R}$ but still equip it with the discrete topology, the only continuous functions will again be constant functions. However, if we equip the codomain with the standard topology $\mathbb{T}_{\mathcal{R}}$ instead, we can get more interesting continuous functions. In particular, by introducing the standard metric for $\mathcal{R}$, which is the absolute value function, $d(a, b) = |a - b|$, and leveraging definition (15), we get continuous functions defined with the standard $\epsilon - \delta$ definition from real analysis.

For a signal, whether its initial value function and final value function are continuous functions depends on the value set of the signal. If the value set is $\mathcal{V}_{\varepsilon}$, then the initial and final value functions are continuous functions only if they are constant functions according to Proposition (1). If the

value set is restricted to $\mathcal{R}$, then the initial and final value functions are continuous functions if they comply with the continuous function definition in real analysis.

Example 12 The function $f : \mathcal{R}_0 \to \mathcal{R}$, where $f(r) = r$ for all $r \in \mathcal{R}_0$, is a continuous function.

Example 13 The function $f : \mathcal{R}_0 \to \mathcal{R}$, where

$$f(r) = \begin{cases} 1 & \text{if} \quad 0 \leq r < 1 \\ -1 & \text{if} \quad r \geq 1 \end{cases},$$

is not a continuous function.

Definition 17 (One-Sided Continuity) Let $f : X \to Y$. The function f is called **continuous from the left side** if at any point $c \in X$, $\lim_{x \to c^-} f(x) = f(c)$. It is called **continuous from the right side** if at any point $c \in X$, $\lim_{x \to c^+} f(x) = f(c)$.

The function in Example (12) is continuous from both right and left side. The function in Example (13) is continuous from the right side but not from the left side.

2.3.2 Piecewise Continuous Signal

Definition 18 (Piecewise Continuous Signal) A signal s is a **piecewise continuous signal** if it satisfies the following conditions:

1. the initial value function s_i defined on $\mathscr{R}_s$ is continuous from the left,
2. the final value function s_f defined on $\mathscr{R}_s$ is continuous from the right.

Example 14 The following signal s with $\text{dom}(s) = \mathcal{T}$ is a piecewise continuous signal.

$$s(t) = \begin{cases} \varepsilon & \text{if} \quad t = (0, 1) \\ 1 & \text{otherwise} \end{cases}.$$

The initial value function is $s_i(r) = 1$ and the final value function is $s_f(r) = 1, \forall r \in \mathcal{R}_0$.

Example 15 The following signal s with $\text{dom}(s) = \mathcal{T}$ is a piecewise continuous signal.

$$s(t) = \begin{cases} 1 & \text{if} \quad (0, 0) \leq t \leq (1, 0) \\ \varepsilon & \text{otherwise} \end{cases}.$$

The initial value function s_i defined over $\mathcal{R}_0$ is

$$s_i(r) = \begin{cases} 1 & \text{if} \quad 0 \leq r \leq 1 \\ \varepsilon & \text{if} \quad r > 1 \end{cases}.$$

The final value function s_f defined over $\mathcal{R}_0$ is

$$s_f(r) = \begin{cases} 1 & \text{if} \quad 0 \leq r < 1 \\ \varepsilon & \text{if} \quad r \geq 1 \end{cases} .$$

The initial value function is continuous from the left and the final value function is continuous from the right.

We now revisit the signal s in Example (5). The initial value function and final value function of this signal are the same as the function f in Example (13), which is not continuous from left. Therefore, the signal s is not a piecewise continuous signal.

From now on we will only study piecewise continuous signals. In the rest of this section, we will give a few more definitions to further classify signals.

2.3.3 Continuous Signal

Definition 19 (Continuous Signal) A **continuous signal** s is a piecewise continuous signal where $\forall r \in \mathscr{R}_s$, $\mathcal{M}_s(r) = 0$.

Example 16 The following signals are simple examples of continuous signals.

1. $s_1(t) = \texttt{sin}(\texttt{time}(t)), \quad \forall t \in \mathcal{T}$,
2. $s_2(t) = \varepsilon, \quad \forall t \in \mathcal{T}$.
3. Any signal defined over the singleton set $\{t_\perp\} \subset \mathcal{T}$, where $t_\perp = (0, 0)$, is a continuous signal.

It is easy to tell that if a signal s is a continuous signal, then it satisfies the following condition,

$$s_i(r) = s_f(r), \forall r \in \mathscr{R}_s.$$

However, this is only a necessary condition, not a sufficient condition. For example, the signal s in Example (14) has $s_i(r) = s_f(r) = 1$ for all $r \in \mathscr{R}_s$, but it is not a continuous signal.

For a continuous signal, its initial value function or final value function alone carries all the information about how the signal value evolves over the time.

The following signal is the simplest continuous signal.

Definition 20 (Constant Signal) A **constant signal** s has the same value for each tag, that is, $\{s(t) \mid t \in \text{dom}(s)\}$ is a singleton set.

Next we will define continuous-time and discrete-event signals.

2.3.4 Continuous-Time Signals

Definition 21 (Continuous-Time Signal) A **continuous-time (CT) signal** s is a piecewise continuous signal whose value set is $\mathcal{R}$. Formally, $\forall t \in \mathcal{T}, s(t) \in \mathcal{R}$.

The reason to restrict the value to be $\mathcal{R}$ is to allow derivatives to be defined on CT signals.

Example 17

1. The following signal s_1 with $\text{dom}(s_1) = \mathcal{T}$ is a CT signal.

$$s_1(r, n) = \begin{cases} -1 & \text{if} \quad r = 1 \text{ and } n = 1 \\ -1 & \text{if} \quad r = 2 \text{ and } n = 1 \\ 1 & \text{otherwise} \end{cases}.$$

2. The following signal s_2 with $\text{dom}(s_2) = \mathcal{T}$ is a CT signal. We call this signal a `Continuous Clock` signal with a period 2 in contrast to the `Discrete Clock` signal in Example (7).

$$s_2(t) = \begin{cases} 0 & \text{if} \quad t = (0, 0) \\ 0 & \text{if} \quad t \in [(2k, 1), (2k + 1, 0)], k \in \mathcal{N} \\ 1 & \text{if} \quad t \in [(2k + 1, 1), (2k + 2, 0)], k \in \mathcal{N} \end{cases}.$$

3. The signal s in Example (14) is not a CT signal because it contains ε as signal value.

Definition 22 (Continuous CT Signal) A **continuous CT (CCT) signal** is both a CT signal and a continuous signal.

A CCT signal has exactly one value at any time point. The initial value and final value functions of this signal are the well-known continuous function defined in real analysis. To analyze a CCT signal, we only need to study its initial value function. An example of CCT signal is the signal s_1 in Example (16)–1.

2.3.5 Discrete-Event Signals

Definition 23 (Absence Function) Let $\mathcal{P}$ be a down set of $\mathcal{R}_0$. A function $f : \mathcal{P} \to \mathcal{V}$, where

$$f(r) = \varepsilon, \forall r \in \mathcal{P},$$

is called the **absence function**.

The absence function is both a constant function and a continuous function.

Definition 24 (Discrete-Event Signal) A **discrete-event (DE) signal** s is a piecewise continuous signal, whose initial value function and final value function are the absence function. Formally, $\forall r \in \mathscr{R}_s, s_i(r) = s_f(r) = \varepsilon$.

Example (7) and Example (11) are examples of DE signals. However, the signal s_2 in Example (8) is not a DE signal because neither its initial value function nor its final value function is an absence function.

Definition 25 (Absence Signal) An **absence signal** is both a DE signal and a continuous signal.

The signal s_2 in Example (16)–2 is the only possible absence signal.

2.4 Discrete Representation of Signals

Recall that the challenging problem is that a computer execution of a hybrid system model is constrained to provide signal values on a discrete set, in other words, to construct a discrete representation of signals. In this section, we only describe what a discrete representation looks like but leave the question of how to construct such a discrete representation for later discussion. The construction details will be extensively discussed in Section 5.2.

2.4.1 Minimum Discrete Representation

Let $\mathcal{DS}_s$ be a **discrete representation** of signal s and $\mathcal{D}_s \subset \text{dom}(s)$ be the **discrete subset of tags** where $\mathcal{DS}_s$ is defined. Formally,

Definition 26 (Discrete Representation) A **discrete representation** of signal s, denoted as $\mathcal{DS}_s$, is a subset of s defined on the discrete subset, $\mathcal{D}_s$,

$$\mathcal{DS}_s = \{(t, s(t)) \mid t \in \mathcal{D}_s\}.$$

Discontinuities indicate the interactions of discrete dynamics and continuous dynamics, which is one of the topics of this dissertation. Therefore, we require a discrete representation $\mathcal{DS}_s$ to capture all discontinuities.

A signal may have more than one discrete representations satisfying the above requirement. The smallest discrete representation satisfying this requirement is called the **minimum discrete representation** and the corresponding discrete subset of tags is called the **minimum discrete subset**. We define the minimum discrete representation of signal next.

We first select a subset of tags based on the discontinuous time points as follows,

$$\mathcal{D}_s = \{t \in \text{dom}(s) \mid \texttt{time}(t) \in D_s \text{ and } \texttt{index}(t) \leq \mathcal{M}_s(\texttt{time}(t))\}. \tag{2.4}$$

Theorem 1 *The subset of tags $\mathcal{D}_s$ constructed from Equation (2.4) is a discrete set.*

Proof: According to Definition (10), we need to find an order embedding from the set $\mathcal{D}_s$ to the natural numbers $\mathcal{N}$. We construct such a function as follows.

Since the discontinuous time points D_s is a discrete set, $\forall r \in D_s$, there exists a subset $T_r = \{r' \mid r' < r\}$, which is a finite set. The set T_r may be an empty set if r is the first element in D_s.

Recall that we require a signal to satisfy the finite change condition at each time point, including where discontinuities happen. Therefore, the final index of signal s at any time point r, $\mathcal{M}_s(r)$, is finite. Consequently, the summation

$$\sum_{r_i \in T_r} \mathcal{M}_s(r_i)$$

is also finite.

For any $t \in \mathcal{D}_s$, let $r = \texttt{time}(t)$, then $r \in D_s$. Define a function $f : \mathcal{D}_s \to \mathcal{N}$,

$$f(t) = \texttt{index}(t) + \sum_{r_i \in T_r} (1 + \mathcal{M}_s(r_i)).$$

The function f is an order embedding from $\mathcal{D}_s$ to $\mathcal{N}$ according to Definition (9). Therefore, the set $\mathcal{D}_s$ is discrete. □

Definition 27 (Minimum Discrete Subset) The discrete subset of tags for signal s given by Equation (2.4) is called the **minimum discrete subset**.

Now we get the minimum discrete representation of signal s. A discrete representation of a signal is called **minimum** if it is defined on the minimum discrete subset of that signal.

Recall that an event is a tag-value pair (t, v) where $t \in \mathcal{T}$ and $v \in \mathcal{V}$ and $\mathcal{E} = \mathcal{T} \times \mathcal{V}$ is the set of all such events. Therefore, a discrete representation is a subset of $\mathcal{E}$.

A discrete representation $\mathcal{DS}_s \subset s$ is just a set of events, but not a signal any more because $\mathcal{D}_s \subset \text{dom}(s)$ is a discrete subset but not a down set of $\mathcal{T}$.

For a signal s, since the set of discontinuous time points is unique, the minimum discrete subset of tags is unique, and the minimum discrete representation is also unique.

Example 18 (Minimum Discrete Representation of CT Signal) For the signal s in Example (3), its minimum discrete subset is

$$\mathcal{D}_s = \{(r, n) \mid r \in D_s \text{ and } 0 \leq n \leq 3\},$$

where $D_s = \{0\}$.

To be more explicit,

$$\mathcal{D}_s = \{(0, 0), (0, 1), (0, 2), (0, 3)\}.$$

The minimum discrete representation of signal s is

$$\mathcal{DS}_s = \{((0, 0), 1), ((0, 1), -1), ((0, 2), -0.5), ((0, 3), 1)\},$$

where $((0, 0), 1)$ is an event at tag $(0, 0)$ with value 1.

Example 19 (Minimum Discrete Representation of DE signal) For the signal s_5 in Example (11), its minimum discrete subset is

$$\mathcal{D}_{s_5} = \{(r, n) \mid r \in D_{s_5} \text{ and } 0 \leq n \leq 2\},$$

where $D_{s_5} = \{1 - \frac{1}{2^k} \mid k \in \mathcal{N}\}$.

The minimum discrete representation of signal s_5 is as follows, for any $t \in \mathcal{D}_{s_5}$,

$$\mathcal{DS}_{s_5}(t) = \begin{cases} 1 & \text{if} \quad \texttt{index}(t) = 1 \\ \varepsilon & \text{otherwise} \end{cases}.$$

We now introduce the concept of **discrete events** for DE signals.

Definition 28 (Discrete Event) An event in the minimum discrete representation of a DE signal is called a **discrete event** of that signal, if its value is not ε.

For example, the set of discrete events in signal s_5 is

$$\{s_5(r, n) \mid r \in D_{s_5} \text{ and } n = 1\} \subset \mathcal{DS}_{s_5},$$

where $D_{s_5} = \{1 - \frac{1}{2^k} \mid k \in \mathcal{N}\}$. It is a subset of the minimum discrete representation.

As the above example shows, the minimum discrete representation for a DE signal already captures all its dynamics. This complies with the common way that people represent DE signals such as in VHDL [10], DEVS [94], and discrete-event systems [27], except for a few subtle differences.

The first subtlety of our discrete representation of DE signal is that at any time point where a discontinuity happens, the signal allows more than discrete events to happen. This is similar to the "concurrent operations" that VHDL offers with the *delta delay* semantics but different from the others.

The second subtlety is that at any time point where a discrete event happens, at least 3 events (the tag-value pairs) are used to represent the value changes. The final index at this time point is always bigger than or equal to 2. The first event always has a tag with index as 0 and carries the absence value ε. The last event, whose tag index equals the final index, also carries absence value. Intuitively, these events together illustrate the life of a discrete event, in particular, starting with no existence of discrete events (ε), the emergence of some event (non-ε value), and then the dying of that event (ε again and forever if no more events appear in the future indexes).

For example, the minimum discrete representation of signal s_5, $\mathcal{DS}_{s_5}$, contains three events at time 0, $((0,0),\varepsilon), ((0,1),1), ((0,2),\varepsilon)$.

Therefore, when we study DE signals, analyzing their minimum discrete representations is sufficient. However, this is not true for CT signals, because the dynamics of signals evolving along the tags is completely missing in the minimum discrete representation.

Let us revisit Example (18), the minimum discrete representation of signal s only contains four events,

$$\mathcal{DS}_s = \{((0,0),1), ((0,1),-1), ((0,2),-0.5), ((0,3),1)\}.$$

This does not include what happens after time 0, which is however the majority of the dynamics of the signal (note that the signal is defined on the whole tag set). Therefore, we want to give a more practical discrete representation to CT signals to reflect the dynamics, especially the continuous dynamics. We do this next.

2.4.2 Practical Discrete Representation of CT Signals

The construction of the minimum discrete subset for a signal introduced in Equation (2.4) reflects a preference for discontinuities. Specifically, we only include the tags at the discontinuous time points in the discrete subset. This choice completely ignores what happens between discontinuities. For DE signals, it is okay since the signal values between discontinuities are the absence value and they never change. The absence value is of little significance in computation and is usually discarded anyway. However, for CT signals, what happens between discontinuities actually reflects the evolution of the signal dynamics over the time, which may have significant impact on the dynamics of other CT signals and even the appearance of discrete events in DE signals (e.g. through `LevelCrossingDetector`, discussed in the next chapter). Therefore, ignoring the continuous dynamics completely is a serious omission.

CCT Signals in CT Signal

We need to find a way to represent what happens to a CT signal in between discontinuities. First, we find the **maximum continuous subsets** of the domain of a signal where discontinuities do not occur.

Let s be a CT signal and D_s be the discontinuous time points. We assume D_s is not empty. Otherwise, the maximum continuous subset is just the whole domain. Let $r_0, r_1, r_2, \cdots \in D_s$ and $r_0 < r_1 < r_2 < \cdots$. If D_s is finite then let $r_{max} = \bigvee D_s$.

If D_s is infinite, then $\bigvee D_s \notin \mathscr{R}_s$. In other words, for any $r_i \in D_s$, there always exists $r_{i+1} > r_i$ and $r_{i+1} \in D_s$. This is the discrete continuous time points constraint for non-Zeno conditions.

Definition 29 (Maximum Continuous Subset) A **maximum continuous subset** of signal s is a subset of the domain of signal s where discontinuities do not happen, and it has one of the following forms:

1. $\{t \in \mathrm{dom}(s) \mid t \leq (r_0, 0)\}$,
2. $\{t \in \mathrm{dom}(s) \mid (r_i, \mathcal{M}_s(r_i)) \leq t \leq (r_{i+1}, 0)\}$,
3. $\{t \in \mathrm{dom}(s) \mid t \geq (r_{max}, \mathcal{M}_s(r_{max}))\}$.

The maximum continuous subsets are completely isolated by the minimum discrete subset, therefore they do not have any intersections and are independent from each other.

Take the signal s_1 from Example (17) as the study case. It has 3 maximum continuous subsets,

- $\mathcal{T}_1 = [(0,0),(1,0)]$,
- $\mathcal{T}_2 = [(1,2),(2,0)]$,
- $\mathcal{T}_3 = \mathrm{dom}(s_1) \setminus [(0,0),(2,2)]$.

Given a signal s, let $\mathcal{T}' \subset \mathrm{dom}(s)$ be a maximum subset and denote the subset of a signal s defined on $\mathcal{T}'$ as

$$s \downarrow_{\mathcal{T}'} = \{s(t) \mid t \in \mathcal{T}'\}.$$

Then $s \downarrow_{\mathcal{T}'}$ is a CCT signal defined on $\mathcal{T}_1$.

Note that $s \downarrow_{\mathcal{T}'}$ may not even be a signal with a domain $\mathcal{T}$ because $\mathcal{T}'$ may not be a down set. However, since the maximum continuous subsets are independent from each other, we can define the subset of signal on these subsets and analyze them independently. This handling allows us to decompose a relatively complicated CT signal into a few much less complicated CCT signals, which makes our analysis much easier.

By decomposing a CT signal into a set of CCT signals whose domains are the maximum continuous subsets, we can separate the semantics studies of a CT signal into two parts, the semantics of CCT signals and that of discontinuities. We have shown that discontinuities are completely captured with the minimum discrete representation in the previous subsections. In this subsection, we focus on finding discrete representations for CCT signals.

Recall a CCT signal is both a CT signal and a continuous signal. Therefore, its initial value function and final value function are equal. Either of them alone carries all the information about how the signal value evolves over the time. We choose to study the initial value function only. In this way, we can ignore the index part of the signal tags. This is valid since the signal has only one value at any time point anyway. Consequently, our problem is reduced to find a discrete representation for the initial value function instead. Also note that the initial value function of a CT signal is the well known continuous function in real analysis. Therefore, finding a discrete representation for the initial value function naturally leads to solving differential equations.

ODE Solvers

We call what happens to a CT signal between discontinuities "continuous dynamics" to distinguish it from what happens at discontinuous time points. Continuous dynamics are governed by differential equations. In this dissertation, we only consider and discuss ordinary differential equations (ODE), but the major conclusions also apply to partial differential equations (PDE).

An ODE can be represented by a set of first-order differential equations on a vector-valued state,

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

where $x : \mathcal{R} \to \mathcal{R}^n$, $y : \mathcal{R} \to \mathcal{R}^m$, and $u : \mathcal{R} \to \mathcal{R}^l$ are state, output, and input signals. The functions $g : \mathcal{R}^n \times \mathcal{R}^l \times \mathcal{R} \to \mathcal{R}^n$ and $f : \mathcal{R}^n \times \mathcal{R}^l \times \mathcal{R} \to \mathcal{R}^m$ are state functions and output functions respectively. A schematic view of the ODE problem is shown in Figure 2.4.

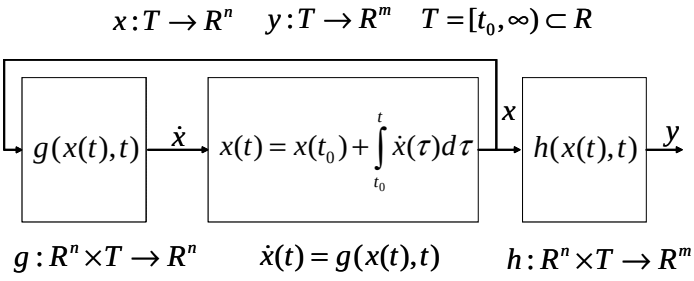

Figure 2.4. Schematic of the ODE solver problem.

Usually, ODE solvers are used to get the solutions of ODE's. We first put aside the technicality of different kinds of ODE solvers and their numerical accuracies, but instead explain the notion of "ideal ODE solver," which was introduced in [67].

An ideal ODE solver solves the ODEs exactly. In particular, given a known point $x(t)$ on the trajectory, a time instant $t' > t$, such that the ODE satisfies the Lipschitz condition during the interval $[t, t']$, and known input $u[t, t')$, an ideal ODE solver gives the exact value of $x(t')$ and $y(t')$.

This "ideal solver semantics" is not as far-fetched as it might sound. For example, many of the ODEs can be solved exactly (including those for the sticky masses example in Figure 1.4 and the Newton's cradle example in Figure 1.11) by finding a closed form expression for the solution over the intervals of continuous behavior. Even in cases where the solution must be approximated, we

would like to separate the issue of approximate ODE solutions from the other semantic issues in hybrid systems. Hence, the idealization remains useful.

Under this idealization, we only need two samples, specifically the values at t and t', along with the dynamics equations to represent a continuous function. This discrete representation loses nothing by not representing the intermediate values. Recovering the whole signal from a discrete representation is easy in that the signal value at any tag can be computed with the ideal solver semantics.

More practical solvers will provide more samples of the continuous function solutions in order to maintain adequate accuracy by controlling the time steps between samples.

One family of solvers use *Runge-Kutta* (RK) methods, which perform interpolation at each integration step to approximate the derivative at a discrete subset of time points. An explicit k stage RK method has the form

$$x(t_n) = x(t_{n-1}) + \sum_{i=0}^{k-1} c_i K_i, \tag{2.5}$$

where

$$\begin{aligned} K_0 &= h_n g(x(t_{n-1}), u(t_{n-1}), t_{n-1}), \\ K_i &= h_n g(x(t_{n-1}) + \sum_{j=0}^{i-1} A_{i,j} K_j, u(t_{n-1} + hb_i), t_{n-1} + hb_i), \quad i \in \{1, \cdots, k-1\} \end{aligned}$$

and $A_{i,j}$, b_i and c_i are algorithm parameters calculated by comparing the form of a Taylor expansion of x with (2.5).

The first order RK method, also called the *forward Euler* method, has the (much simpler) form

$$x(t_n) = x(t_{n-1}) + h_n \dot{x}(t_{n-1}).$$

This method is conceptually important but not recommended for practical usage on real applications. More practical RK methods have $k = 3$ or 4, and also control the step size for each integration step. An RK method implemented in Matlab ODE suite, called RK2-3 method, is a $k = 3$ stage method and given by

$$x(t_n) = x(t_{n-1}) + \frac{2}{9} K_0 + \frac{3}{9} K_1 + \frac{4}{9} K_2, \tag{2.6}$$

where

$$K_0 = h_n g(x(t_{n-1}), t_{n-1}), \tag{2.7}$$

$$K_1 = h_n g(x(t_{n-1}) + 0.5K_0, u(t_{n-1} + 0.5h_n), t_{n-1} + 0.5h_n), \tag{2.8}$$

$$K_2 = h_n g(x(t_{n-1}) + 0.75K_1, u(t_{n-1} + 0.75h_n), t_{n-1} + 0.75h_n). \tag{2.9}$$

In addition, after these steps are complete, the RK2-3 method estimates the local truncation error as follows,

$$\begin{aligned} K_3 &= g(x(t_n), t_n) \\ \varepsilon &= h_n \frac{-5}{72} K_0 + \frac{1}{12} K_1 + \frac{1}{9} K_2 + \frac{-1}{8} K_3. \end{aligned}$$

This estimate will be larger when the derivative of the signal varies more over the interval $[t_{n-1}, t_n]$. If the error estimate exceeds some specified threshold, then the whole process needs to be repeated with a smaller step size.

Therefore, a discrete representation of an initial value function includes some samples of the function, where the intervals between consecutive samples reflect the step sizes chosen by the practical ODE solvers. Again, we assume the samples can be computed exactly from the preceding samples by deploying the ideal ODE solver semantics.

Also notice that in order to complete one integration step, this method requires evaluation of the function g at intermediate times $t_{n-1} + 0.5h_n$ and $t_{n-1} + 0.75h_n$, in addition to the times t_{n-1}, where h_n is the step size. This requires the discrete representation of a signal to include even more samples. These intermediate samples will play a very important role when we develop an operational semantics that supports hierarchical and modular executions. We will come back to this and give detailed discussion later.

In summary, a practical discrete representation of a CT signal consists of two parts. The first part is the minimum discrete representation that captures discontinuities. The second part is the samples of numerical solutions for the initial value function of CCT Signals defined on the maximum continuous subsets.

Recovering the whole signal from a discrete representation is easy. For CT signals, the signal values at any tag can be computed with the ideal solver semantics. For DE signals, the signal has an ε value at any tag not in the discrete subset.

2.5 Tuple of Signals

Let $\mathcal{S}^n$ be the set of all n-tuple signals. The prefix order extends naturally to $\mathcal{S}^n$, and $\mathcal{S}^n$ is also a CPO and complete lattice.

Recall that a signal $s \in \mathcal{S}$ is complete if $\text{dom}(s) = \mathcal{T}$. A signal tuple is complete if all of its component signals are complete.

A n-tuple signal $\mathbf{s} = (s_1, s_2, \cdots, s_n) \in S^n$. The domain of this signal tuple $\mathbf{s}$ is

$$\text{dom}(\mathbf{s}) = \bigcap_{s_i \in \mathbf{s}} \text{dom}(s_i).$$

Let $\mathbf{s}(t) = (s_1(t), s_2(t), \cdots, s_n(t))$ for any $t \in \text{dom}(\mathbf{s})$.

The reason to choose the intersection of the domains of all component signals is to ensure the signal tuple is a total function defined over $\text{dom}(\mathbf{s})$.

The final index of a tuple of signals is defined as the supremum of the final indexes of all signals in the tuple. Formally, let $\mathscr{R}_\mathbf{s} = \texttt{times}(\text{dom}(\mathbf{s}))$,

$$\forall r \in \mathscr{R}_\mathbf{s},\ \mathcal{M}_\mathbf{s}(r) = \bigvee\{\mathcal{M}_{s_1}(r), \mathcal{M}_{s_2}(r), \cdots, \mathcal{M}_{s_n}(r)\},$$

where $\mathcal{M}_{s_i}(r)$ is the final index of component signal s_i at time r.

For a tuple of signal $\mathbf{s} = (s_1, s_2, \cdots, s_n) \in S^n$, its initial value and final value are

$$\begin{aligned}
\mathbf{s}(r, 0) &= (s_1(r, 0), s_2(r, 0), \cdots, s_n(r, 0)), \\
\mathbf{s}(r, \mathcal{M}_\mathbf{s}(r)) &= (s_1(r, \mathcal{M}_\mathbf{s}(r)), s_2(r, \mathcal{M}_\mathbf{s}(r)), \cdots, s_n(r, \mathcal{M}_\mathbf{s}(r))), \\
&= (s_1(r, \mathcal{M}_{s_1}(r)), s_2(r, \mathcal{M}_{s_2}(r)), \cdots, s_n(r, \mathcal{M}_{s_n}(r))).
\end{aligned}$$

Example 20 (Domain and Final Index Function)

- For signal tuple, $\mathbf{s} = (s, s_5)$, where s is from Example (3) and s_5 is from Example (11), its domain is

  $$\text{dom}(\mathbf{s}) = [(0, 0), (1, 0)),$$

  and its final index function is

  $$\mathcal{M}_\mathbf{s}(r) = \begin{cases} 3 & \text{if} \quad r \in \{1 - \frac{1}{2^k} \mid k \in \mathcal{N}\} \\ 0 & \text{otherwise} \end{cases}.$$

- For signal tuple, $\mathbf{s} = (s_1, s_5)$, where s_1 is from Example (7) and s_5 is from Example (11), its domain is

$$\text{dom}(\mathbf{s}) = [(0,0),(1,0)),$$

and its final index function is

$$\mathcal{M}_{\mathbf{s}}(r) = \begin{cases} 2 & \text{if} \quad r \in \{1 - \frac{1}{2^k} \mid k \in \mathcal{N}\} \\ 0 & \text{otherwise} \end{cases}.$$

The definitions of initial and final value functions easily extend to signal tuples. Let $\mathbf{s}$ be a signal tuple, then $\mathscr{R}_{\mathbf{s}} = \texttt{times}(\text{dom}(\mathbf{s}))$. The initial value function $\mathbf{s}_i : \mathscr{R}_{\mathbf{s}} \to \mathcal{V}$ for signal tuple $\mathbf{s}$ is $\forall\ r \in \mathscr{R}_{\mathbf{s}}$, $\mathbf{s}_i(r) = \mathbf{s}(r, 0)$ and the final value function $\mathbf{s}_f : \mathscr{R}_{\mathbf{s}} \to \mathcal{V}$ for signal tuple $\mathbf{s}$ is $\forall\ r \in \mathscr{R}_{\mathbf{s}}$, $\mathbf{s}_f(r) = \mathbf{s}(r, \mathcal{M}_{\mathbf{s}}(r))$.

Definition 30 (Discontinuous Time Points for Signal Tuple) Let $\mathbf{s} = (s_1, s_2, ..., s_n) \in S^n$ be a tuple of signals and

$$U = \bigcup_{s_i \in \mathbf{s}} D_{s_i}$$

be the union of the discrete time points of all component signals. The discontinuous time points of the tuple signal $\mathbf{s}$, denoted as $D_{\mathbf{s}}$, is the intersection of $\mathscr{R}_{\mathbf{s}} = \texttt{times}(\text{dom}(\mathbf{s}))$ and U.

$$\begin{aligned} D_{\mathbf{s}} &= \mathscr{R}_{\mathbf{s}} \cap U \\ &= \mathscr{R}_{\mathbf{s}} \cap \big(\bigcup_{s_i \in \mathbf{s}} D_{s_i}\big). \end{aligned}$$

Lemma 1 *Let $\boldsymbol{s} = (s_1, s_2, ..., s_n) \in S^n$ be a tuple of signals. If all s_i's satisfy the non-Zeno constraints specified in Section 2.2, then the signal tuple $\boldsymbol{s}$ also satisfies the constraints.*

Example 21 (Discontinuous Time Points)

- For signal tuple $\mathbf{s} = (s_1, s_3)$, where s_1 is from Example (7) and s_3 is from Example (9), its discontinuous time points set

$$D_{\mathbf{s}} = \{1 - \frac{1}{2^k} \mid k \in \mathcal{N}\} \cup \mathcal{N},$$

is not a discrete set.

- For signal tuple $\mathbf{s} = (s_1, s_5)$, where s_1 is from Example (7) and s_5 is from Example (11), its discontinuous time points set

$$D_{\mathbf{s}} = \{1 - \frac{1}{2^k} \mid k \in \mathcal{N}\},$$

is a discrete set.

2.5.1 Discrete Representation of Tuples of Signals

The discrete subset of a tuple of signals is the intersection of the union of the discrete subsets of the component signals and the domain of the signal tuple,

$$\mathcal{D}_{\mathbf{s}} = (\bigcup_{s_i \in \mathbf{s}} \mathcal{D}_{s_i}) \cap \ \mathrm{dom}(\mathbf{s}).$$

However, the discrete representation of a tuple of signals is not the simple union of the discrete representations of the component signals. Formally,

$$\mathcal{DS}_{\mathbf{s}} \neq \bigcup_{s_i \in \mathbf{s}} \mathcal{DS}_{s_i}.$$

This is because all component signals are required to have a value at each tag in $\mathcal{D}_{\mathbf{s}}$. Instead, following Definition (26),

$$\mathcal{DS}_{\mathbf{s}} = \{(t, \mathbf{s}(t)) \mid t \in \mathcal{D}_{\mathbf{s}}\},$$

where $\mathbf{s}(t) = (s_1(t), s_2(t), \cdots, s_n(t))$.

The following example illustrates an example of the discrete representation of a signal tuple.

Example 22 (Discrete Representation of Signal Tuple) For signal tuple, $\mathbf{s} = (s, s_5)$, where s is from Example (3) and s_5 is from Example (11), the minimum discrete subset of the signal tuple is

$$\begin{aligned}
\mathcal{D}_{\mathbf{s}} &= (\mathcal{D}_s \cup \mathcal{D}_{s_5}) \cap \mathrm{dom}(\mathbf{s}) \\
&= (\mathcal{D}_s \cup \mathcal{D}_{s_5}) \cap (\mathcal{T} \cap \mathrm{dom}(s_5)) \\
&= \{(0,0),(0,1),(0,2),(0,3)\} \cup \{(r,n) \in \mathrm{dom}(s_5) \mid r \in D_{s_5} \text{ and } 0 \leq n \leq 2\},
\end{aligned}$$

where $\mathrm{dom}(s_5) = [(0,0),(1,0))$ and $D_{s_5} = \{1 - \frac{1}{2^k} \mid k \in \mathcal{N}\}$.

The discrete representation of the signal tuple is

$$\mathcal{DS}_{\mathbf{s}} = \{(t, (s(t), s_5(t))) \mid t \in \mathcal{D}_{\mathbf{s}}\}.$$

The values $s(t)$ and $s_5(t)$ can be easily retrieved from Example (3) and Example (11).

The discrete representation $\mathcal{DS}_{\mathbf{s}}$ requires the event $((0,3), s_5(0,3))$ to be included, which is not in $\mathcal{DS}_{s_5}$. Also, the event $((\frac{1}{2},0), s(\frac{1}{2},0))$ is included in $\mathcal{DS}_{\mathbf{s}}$ but not in $\mathcal{DS}_s$.

This example shows that the discrete representation of a tuple of signals is not the simple union of the discrete representations of the component signals. Usually, the discrete representation

of a tuple of signals includes more events than the simple union of the discrete representations of component signals.

Chapter 3

Actors

Signals interact through actors. In this chapter, we study actors to understand how signals interact with each other. We define a few common and typical actors used in hybrid systems with emphasis on the interactions between CT and DE signals and between DE signals themselves. We give two useful mechanisms to model and generate discontinuities. In this section, we also study general properties of actors, such as monotonicity, continuity, and causality, to reason about some useful properties of composition of actors, including the existence, uniqueness, and liveness properties of behaviors.

3.1 Actor

In the tagged signal model [61], an *actor* receives and produces signals on *ports*. An actor a with n ports is a subset of $\mathcal{S}^n$. A particular $s \in \mathcal{S}^n$ is said to satisfy the actor if $s \in a$; $s \in a$ is called a *behavior* of the actor. Thus an actor is a set of possible behaviors. An actor therefore asserts constraints on the signals at its ports.

A `Connector` c connecting a set of ports P is a particulary simple actor where signals at each port $p \in P$ are constrained to be identical.

A set A of actors and a set C of connectors defines a *composite actor*. The composite actor is defined to be the intersection of all possible behaviors of the actors A and connectors C.

In many actor-oriented formalisms, ports are either inputs or outputs to an actor but not

both. Consider an actor $a \subseteq \mathcal{S}^n$ where $I \subseteq \{1, ..., n\}$ denotes the indices of the input ports, and $O \subseteq \{1, ..., n\}$ denotes the indices of the output ports. We assume that $I \cup O = \{1, ..., n\}$ and $I \cap O = \emptyset$. Given a signal tuple $s \in a$, we define $\pi_I(s)$ to be the projection of s on a's input ports, and $\pi_O(s)$ on output ports.

An actor is said to be *functional* if

$$\forall\ s, s' \in a, \quad \pi_I(s) = \pi_I(s') \Rightarrow \pi_O(s) = \pi_O(s').$$

Such an actor can be viewed as a function from input signals to output signals. Specifically, given a functional actor a with I input ports and O output ports, we can define an *actor function* with the form

$$F_a \colon \mathcal{S}^n \to \mathcal{S}^m, \tag{3.1}$$

where $n = |I|$ and $m = |O|$ denote the size of the input and output port sets. When it creates no confusion, we make no distinction between the actor a (a set of behaviors) and the actor function F_a. In this dissertation, we only discuss functional actors. So when we say "actors", we mean "functional actors."

A *source* actor is an actor with no input ports (but only output ports). It is functional if and only if its behavior set is a singleton set. That is, it has only one behavior. A *sink* actor is an actor with no output ports, and it is always functional.

3.2 Example Actors

In this section, we will introduce some typical and commonly used actors in hybrid systems. We emphasize on their operations on the CT and DE signals aforementioned in Chapter 2.

Here are some notations used throughout this section. Let s and s' be input and output signals respectively. If more than one input or output signals are used, an integer subscript is used to index them, such as s_1 and s'_2.

3.2.1 Strict Functional Actors

A **strictly functional** actor, denoted as $f : \mathcal{S}^n \to \mathcal{S}^m$, takes an input signal tuple $\mathbf{s} \in \mathcal{S}^n$ and generates an output signal tuple $\mathbf{s}' \in \mathcal{S}^m$ in the following way.

$$\begin{aligned} \mathrm{dom}(\mathbf{s}') &= \mathrm{dom}(\mathbf{s}), \\ (s'_1(t), s'_2(t), \cdots, s'_m(t)) &= f(s_1(t), s_2(t), s_3(t), \cdots, s_n(t)). \end{aligned}$$

Typical examples include the `Adder` actor which sums up all input signals and the `Expression` actor which implements a general function.

3.2.2 Source Actors

A source actor takes no input signals and produces only one signal. A `CTSource` actor produces a CT signal. For example, a `ContinuousClock` actor generates a square wave signal (also called the `Continuous Clock` signal) as in Example (17)–2. A `CTSource` actor can also generate CCT signals. For example, a `ContinuousSinwave` actor generates a sinusoidal wave signal in Example (16)–1; a `Const` actor generates a constant CT signal.

A `DESource` actor produces a DE signal with a sequence of discrete events at intervals that are spaced randomly or periodically. For example, a `DiscreteClock` actor generates the `Discrete Clock` signal in Example (7) with a fixed interval of 1 time unit. The simplest `DESource` actor is the `SingleEvent` actor, which produces a single discrete event.

3.2.3 Integrator

An `Integrator` actor takes a CT signal and generates a CCT signal. The initial value function of the input CT signal specifies the derivatives of the initial value function of the output CCT signal. Therefore, the input CT signal s is also called "derivative signal" and the output signal s' is called "state signal."

In the theory of Lebesgue integration (c.f. [82]) defined on reals, we have that for any $r \in \mathcal{R}$, the derivative $\dot{x}(r)$ has measure 0, thus, the integration result $x(t)$ depends only on $\dot{x}[0, t)$ but not on $\dot{x}(t)$. A small extension of this conclusion to the tag set gives us the relationship between the

domains of the input signal and output signal. In particular,

$$\mathrm{dom}(s') = \begin{cases} \mathrm{dom}(s) & \text{if } \mathrm{dom}(s) \neq \emptyset \\ t_\perp & \text{if } \mathrm{dom}(s) = \emptyset \end{cases}.$$

Assume $t_\perp = (0, 0)$. Given $s'_i(0) = s'(0, 0)$ as the initial value of the output signal s', the rest of the output signal values are defined as follows.

$$\begin{aligned} s'_i(r) &= s'_i(0) + \int_0^r s_i(\tau, 0) d\tau, \\ s'(t) &= s'_i(\texttt{time}(t)), \quad \forall t \in \mathrm{dom}(s'), \end{aligned}$$

where s_i and s'_i are the initial value functions of the input and output signals respectively.

The above equations ignore the discontinuities of the input CT signal, in particular, the signal values with indexes bigger than 0. The discontinuities have no effect on the output CCT signal because they have zero duration.

3.2.4 Merge

A `Merge` actor takes a tuple of n input signals and generates one output signal. The subtlety of defining a merge process is about how to define the output signal value at a tag if more than one input signal has a present value at the same tag. We will give two different definitions, one is called "simple merge" and the other one is called "lossless merge."

We will only give the definition of the `Merge` actor that takes 2 input signals and generates 1 output signal. For a general `Merge` actor that takes n input signals, $\texttt{Merge} : \mathcal{S}^n \rightarrow \mathcal{S}$, we define it as a composition of the `Merge` actor that takes 2 input signals. Specifically,

$$\begin{aligned} \texttt{Merge}(s_1, s_2, \cdots, s_n) &= \texttt{Merge}(\texttt{Merge}(s_1, s_2), \cdots, s_n) \\ &= \texttt{Merge}(\texttt{Merge}(\texttt{Merge}(s_1, s_2), s_3), \cdots, s_n) \\ &= \texttt{Merge}(\texttt{Merge}(\texttt{Merge}(s_1, s_2), \cdots, s_{n-1}), s_n). \end{aligned}$$

All definitions of the `Merge` actor given in this subsection are *biased* in that the actor prioritizes the input signals based on their appearance order in the signal tuple.

Let s_1 and s_2 be two input signals and s' be the output signal. The `Merge` actor does not alter the signal types and both input signals are required to be of the same type. If the input signals are CT (DE) signals, the output signal is also a CT (DE) signal.

There are two ways to define the `Merge` actor. A `SimpleMerge` is defined as follows.

$$\begin{aligned} \mathrm{dom}(s') &= \mathrm{dom}(s_1) \cap \mathrm{dom}(s_2), \\ s'(t) &= \begin{cases} s_1(t) & \text{if } s_1(t) \neq \varepsilon \\ s_2(t) & \text{if } s_1(t) = \varepsilon \end{cases}. \end{aligned}$$

This definition says that if neither of the input signals has a present value, the output signal value is ε. Also, if both input signals have present values at a tag, then output signal only carries the present signal value of the signal s_1 but discards the presence value of s_2.

Apparently, if both input signals are CT signals, then the output signal is always the input signal s_1. In this case, the `Merge` actor functions like a `Connector` connecting the first input signal to the output signal.

The behavior of the `Merge` actor is more interesting if both input signals are DE signals. Let $\mathscr{R}_{s'} = \mathscr{R}_{s_1} \cap \mathscr{R}_{s_2}$. At any $r \in \mathscr{R}_{s'}$, according to the above definition of `SimpleMerge`, there exists a *finite* final index for the output signal s',

$$\begin{aligned} \mathcal{M}_{s'}(r) &= \mathtt{max}(\mathcal{M}_{s_1}(r), \mathcal{M}_{s_2}(r)), \\ s'(r, n) &= s'(r, \mathcal{M}_{s'}(r)), \text{ for all } n > \mathcal{M}_{s'}(r). \end{aligned}$$

Therefore, the merging process always terminates (meaning that there is no more discrete event in any input signals to be handled) in a finite number of steps and the merged output signal satisfies the finite change condition.

Sometimes, we want the output signal to carry all the information of the input signals. This requires a `LosslessMerge` actor as explained next. This `LosslessMerge` actor is also a biased one where the appearance order of component signals in a signal tuple determines the appearance order of their events in the output signal.

A complete mathematical definition of the `LosslessMerge` actor will be cumbersome. So, here we give a procedure describing the merge process instead. At any $r \in \mathscr{R}_{s'}$, let n be the index of the input signals s_1 and s_2 and n' the index of the output signal s', where both indexes start with 0. The output signal s' is constructed in the following procedure.

Procedure 1 *Lossless Merge Procedure:*

1. *If no input signal has a presence value, the output signal value* $s'(r, n') = \varepsilon$.

2. *If only one signal (e.g. s_1) has a presence value, then the output signal value $s'(r, n') = s_1(r, n)$.*

3. *If both signals have presence values, then the output signal is first given the value from s_1, $s'(r, n') = s_1(r, n)$. Then the index n' is increased by 1 and the output signal value is given from s_2, $s'(r, n') = s_2(r, n)$.*

4. *Increase n and n' by 1 and repeat the process starting from step 1 until the final index of the output signal $\mathcal{M}_{s'}(r)$ is reached.*

Figure 3.1 shows an example of how the `Merge` actor merges two signals s_1 and s_2 into one signal. The signal s' is the result of simple merge that discards the second input if both inputs are present, and signal s'' is the result of lossless merge.

n	0	1	2	3	4	5	6	7
$s_1(0,n)$	ε	v_1	ε	v_2	v_3	ε	$\cdots$	$\cdots$
$s_2(0,n)$	ε	u_1	u_2	ε	u_3	ε	$\cdots$	$\cdots$
$s'(0,n)$	ε	v_1	u_2	v_2	v_3	ε	$\cdots$	$\cdots$
$s''(0,n)$	ε	v_1	u_1	u_2	v_2	v_3	u_3	ε

Figure 3.1. An example shows how the `Merge` actor merges two signals into one signal.

Note that at time 0, the output signal s'' from the lossless merge process has a bigger final index than any final index of the input signal. Sometimes, the final index of the merged signal may not exist at all. For example, the lossless merge result of the following two CT signals,

$$\begin{aligned} s_1(t) &= 1, \\ s_2(t) &= -1, \end{aligned}$$

is a chattering Zeno signal and does not have a final index at any time.

However, if both input signals are DE signals, where the final values are always ε, then at any $r \in \mathscr{R}_{s'}$, there exists a finite final index for the output signal s', in particular, the final index has an upper bound, $\mathcal{M}_{s'}(r) \leq \mathcal{M}_{s_1}(r) + \mathcal{M}_{s_2}(r)$. Therefore, the lossless merge process following Procedure (1) always terminates in a finite number steps and the output signal satisfies the finite change condition.

Therefore, we reserve the `LosslessMerge` actor exclusively for DE input signals, while the `SimpleMerge` actor can be applied on any pair of piecewise continuous signals.

3.2.5 Time Delay

A `TimedDelay` actor delays the input signal for a fixed or variable amount of time. The amount of time is required to be non-negative to ensure the causal relation between the input and output signals. If the amount of delay is bigger than 0, then the signal is delayed in the time dimension. If the amount is 0, then the signal is delayed in the index dimension.

We first talk about the `TimedDelay` actor with a fixed time delay, denoted as δ, which is usually specified as a parameter of the actor. Recall the set $\mathscr{R}_s$ includes the time parts of the tags where the input signal s is defined. The output signal s' is defined below.

The domain of the output signal is

$$\mathrm{dom}(s') = \mathrm{dom}(s) \cup \{t +_r \delta \mid t \in \mathrm{dom}(s)\}.$$

If $\delta > 0$, then

$$s'(r,n) = \begin{cases} s(r-\delta, n) & \text{if} \quad r - \delta \in \mathscr{R}_s \\ \texttt{DefaultValue} & \text{else} \end{cases}.$$

If $\delta = 0$, then

$$s'(r,n) = \begin{cases} s(r, n-1) & \text{if} \quad n \geq 1 \\ \texttt{DefaultValue} & \text{else} \end{cases}.$$

The "DefaultValue" is another parameter of the `TimedDelay` actor, which is used as the signal value when no value from the original signal can be used. The default value varies for different types of input signals. In most cases, the default value is ε for DE input signals and some real number (e.g. 0) for CT input signals.

Example 23 Let the input signal to a `TimedDelay` actor be following CT signal, where $\mathrm{dom}(s) = \mathcal{T}$ and

$$s(r,n) = \begin{cases} 1 & \text{if} \quad r = 1 \text{ and } n = 2 \\ 2 & \text{if} \quad r = 2 \text{ and } n = 1 \\ 0 & \text{otherwise} \end{cases}.$$

This signal is shown in Figure 3.2, where the red (grey if in black and white printouts) arrows indicate discontinuities. The output signals with amount of delay $\delta = 1$ and $\delta = 0$ are shown in Figure 3.3 and Figure 3.4 respectively.

It is important to notice that even when the amount of delay $\delta = 0$, the `TimedDelay` actor still imposes a *minimum delay*, index increment by 1, between the output and input signals as shown

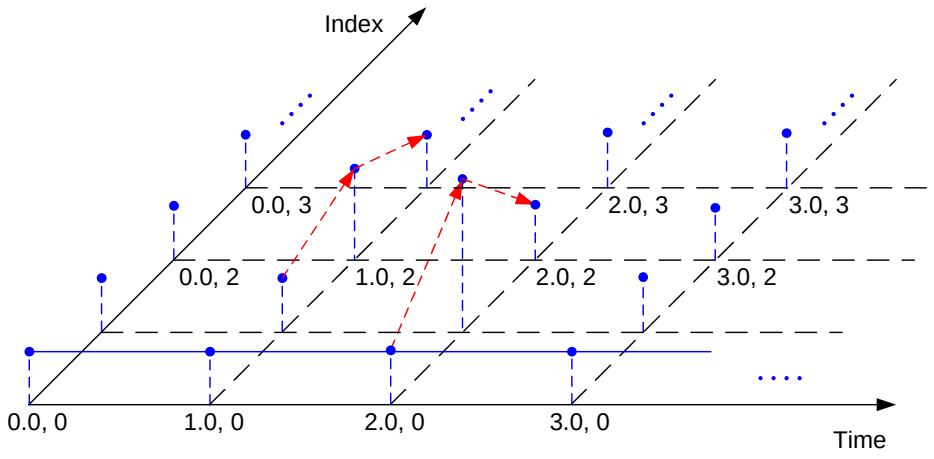

Figure 3.2. A CT input signal to the `TimedDelay` actor.

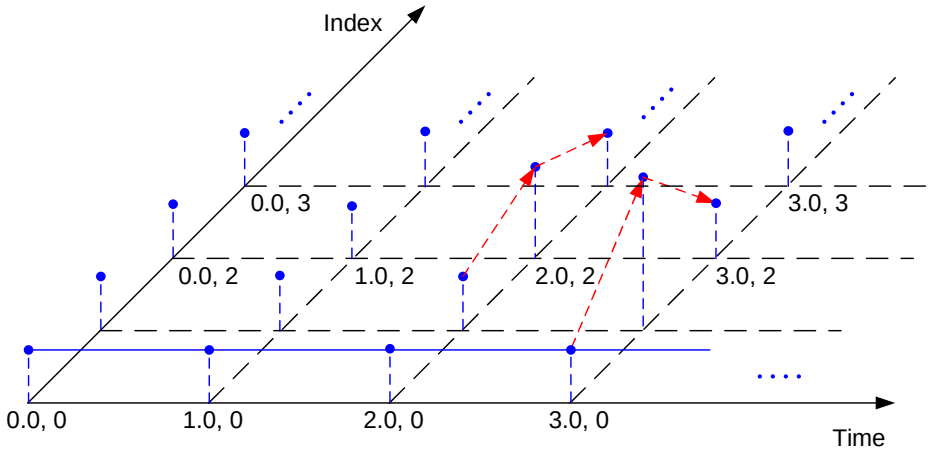

Figure 3.3. The output signal from the `TimedDelay` actor with delay $\delta = 1$ and input signal as shown in Figure 3.2.

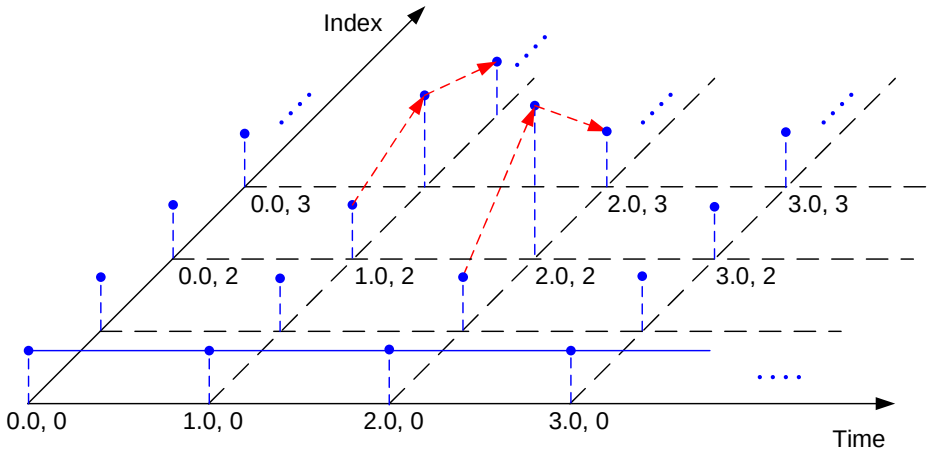

Figure 3.4. The output signal from the `TimedDelay` actor with delay $\delta = 0$ and input signal as shown in Figure 3.2.

in Figure 3.4. This is semantically different from the `Connector` actor introduced in the previous section, where the `Connector` actor requires the input and output signals to be identical and does not introduce any delay. This "minimum delay" property of the `TimedDelay` actor is particularly useful to help break feedback loops and find execution schedules based on topological sorting.

Now we discuss a more complicated `TimedDelay` actor, where the amount of delay is specified by the values of a second input signal, s_2. In other words, the amount of delay now becomes a function of the tags, $\delta(t) = s_2(t) \in \mathcal{R}_0$, which may vary over the tag. We call such an actor `VariableTimedDelay` actor. The domain of the output signal now becomes

$$\mathrm{dom}(s') = \mathrm{dom}(s) \cup \{t +_r s_2(t) \mid t \in \mathrm{dom}(s) \cap \mathrm{dom}(s_2)\}.$$

A first thought of the semantics of the `VariableTimedDelay` actor is simple and intuitive: at each tag $t \in \mathrm{dom}(s) \cap \mathrm{dom}(s_2)$, this `VariableTimedDelay` actor delays each input event $s(t)$ to a new tag t', where $t' = t +_r s_2(t)$. However, there is a subtlety about the above semantics. The complication comes from the fact that multiple input events may be delayed to the same tag. We illustrate this by extending Example (23) with a second input signal s_2 as shown in Figure 3.5. This signal is a CT signal with domain $\mathrm{dom}(s_2) = \mathcal{T}$ and

$$s_2(r, n) = \begin{cases} 0 & \text{if} \quad r = 2 \text{ and } n = 1 \\ 1 & \text{otherwise} \end{cases}.$$

The amount of delay, which is the value of signal s_2, is 1 everywhere except at tag $(2.0, 1)$, where the delay is 0. Therefore, both the input events $s(2.0, 1)$ and $s(1.0, 2)$ are delayed to the tag $(2.0, 2)$ according to the above intuitive semantics, which complies with the specifications of the `TimedDelay` actor. Since only one event is allowed at a tag for the output signal, one of these two events has to be delayed *further*. We choose the event with a larger index in the original input signal s, because we constrain our operational semantics to handle events according to their tag order. The constraint is that once a final decision is made at tag $(1.0, 2)$ to delay the event $s(1.0, 2)$ to tag $(2.0, 2)$, the decision cannot be recalled at a later tag (e.g., $(2.0, 1)$). The amount of the *further delay* is *one increment of the index*. That is, the event $s(2.0, 1)$ is delayed to tag $(2.0, 3)$ instead of $(2.0, 2)$. In turn, the events at time 2.0 with indexes bigger than 2 will be delayed by 1 in index. The resulting output signal is show in Figure 3.6. It is straightforward to apply this semantics to the cases where more than two events are delayed to the same tag.

The `TimedDelay` actor and the `VariableTimedDelay` actor do not change signal types. That is, if the input signal s is a CT (DE) signal, then the output signal s' is also a CT (DE) signal.

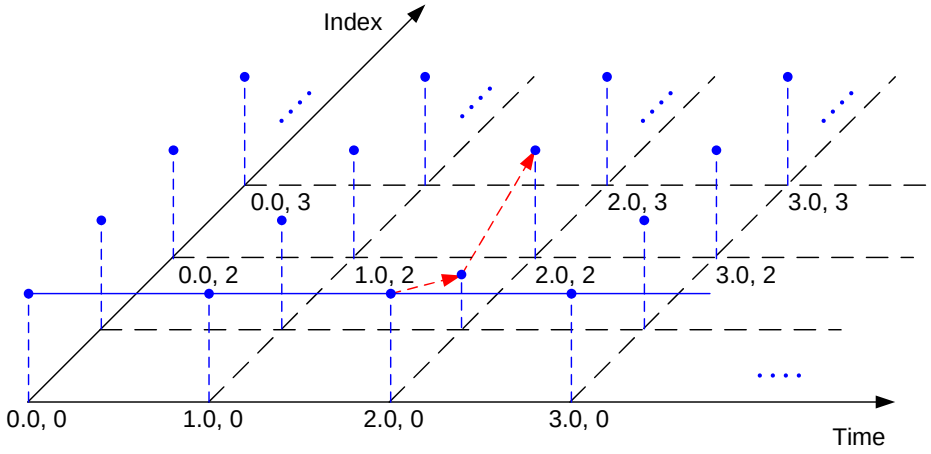

Figure 3.5. The second input signal to the `TimedDelay` actor, which specifies the amount of delay to be applied to the input signal in Figure 3.2 at each tag.

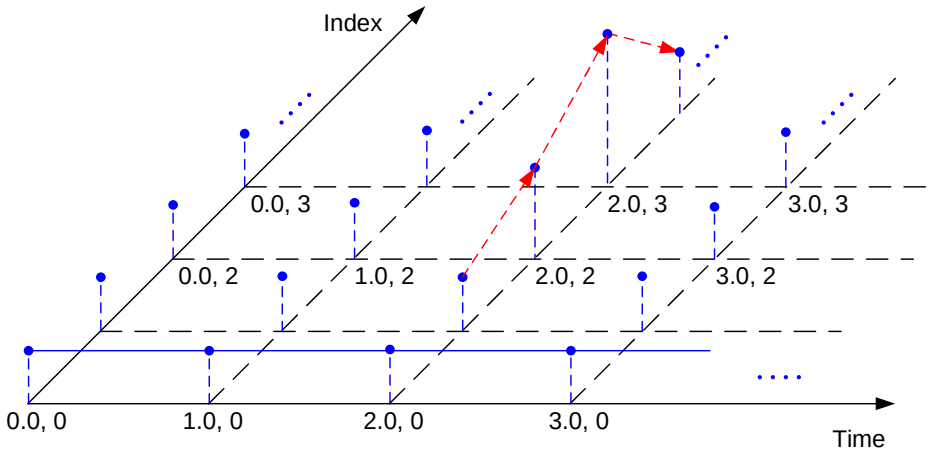

Figure 3.6. The output signal from the `VariableTimedDelay` actor with delay specified by the input signal shown in Figure 3.5.

Next we provide a small library of actors to create DE signals from CT signals, and vice versa.

3.2.6 Zero Order Hold

The `ZeroOrderHold` actor takes a DE input signal s and produce a CT output signal s', where

$$\text{dom}(s') = \text{dom}(s).$$

The output signal value in the interval $t \in [t_i, t_{i+1})$ is equal to the value of the input signal at t_i, where $t_i, t_{i+1} \in \mathcal{D}_s$ are the tags of two discrete events in the DE input signal. The tag t_i immediately precedes t_{i+1}, meaning that there does not exist a tag $t' \in \mathcal{D}_{s_i}$ such that $t_i < t' < t_{i+1}$ and $s(t') \neq \varepsilon$. The output signal takes the `DefaultValue`, specified by a parameter, as its value before any discrete event appears in the DE input signal.

Intuitively, the CT output signal value cannot be changed until the DE input signal has a discrete event. What is more, if the new discrete event carries the same value as the previous discrete event, the signal value in fact does not change.

3.2.7 Samplers

The `PeriodicSampler` actor takes a CT input signal s and produce a DE output signal s'. This actor has a *sampling period* parameter p. The output signal value is absence at all tags except at a subset of tags $\{(k \cdot p, 1) \mid k \in \mathcal{N}\}$. At these tags, the output signal value is equal to the input signal value. Formally,

$$\begin{aligned} \text{dom}(s') &= \text{dom}(s), \\ s'(r, n) &= \begin{cases} s(r, n) & \text{if} \quad n = 1 \text{ and } r = k \cdot p,\ k \in \mathcal{N} \\ \varepsilon & \text{else} \end{cases} . \end{aligned}$$

We give this actor an unambiguous semantics even for samples at discontinuities. Specifically, the actor always picks up the input signal value at tags with index 1, because discrete events in DE signals always have tags with indexes greater than or equal to 1.

The `TriggeredSampler` actor uses a second DE input signal s_2 to specify when to take samples of a CT input signal s. Whereas the `PeriodicSampler` uses the value of the input signal at index 1, the `TriggeredSampler` uses whatever value of the input CT signal has the same tag as the trigger

discrete event in the DE input signal. The generated output signal s' is a DE signal. Formally,

$$\begin{aligned} \mathrm{dom}(s') &= \mathrm{dom}(s) \cap \mathrm{dom}(s_2), \\ s'(t) &= \begin{cases} s(t) & \text{if} \quad s_2(t) \neq \varepsilon \\ \varepsilon & \text{else} \end{cases} . \end{aligned}$$

Example 24 Let the second DE input signal s_2 be the `discrete clock` signal in Example (7), then the `TriggeredSampler` actor functions exactly the same as a `PeriodicSampler` with a sampling period $p = 1$.

3.2.8 Level Crossing Detector

The `LevelCrossingDetector` actor takes a CT input signal s and produces a DE output signal s', where

$$\mathrm{dom}(s') = \mathrm{dom}(s).$$

If the input signal value crosses a specified threshold at tag (r, n), then a discrete event appears in the output signal at tag $(r, n + 1)$.

A common usage of the `LevelCrossingDetector` actor is to feed a CCT signal as the input signal, then the level crossing only happens at tags with indexes equal to 0. Therefore, the discrete events only appear at tags with indexes equal to 1.

If the input signal has discontinuities, such as the signal shown in Figure 3.2, and the `LevelCrossingDetector` actor detects level crossings of threshold 0.5, then there will be 4 level crossings at $(1.0, 2)$, $(1.0, 3)$, $(2.0, 1)$, and $(2.0, 2)$ respectively. The generated output signal will have 4 discrete events at tags $(1.0, 3)$, $(1.0, 4)$, $(2.0, 2)$, and $(2.0, 3)$.

In summary, the `LevelCrossingDetector` actor detects level crossings of both continuous dynamics and discontinuities.

Remark 2 The semantics of the `LevelCrossingDetector` actor mentioned above has great impact on the definition of DE signals. In particular, we require a discrete event to appear in the output DE signal at the exactly the same tag where the new signal value takes place, in order to reflect the *causal but not strictly causal* relation between discrete events and discontinuities. The design of the `PeriodicSampler`, where samples take the input signal values at tags with index 1, also reflects the concern for this relation. We will see more explanations next.

3.3 Discontinuities

Discontinuities in CT signals are the results of interactions between CT signals and DE signals. In this section, I will further investigate the causes of discontinuities and provide two common mechanisms to generate discontinuities. One is through *discrete events*, the other one is through *transient states*.

A discontinuity in a CT signal is always caused by discrete events. When a discontinuity of a CT signal happens, the signal value changes across two consecutive tags. Specifically, let t, t' be the two tags corresponding to the values before and after change respectively, then $t < t'$ and $t' = t +_n 1$. This value change is caused by a discrete event at tag t'. This relation is illustrated in the example in Figure 3.7 and Figure 3.8 .

The discrete events of the DE signal shown in Figure 3.7 at tags $(2.0, 1)$, $(2.0, 2)$, and $(3.0, 2)$ cause the value of the CT signal shown in Figure 3.8 to change, indicated by the red arrows.

We have introduced a few primitive actors that create discontinuities in CT signals, specifically, the `CTSource` actor, `VarialbleTimedDelay` actor, and `ZeroOrderHold` actor. In the rest of this section, we will study more complicated mechanisms to generate discontinuities. These mechanisms provide us more power to model more interesting and complicated hybrid systems.

3.3.1 Discrete Events

In this subsection, we introduce the first mechanism for creating more complicated discontinuities in CT signals via *simultaneous* discrete events. We explain this mechanism by giving an example. The example extends the `Integrator` actor, which produces CCT signal only, to accept DE input signals and generate a CT output signal with discontinuities. We emphasize the effect of the discrete events of the DE input signals on the CT output signal.

Figure 3.9 shows a model that utilizes the extended `Integrator` actor to create a CT signal with discontinuities. This model includes a `Const` actor, an extended `Integrator`, a `TimedPlotter`, and two `DEScource` actors.

The `Const` actor produces a constant CT signal $s_1(t) = 1, \forall t \in \mathcal{T}$. The two `DEScource` actors generate two DE signals, one is called "impulse" signal, which adjusts the integration result by adding its signal values to the output signal values. The other one is called "reset" signal, which resets the

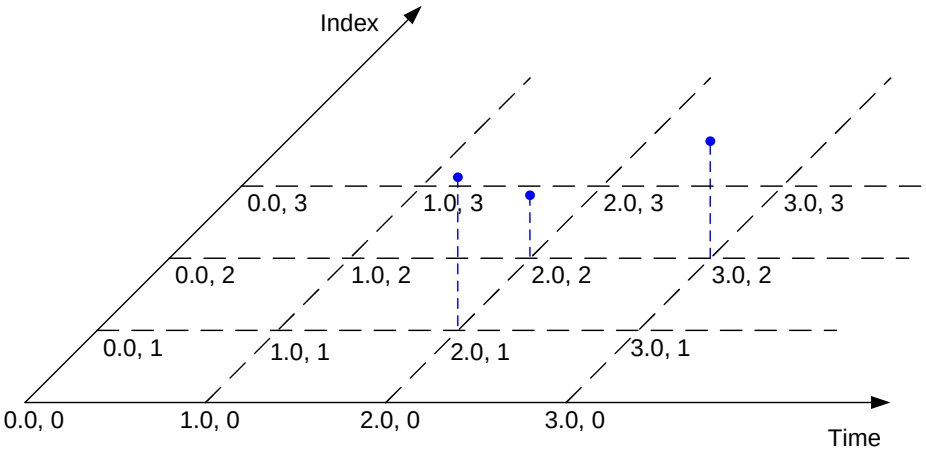

Figure 3.7. A DE signal with discrete events at time 2.0 and 3.0. This signal causes the discontinuities of the CT signal Figure 3.8.

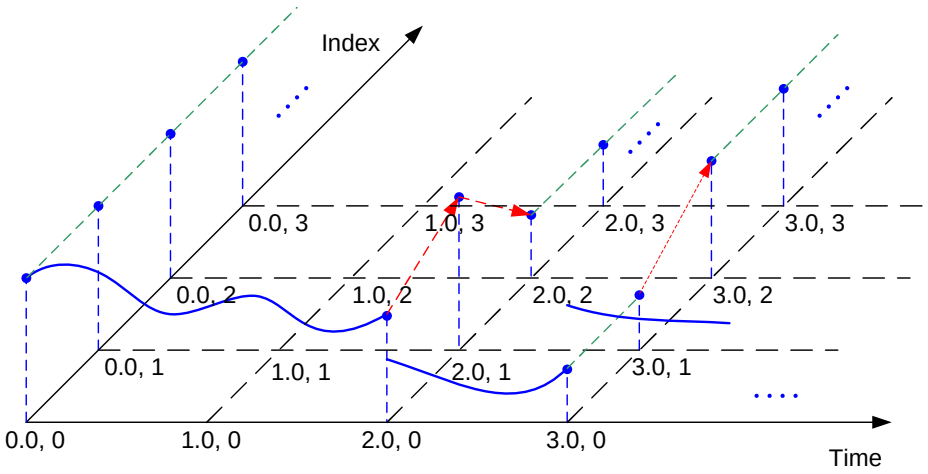

Figure 3.8. A CT signal with discontinuities at time 2.0 and 3.0.

integrator result with its signal values[1]. The impulse signal models the how Dirac delta functions affect integration results. Let $s_2(t)$ and $s_3(t)$ be the impulse signal and reset signal respectively.

The `Integrator` takes all these three signals as inputs. We have

$$\mathrm{dom}(s') = \mathrm{dom}(s_1) \cap \mathrm{dom}(s_2) \cap \mathrm{dom}(s_3).$$

Let T_2 and T_3 be the subsets of tags where discrete events in the impulse signal s_2 and the reset signal s_3 happen respectively. Note that an event with the absence value is not a discrete event.

At any $t \in \mathrm{dom}(s')$, let t' be the largest tag in T_3 that is smaller than t. Formally,

$$T' = T_3 \cap [t_\perp, t], \quad t' = \begin{cases} \bigvee T' & \text{if } T' \neq \emptyset \\ t_\perp & \text{else.} \end{cases}.$$

Let $v(t)$ be the initial value of the output signal s' at tag t. Then

$$v(t) = \begin{cases} s_3(t') & \text{if } t' \in T' \\ s'(t_\perp) & \text{else.} \end{cases}.$$

Then the output signal s' is defined as follows.

$$s'(t) = v(t) + \int_{t'}^{t} s(\tau) d\tau + \sum_{t_i \in (\,(t', t] \cap T_2\,)} s_2(t_i). \tag{3.2}$$

Note that $\int_{t'}^{t} s(\tau) d\tau$ and $\tau \in \mathrm{dom}(s')$ have exactly the same semantics as defined in Section 3.2.3 for the original `Integrator` actor, which is first defined on the initial value functions s_i of signal s and then extended to the whole tag set.

Figure 3.10 shows the integration result of a simple setting, where the impulse signal contains only one discrete event at tag $(1.0, 1)$ and $s_2(1.0, 1) = 2.0$, and the reset signal also contains only one discrete event at tag $(2.0, 2)$ and $s_2(2.0, 2) = -2.0$. The dots in the figure are where the operational semantics evaluates signal values. We see two discontinuities. In particular, the signal value jumps from 1.0 to 3.0 at time 1.0 due to the discrete event $s_2(1.0, 1)$, and then resets to -2.0 at time 2.0 due to the discrete event $s_3(2.0, 2)$.

The above figure is 2-dimensional and has difficulty to show the details of how signal value changes along the index dimension. We give the following example, shown in Figure 3.11, to explicitly show how the output signal value $s'(t)$ varies along the indexes at time 1 and 2 respectively when provided with slightly more complicated DE input signals s_2 and s_3. We assume $s'(0, 0) = 0$.

[1]To be more precise, the discrete events in the DE input signals affect the internal *state* of the `Integrator` actor rather than the output signal values at particular tags.

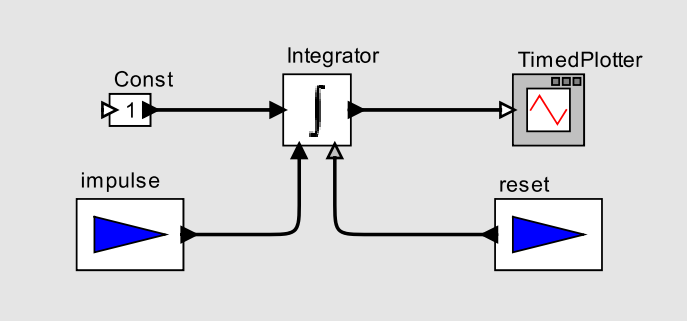

Figure 3.9. A model shows that an `Integrator` actor accepts one CT input signal and two DE input signals.

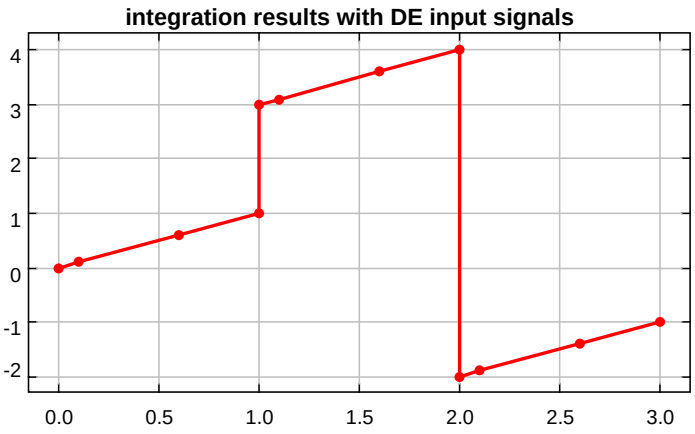

Figure 3.10. The simulation result of the model show in Figure 3.9.

n	0	1	2	3	4	5	6	7
$s_1(1,n)$	1	1	1	1	1	1	1	$\cdots$
$s_2(1,n)$	ε	2	ε	-1	ε	ε	ε	$\cdots$
$s_3(1,n)$	ε	ε	2	ε	3	ε	ε	$\cdots$
$s'(1,n)$	1	3	2	1	3	3	3	$\cdots$
$s_1(1,n)$	1	1	1	1	1	1	1	$\cdots$
$s_2(2,n)$	ε	-1	ε	1	-1	ε	ε	$\cdots$
$s_3(2,n)$	ε	ε	-2	ε	0	ε	ε	$\cdots$
$s'(2,n)$	4	3	-2	-1	0	0	0	$\cdots$

Figure 3.11. An example shows how the `Integrator` actor performs integration with two extra DE input signals s_2 and s_3. The signal s_1 is the derivative input signal and s' is the output signal with an initial value at tag (0, 0) as 0.

It is easy to see that the output signal has discontinuities only at tags where discrete events are present. For example, the signal value changes from 1 to 3 at tag $(1, 1)$, where $s_2(1, 1) = 2 \neq \varepsilon$.

There is a subtlety about the coexistence of the impulse event and reset event, such as at tag $(2, 4)$. In this case, the reset event has higher priority and determines the integrator state, therefore $s'(2, 4) = 0$. This is exactly the reason why we require a *left-open* interval $(t', t]$ in

$$t_i \in (\,(t', t] \cap T_2\,) \text{ in the last summation term } \sum_{t_i \in (\,(t', t] \cap T_2\,)} s_2(t_i)$$

from Equation (3.2). In particular, if $t_i = t'$, meaning both impulse and reset signals have an event at t', then $s_2(t')$ is excluded from integration.

3.3.2 Transient States

In the previous subsection, all discrete events of DE input signals are given beforehand. In this subsection, we introduce a mechanism to construct simultaneous events in DE signals at run-time. The mechanism uses *transient states* in modal models.

In fact, we have used transient states in the Newton's cradle model in Chapter 1 to model collisions. In this subsection, we will use a much simpler model for more detailed explanation. Figure 3.12 shows a model that uses a modal model with transient states to create simultaneous events in DE signals, which in turn create discontinuities in the CT output signal from the modal model.

The `SignalWithGlitches` actor implements the modal model. Figure 3.13 shows the inside details of the `SignalWithGlitches` actor. There are totally 5 states inside the modal model. The refinement of the `init` state produces a CCT signal as shown in Figure 3.14. The other states share the same refinement as shown in Figure 3.15, which simply outputs a constant CCT signal whose value is specified by the `output` parameter of a `Const` actor. The states, `state1`, `state2`, and `state3`, are transient states, because the guards on their outgoing transitions are always true. As explained earlier, the semantics of an enabled transition is that when guard is true the transition must be taken, therefore the time elapsed in the state is *zero*. Similarly, the `run` state in the Newton's cradle model shown in Figure 1.12 is also a transient state when collisions happen.

When the `output` signal value reaches the 1.5, the transition from `init` to `state1` is enabled. After this transition is executed, the outgoing transitions of the transient states will be executed in sequence at the same time point. Each enabled transition corresponds to a discrete event. When an

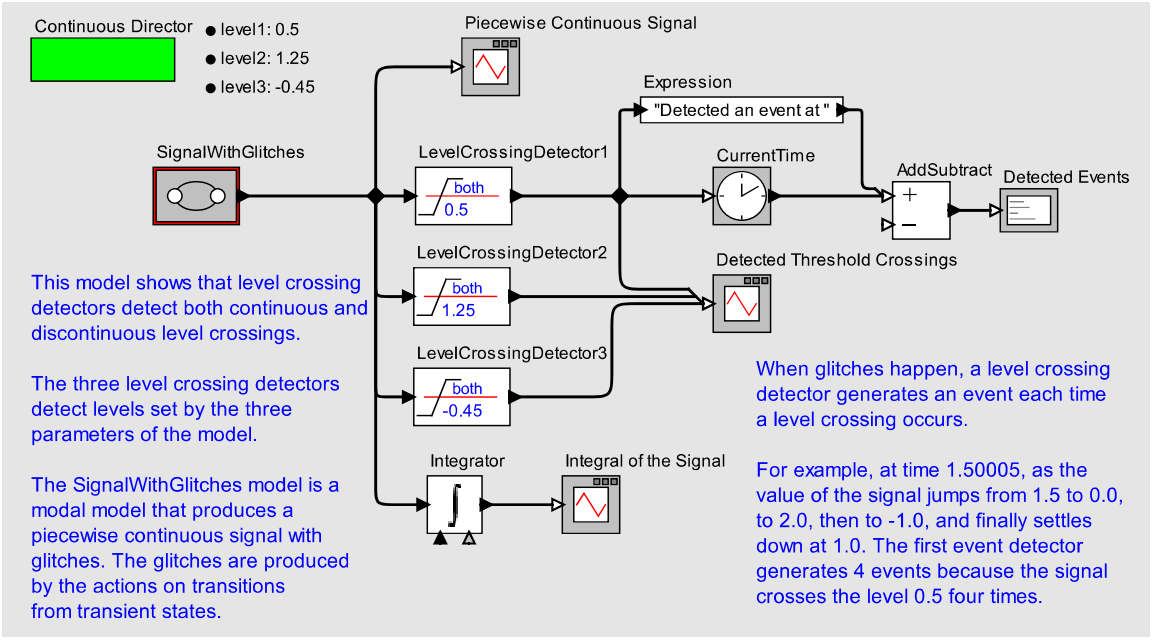

Figure 3.12. A model illustrates that *transient states* cause discontinuities in CT signals.

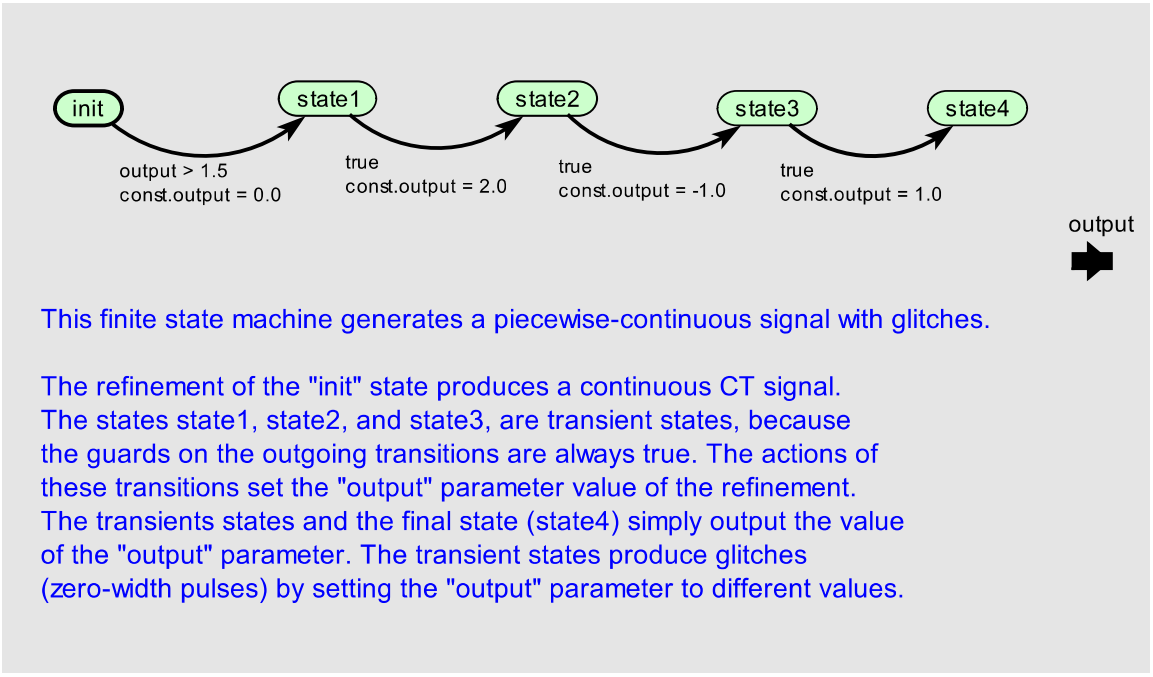

Figure 3.13. The inside details of the `SignalWithGlitches` actor in Figure 3.12, which is a *modal model* with three transient states: `state1`, `state2`, and `state3`.

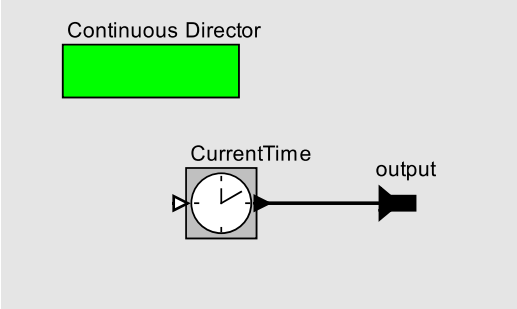

Figure 3.14. The refinement of the `init` state in Figure 3.13.

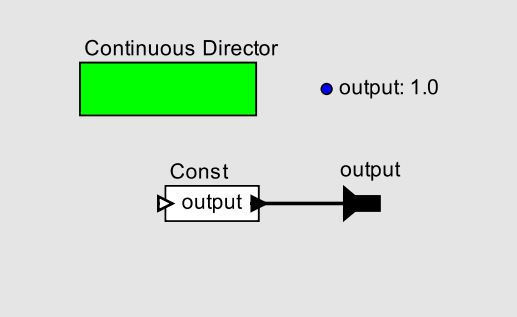

Figure 3.15. The refinement of the states in Figure 3.13 except the `init` state.

enabled transition is taken, its `action` changes the `output` parameter value such that the value of the CT output signal is changed. There are totally 4 transitions that will be executed in sequence, causing the signal value to change 4 times as follows: $1.5 \longrightarrow 0.0 \longrightarrow 2.0 \longrightarrow -1.0 \longrightarrow 1.0$. When the `state4` is reached, the output value stays at 1.0. Figure 3.16 shows the `output` signal, where the dots in figure are the computed values. There are totally 5 different values at time approximately 1.5, caused by the transitions between transient states.

Figure 3.17 shows the CCT output signal of the `Integrator` actor in Figure 3.12. Although the input signal, i.e. the `output` signal from the `SignalWithGlitches` actor, has discontinuities, the CCT output signal of the `Integrator` actor does not show any glitches at all because the discontinuities have zero time duration and contribute nothing to the integral.

Figure 3.16 only shows how the signal evolves along the time dimension, and it is hard to tell the exact details of how the `output` signal value changes at discontinuities. We use the `LevelCrossingDetector` actor as a probe to explore the mystery. There are three `LevelCrossingDetector` actors in Figure 3.13, which detect thresholds 0.5, 1.25, and -0.45 respectively. The detected level crossings, which are discrete events, are shown as dots in Figure 3.18. Note that these level crossings are detected not only during continuous dynamics but also at discontinuities. For example, the discrete events near time 0.5 and 1.25. Moreover, multiple dots (discrete events) are stacked together. This is revealed by Figure 3.19, which shows that the `LevelCrossingDetector1` actors detects totally 5 level crossings: one happens at time 0.5 and the other four happen at time around 1.5. Note that the figure shows 1.50005 rather than exactly 1.5 because a `LevelCrossingDetector` actor has to overshoot the time a little bit to ensure a level crossing happens.

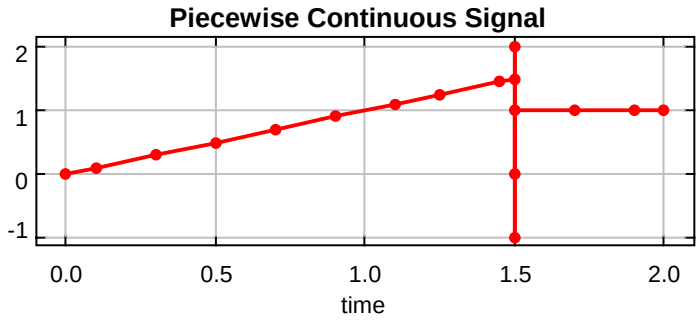

Figure 3.16. The output CT signal from the `SignalWithGlitches` actor in Figure 3.12.

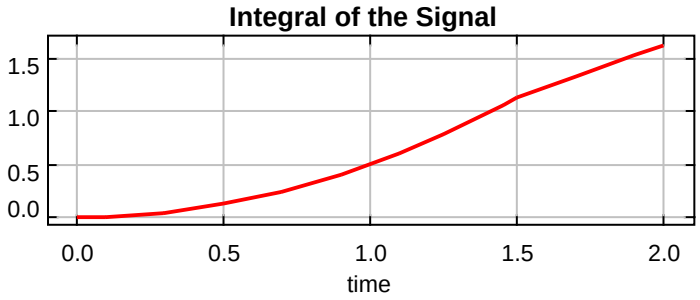

Figure 3.17. The output CCT signal from the `Integrator` actor in Figure 3.12.

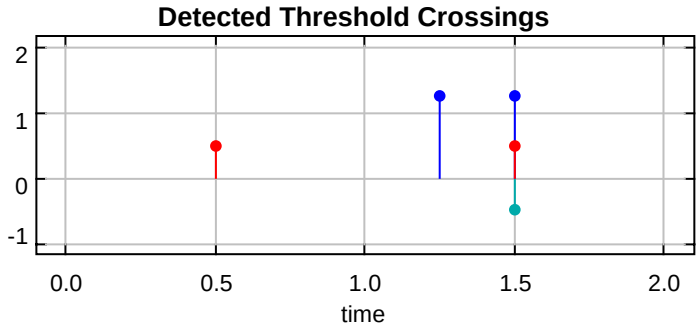

Figure 3.18. The output DE signal from the three `LevelCrossingDetector` actors in Figure 3.12.

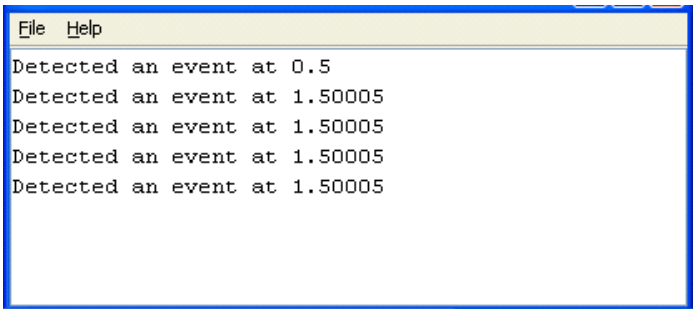

Figure 3.19. This figure shows that the `LevelCrossingDetector1` actor in Figure 3.12 detects 5 level crossings.

Fundamentally, the way that transient states generate simultaneous discrete events is the same as what Figure 3.20 shows, where a DE signals is fed into a model consisting of a feedback loop with *zero* time delay. Recall that the zero time delay causes more events at tags with bigger indexes, which generates simultaneous discrete events.

One difficulty of using the technique is that it lacks an effective way to control how many discrete events, i.e., to specify the stop condition. In fact, an execution of the model in Figure 3.20 will generate a chattering Zeno signal as shown in Figure 3.21 and Figure 3.22, where an infinite number of discrete events are happening at the same time point. While the modal model, which intrinsically is a finite state machine, is excellent at modeling and handling such logic control. Consequently, the combination of transient states with modal model provides more flexibility to generate simultaneous discrete events.

Many hybrid system simulators will require a transient state to remain active for at least one time step of the ODE solver, where the step size is determined by the ODE solver. This effectively results in nondeterministic behavior, since the programmer is typically unaware of the mechanisms that are used to define the time steps. In the above semantics, this would be incorrect behavior. Moreover, it is a poor model for the behavior of software, since it neither models the actual time taken by software nor provides a usable abstraction, such as the synchrony hypothesis [16].

The semantics of transients states makes it possible to model a sequence of software-based actions, which could, for example, be used to model software-based controllers. In particular, sequences of mode transitions can be used to model the software execution sequence, which are typically ordered but not timed. This fits the realities of software, where timing is not part of the semantics, and is consistent with abstractions for software that are increasingly used for embedded software such as synchronous languages [16] and time-triggered languages [50].

The semantics of transient states is more important for model-based design, where the support of modularity and hierarchical composition is most important for the success of the model-based design technique. When composing several hybrid systems together, their interactions can easily cause transient states inside individual hybrid system. Unless a hybrid system is fully tested and verified to have no transient states for any and all possible inputs, we have to deal with the effects and consequences of composition. In which case, a clean and rigorous semantics to deal with transient states, or more general, simultaneous discrete events, is necessary.

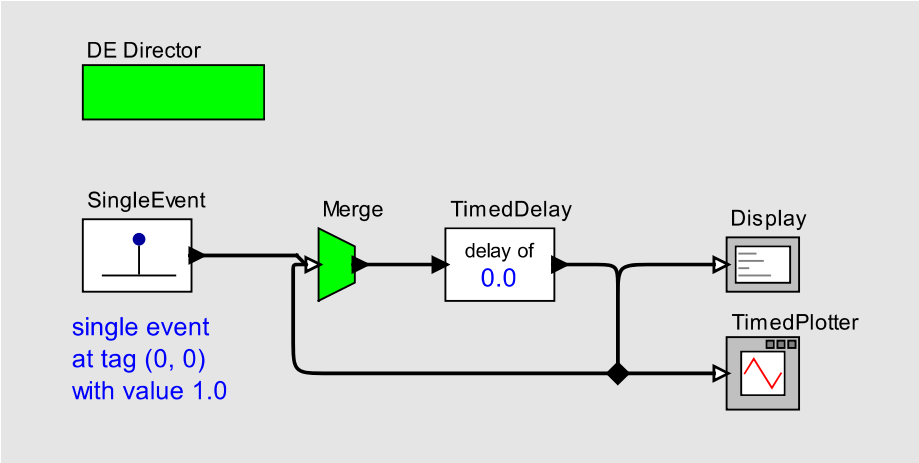

Figure 3.20. A model shows how to generate simultaneous discrete events with feedback loop, the `Merge` actor, and the `TimedDelay` actor with a *zero* time delay.

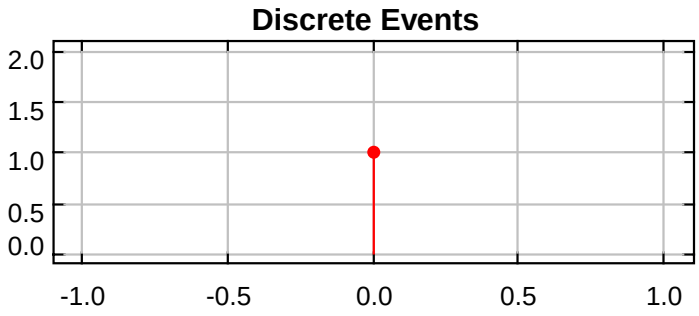

Figure 3.21. The plot shows discrete events happening at *same* time 0.0.

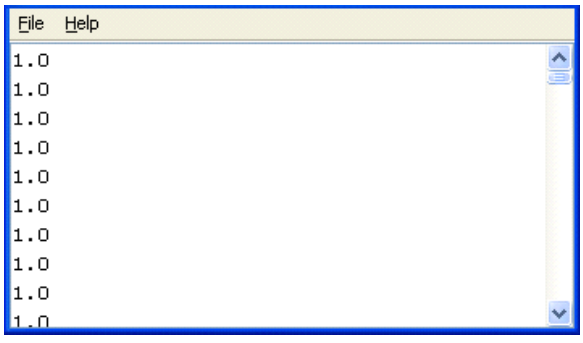

Figure 3.22. The display shows *multiple* (in fact, a infinite number of) events at the same time 0.0.

Chapter 4

Denotational Semantics

In computer science, denotational semantics is an approach to formalizing the semantics of programs by constructing mathematical objects (called denotations or meanings) which express the semantics of these systems. in this chapter, we will study the denotational semantics of compositions of actors.

We first study some important properties of actors, such as monotonicity, continuity, and causality. Then we will study some properties of the compositions of actors, such as *existence*, *uniqueness*, and *liveness*. Our mathematical tool is the well-known least fixed point theorem given in [31]. An important conclusion from this theorem is that a continuous function defined over CPO has a least fixed point solution. This conclusion is very important for us in that both actors and signals in this dissertation are defined as functions. Therefore, the semantics of a composition of actors, which is also a function resulting from composition of functions, is just the least fixed point solution of that function.

This least fixed point theorem also gives a constructive iteration procedure to find the least fixed point solution. In this section, we will apply this denotational semantics to some simple examples to illustrate how the iterative procedure constructs signals by extending their domains. This chapter serves as a foundation for developing an operational semantics in the next chapter.

4.1 Composition of Actors

In Section 3.2, we gave a few commonly used actors for hybrid systems. In this section, we study the general properties of these actors and then reason about some properties of compositions of these actors. We first study how to relate a composition of actors to composition of functions.

Figure 4.1(a) shows a network of actors with *sequential*, *parallel*, and *feedback* compositions. Following [64, 97], we introduce a level of hierarchy and convert the network into a composite actor without feedback compositions internally as shown in Figure 4.1(b). The composite actor then can be abstracted and treated as a (single and atomic) actor as shown in Figure 4.1(c), which can be used for future composition.

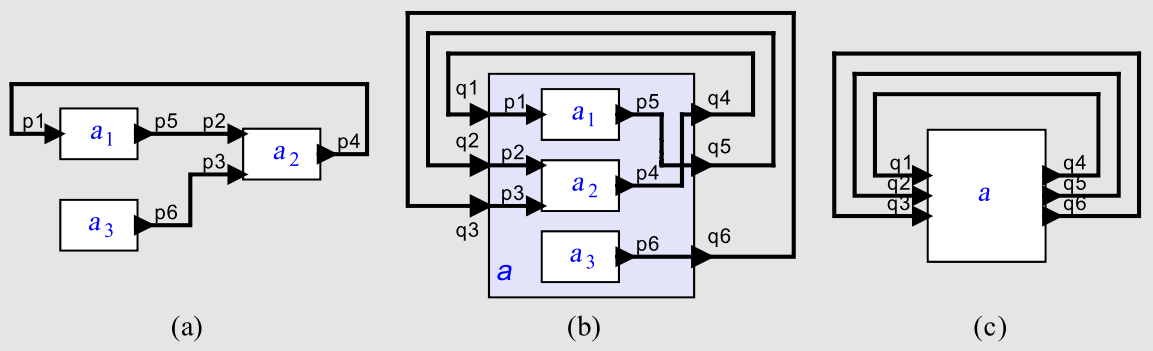

Figure 4.1. A composition of three actors and its interpretation as a feedback system. $P = \{p1, p2, p3, p4, p5, p6\}$ is the set of ports contained by the composite actor a. $Q = \{q1, q2, q3, q4, q5, q6\}$ is the set of external ports of a.

The procedure for converting a network of actors into an composite actor is simple and straightforward, and applies to any network [64, 97].[1]

Procedure 2 *Conversion Procedure:*

1. *For each signal in the network, create an input port and an output port. These ports are then the externally visible ports of the new composite actor. For example, in Figure 4.1(a), the signal going from p5 to p2 induces ports q5 and q2 in Figures 4.1(b) and (c). The original ports are all hidden inside and invisible from the outside of the composite actor, such as ports p2 and p5.*

2. *From the inside of the composite actor, connect the output port providing the signal value (p5*

[1] Here we only consider *closed* networks of actors which have no external input signals. The aforementioned procedure can be also applied to *open* networks. Details can be found in [69].

in this example) to the new output port ($q5$) *and connect the new input port* ($q2$) *to all input ports that receive the signal* ($p2$).

3. *Connect the new output port that provides signal values to the outside* ($q5$) *to the new input port that receives signal values* ($q2$) *from outside of the composite actor (the feedback connection).*

If actors a_1, a_2, and a_3 in Figure 4.1(b) are functional, then the composite actor a in Figure 4.1(c) is also functional. Let F_a denote the actor function for actor a, it has the form

$$F_a \colon \mathcal{S}^3 \to \mathcal{S}^3.$$

Recall that a `Connector` actor requires the signals at its input ports to be the same as the signals at output ports. Therefore, the connectors in Figure 4.1(c) require the input signals of a to be the same as the output signals. Thus the behavior of the feedback composition in Figure 4.1(c) is a fixed point of F_a. That is,

$$F_a(\mathbf{s}) = \mathbf{s}. \tag{4.1}$$

4.2 Least Fixed Point Theorem

Next we study some properties of the fixed point solution of Equation (4.1), such as *existence*, *uniqueness*, and *liveness*. In order to do this, we need to study some general properties of actors.

The general properties for actors presented in this section are defined with the prefix order of the signals given in Definition (3).

Definition 31 (Monotonicity) An actor $a \colon \mathcal{S}^n \rightharpoonup \mathcal{S}^m$ is **monotonic** if it is order-preserving,

$$\forall\, \mathbf{r}, \mathbf{s} \in \mathcal{S}^n,\ \mathbf{r} \sqsubseteq \mathbf{s} \implies a(\mathbf{r}) \sqsubseteq a(\mathbf{s}).$$

Definition 32 (Scott Continuity) An actor $a \colon \mathcal{S}^n \rightharpoonup \mathcal{S}^m$ is **(Scott) continuous** if for any directed set $D \subseteq \mathcal{S}^n$, $a(D) \subseteq \mathcal{S}^m$ is a directed set, and

$$a(\bigvee D) = \bigvee a(D).$$

Continuity implies monotonicity [31]. Monotonic functions and continuous functions are composable.

A well-known fixed point theorem in [31] (Chap. 8) is given below.

Theorem 1 (Least Fixed Point Theorem) *If $f : A \to A$ is a continuous function defined on a CPO A, then f has a* ***least fixed point*** *given by*

$$\bigvee\{f^n(a_\perp) \mid n \in \mathcal{N}\}, \tag{4.2}$$

where $a_\perp$ is the bottom element of CPO A.

The proof of this theorem is omitted and can be found in [31]. This theorem is very useful for deciding the existence and uniqueness of the solutions for Equation (4.1), and determining the existence and uniqueness of the behaviors of compositions of actors. In particular, if all component actors of a composition are continuous, note that the signal set $\mathcal{S}$ forms a CPO, then the composition of actors has a unique behavior, given by the *least fixed point* solution of the function specified by the composite actor formed as described in Procedure (2).

All actors introduced in Section 3.2 are monotonic and continuous, therefore, a hybrid system composed from those actors always has a unique behavior.

4.3 Examples

This fixed point theorem also provides a constructive way to compute the unique behavior. In this subsection, we will apply this denotational semantics to some simple examples to illustrate how the iterative procedure constructs signals by extending their domains.

In order to apply the constructive procedure, we first follow Procedure (2) and transform an example model into a composite actor that has feedback connections from the outside but not inside. We call the composite actor a and the actor function $F_a : \mathcal{S}^n \to \mathcal{S}^n$, where n is the number of input (or output) signals.

To ease our discussions, we give a few common notations that will be used throughout this section. Let $\mathbf{s} = (s_1, s_2, \cdots, s_n)$, and

$$\mathbf{s}^i = (s_1^i, s_2^i, \cdots, s_n^i) = F_a^i(\mathbf{s}) = F_a(\mathbf{s}^{i-1}).$$

Specifically, when $i = 0$,

$$\mathbf{s}^0 = (\underbrace{s_\perp, s_\perp, \cdots, s_\perp}_{n}),$$

where $\text{dom}(s_\perp) = \emptyset$. We will always start with the tuple $\mathbf{s}^0$ to apply the iterative procedure to find the least fixed point solution to F_a. This complies with the constructive procedure defined in Theroem (1). Therefore, $\mathbf{s}^i$ is the result of applying F_a to $\mathbf{s}^0$ i times.

Without special comments, we assume that the domain of the signal tuple $\mathbf{s}$ starts with tag $t_\perp = (0, 0)$ and $\text{dom}(\mathbf{s}) = \mathcal{T}$.

4.3.1 CT Example Without Feedback

We first investigate a very simple CT model without a feedback loop as shown in Figure 4.2(a). Figure 4.2(b) illustrates the transformation of this model into a feedback system following Procedure (2). For simplicity, we ignore the unimportant port names but instead name the signals. Figure 4.2(c) shows the result after transformation: a composite actor F_a with two input ports and two output ports, where the output signals are fed back as the input signals.

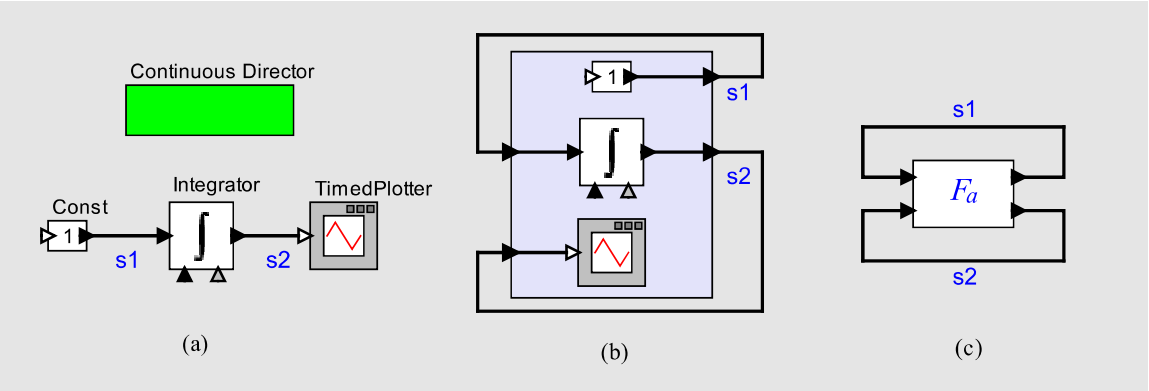

Figure 4.2. A simple CT model without feedback loop and its interpretation as a feedback system.

As we discussed in Section 4.1, the behavior of the feedback composition in Figure 4.2(c) is a fixed point of F_a. That is,

$$F_a(\mathbf{s}) = \mathbf{s}, \quad \mathbf{s} = (s_1, s_2) \in \mathcal{S}^2.$$

Following the iterative procedure given in Theroem (1), we construct the output signal tuple $\mathbf{s} = (s_1, s_2)$ from $\mathbf{s}^0 = (s_\perp, s_\perp)$.

With $\mathbf{s}^0$ as the input signal tuple, the first firing of the F_a actor generates $\mathbf{s}^1 = (s_1^1, s_2^1)$. The signal s_1^1 is the output signal from a `Const` source actor, therefore

$$s_1^1(t) = 1 \text{ and } \text{dom}(s_1^1) = \mathcal{T}.$$

In other words, the signal s_1 is completely resolved (called *complete*). Also because the `Const` actor is monotonic, $\forall k \in \mathcal{N}, s_1^k = s_1^1$, meaning more firings of F_a have no effect on s_1.

The domain of signal s_2^1 also gets extended, in particular, $\text{dom}(s_2^1) = \{t_\perp\}$. This singleton set means that signal s_2^1 has a single event, which takes the initial state of the `Integrator` actor as the initial value of the signal at time 0.

The `TimedPlotter` actor does not have any output ports and is called a `Sink` actor. Its only functionality is to plot signals on a display. In general, a `Sink` actor defines a simple function and has no outputs to affect other actors, therefore we exclude `Sink` actors from our discussions in most cases if without special comments.

With $\mathbf{s}^1$ as the input signal tuple, the second firing of the F_a actor generates $\mathbf{s}^2 = (s_1^2, s_2^2)$. We have $s_1^2 = s_1^1$ as explained before. Following the definition of the `Integrator` actor in Section 3.2.3, since the derivative signal s_1^1 is complete, the output signal s_2^2 is also complete. In particular,

$$s_2^2(t) = \texttt{time}(t) \text{ and } \text{dom}(s_2^2) = \mathcal{T}.$$

Consequently, $\forall k \in \mathcal{N}, s_2^k = s_2^2$, meaning more applications of F_a have no effect on s_2 also.

The signal tuple $\mathbf{s}^2$ is a fixed point to actor F_a. According to Theroem (1), it is the least fixed point. Therefore, $\mathbf{s}^2$ is the behavior of the model shown in Figure 4.2(a). A portion of the least fixed point solution of signal s_2 (until time 5.0) is shown in Figure 4.3. The dots in the figure are computed values from the operational semantics and will be explained later in next chapter.

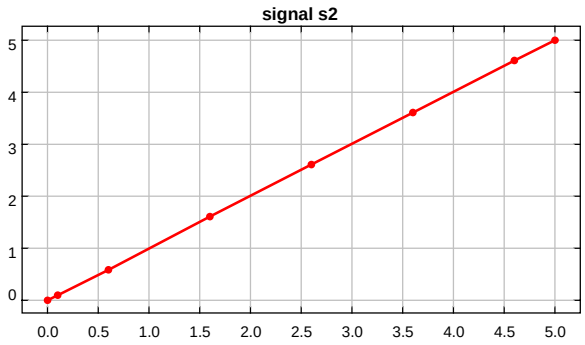

Figure 4.3. Signal s_2 as the result from an execution of the CT model in Figure 4.2(a).

4.3.2 DE Example Without Feedback

Now let us examine another simple model with a discrete-event semantics as shown in Figure 4.4(a). Similarly to the CT model in Section 4.3.1, this model does not have feedback. Using the

same technique, we transform the DE model into a composite actor F_a with feedback composition from outside but not inside shown in Figure 4.4(b).

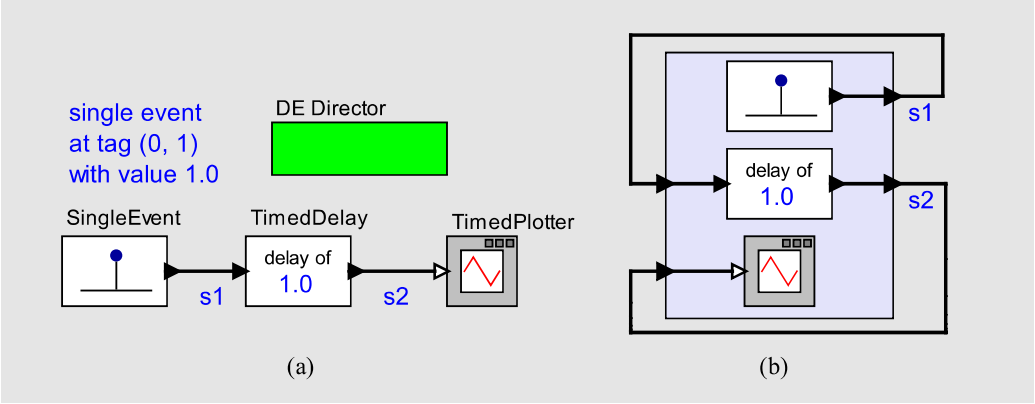

Figure 4.4. A simple DE model without feedback loop and its interpretation as a feedback system.

Again, the behavior of this DE model is defined by the least fixed point solution of actor function F_a. That is,

$$F_a(\mathbf{s}) = \mathbf{s}, \quad \mathbf{s} = (s_1, s_2) \in \mathcal{S}^2.$$

Following the iterative procedure given in Theroem (1), we construct the output signal tuple $\mathbf{s} = (s_1, s_2)$ from the empty signal tuple $\mathbf{s}^0 = (s_\perp, s_\perp)$.

With $\mathbf{s}^0$ as the input signal tuple, the first application of the F_a function generates $\mathbf{s}^1 = (s_1^1, s_2^1)$. The signal s_1^1 is the output signal from a `SingleEvent` actor, therefore,

$$s_1^1(r, n) = \begin{cases} 1 & \text{if} \quad r = 0 \text{ and } n = 1 \\ \varepsilon & \text{otherwise} \end{cases} \quad \text{and} \ \ \text{dom}(s_1^1) = \mathcal{T}. \tag{4.3}$$

The signal s_1^1 is complete. Therefore, $\forall k \in \mathcal{N}, s_1^k = s_1^1$.

The domain of signal s_2^1 gets extended, in particular, $\text{dom}(s_2^1) = [t_\perp, (1, 0)]$. This is because the `TimedDelay` actor has a time delay of 1.0. According to its definition in Section 3.2.5, its output signal is known to have an absence value until tag $(1, 0)$.

With $\mathbf{s}^1$ as the input signal tuple, the second application of the F_a function generates $\mathbf{s}^2 = (s_1^2, s_2^2)$. We have $s_1^2 = s_1^1$ as explained before. Following the definition of the `TimedDelay` actor in Section 3.2.5 again, since the derivative signal s_1^1 is complete, the output signal s_2^2 is also complete.

In particular,

$$s_2^2(r,n) = \begin{cases} 1 & \text{if} \quad r = 1 \text{ and } n = 1 \\ \varepsilon & \text{otherwise} \end{cases} \quad \text{and} \ \ \text{dom}(s_2^2) = \mathcal{T}.$$

The `TimedDelay` actor is monotonic. Consequently, $\forall k \in \mathcal{N}, s_2^k = s_2^2$. Therefore, $\mathbf{s}^2$ is a fixed point to function F_a and gives the behavior of the model shown in Figure 4.4(a). A portion of the least fixed point solution of signal s_2 (until time 5.0) is shown in Figure 4.5. There is only one discrete event at time 1.0 and the events with absence values are not explicitly shown.

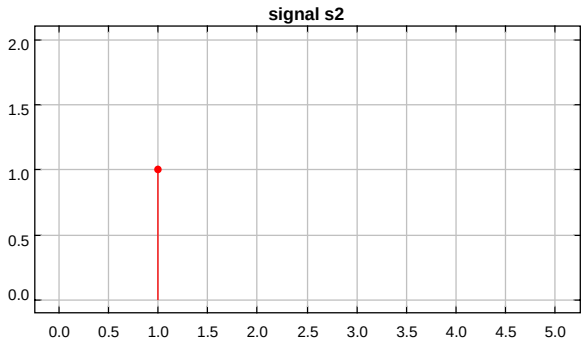

Figure 4.5. Signal s_2 as the result from an execution of the DE model in Figure 4.4(a).

So far we have only considered compositions without feedback loops, the construction process of the signals is simple and straightforward. For compositions with feedback, the process is a little more complicated. We investigate a few more examples next.

4.3.3 CT Example With Feedback

Figure 4.6(a) is a CT model with feedback composition, where the output signal from the `Integrator` actor s_3 and the output signal from the `Const` actor s_1 together determine the input derivative signal s_2 of the `Integrator` actor. Specifically,

$$s_2(t) = s_1(t) - s_3(t), \ \forall t \in \mathcal{T}. \tag{4.4}$$

Figure 4.6(b) shows the transformation of this model into a composite actor F_a with feedback compositions from outside only.

The behavior of this model is defined by the least fixed point solution of actor function F_a,

$$F_a(\mathbf{s}) = \mathbf{s}, \quad \mathbf{s} = (s_1, s_2, s_3) \in \mathcal{S}^3.$$

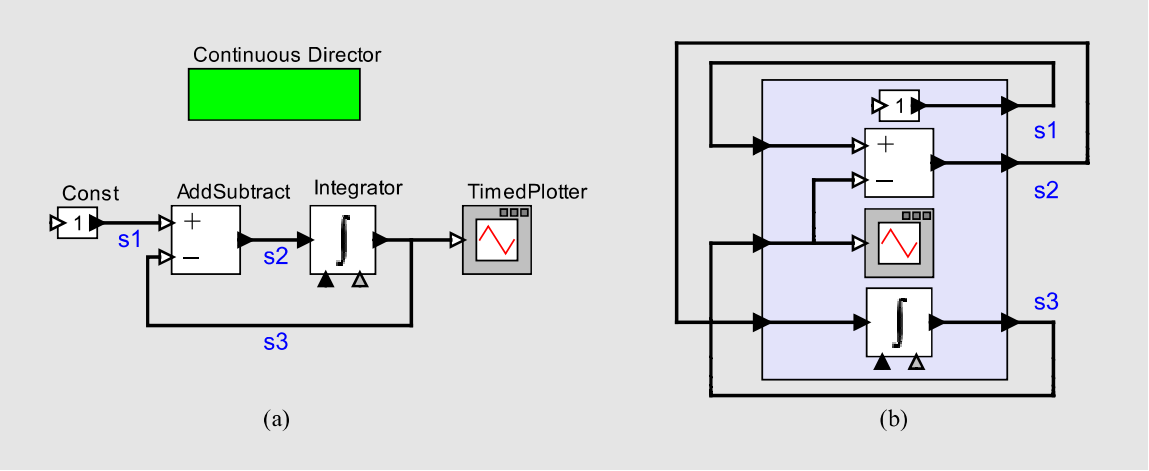

Figure 4.6. A simple CT model with feedback loop and its interpretation as a feedback system.

Following the iterative procedure given in Theroem (1), we construct the output signal tuple $\mathbf{s} = (s_1, s_2, s_3)$ from the empty signal tuple $\mathbf{s}^0 = (s_\perp, s_\perp, s_\perp)$.

With $\mathbf{s}^0$ as the input signal tuple, the first firing of the F_a actor generates $\mathbf{s}^1 = (s_1^1, s_2^1, s_3^1)$. The signal s_1^1 is the output signal from a `Const` source actor, and

$$s_1^1(t) = 1 \text{ and } \mathrm{dom}(s_1^1) = \mathcal{T}.$$

The signal s_1^1 is complete and $\forall k \in \mathcal{N}, s_1^k = s_1^1$.

The domain of signal s_3^1, $\mathrm{dom}(s_3^1)$, becomes $\{t_\perp\}$, meaning that signal s_3^1 contains a single event at tag $t_\perp$. The domain of signal s_2^1 does not change according to the definition of `AddSubtract` actor in Section 3.2.1, where an empty output signal results from empty input signals. Therefore $s_2^1 = s_\perp$.

With $\mathbf{s}^1$ as the input signal tuple, the 2^{nd} firing of the F_a actor generates $\mathbf{s}^2 = (s_1^2, s_2^2, s_3^2)$. We have $s_1^2 = s_1^1$ as explained before. Although s_1^1 is a complete signal, because $\mathrm{dom}(s_3^1) = \{t_\perp\}$, according to Equation (4.4), we have $\mathrm{dom}(s_2^2) = \mathrm{dom}(s_1^1) \cap \mathrm{dom}(s_3^1) = \{t_\perp\}$. Finally, $s_3^2 = s_3^1$ due to s_2^1 as an empty signal.

With $\mathbf{s}^2$ as the input signal tuple, the 3^{rd} firing of the F_a actor generates $\mathbf{s}^3 = (s_1^3, s_2^3, s_3^3)$, where $s_1^3 = s_1^1$. Because $s_3^2 = s_3^1$, $s_2^3 = s_2^2$. As we explained in Section 3.2.3, if the dynamics equations satisfy a global Lipschitz condition, the domain of s_3^3 will be expanded by ODE solvers. Let $\mathcal{T}' = \mathrm{dom}(s_3^3)$, then $\{t_\perp\} \subset \mathcal{T}' \subset \mathcal{T}$. In particular,

$$\mathtt{time}(\bigvee \mathcal{T}') = \delta > 0.$$

If δ has a positive lower bound, then time is guaranteed to diverge.

When F_a is applied on $\mathbf{s}^3$, note the new domain of s_3^3, signal s_2^4 will have a bigger domain $\text{dom}(s_2^4) = \text{dom}(s_1^3) \cap \text{dom}(s_3^3) = \mathcal{T}'$. This will allow the next firing of F_a to further expand the domain of signal s_3^4. Keep repeating this process, the domains of all signals will keep expanding. When the domains of all signals cover the whole tag set, we find the least fixed point solution.

A portion of the least fixed point solution of signal s_3 (until time 5.0) is shown in Figure 4.7. The dots in the figure are computed values from operational semantics. These values comply with the analytic solution of the signal, given below.

$$\begin{aligned} f(r) &= 1 - e^{-r}, \\ s_3(t) &= f(\texttt{time}(t)). \end{aligned}$$

We will explain how to calculate these values and how to choose intervals between these dots later in the next chapter.

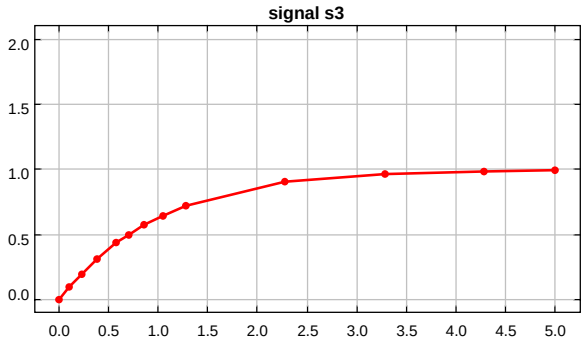

Figure 4.7. Signal s_3 as the result from an execution of the CT model in Figure 4.6(a).

4.3.4 DE Example With Feedback

Lastly, we give a DE example with feedback and show how the constructive procedure to find the least fixed point solution works. This example is much simpler than the CT example with feedback. The intuitive reasoning of the model is that a single event is circulating around the loop, and for each circulation, the event is delayed by 1 time unit.

Figure 4.8 is a DE model with feedback composition, where the merged result of the output signal from the `TimedDelay` actor s_3 and the output signal from the `SingleEvent` actor s_1 gives the

input signal of the `TimedDelay` actor s_2. Specifically,

$$s_2 = \texttt{Merge}(s_1, s_3). \tag{4.5}$$

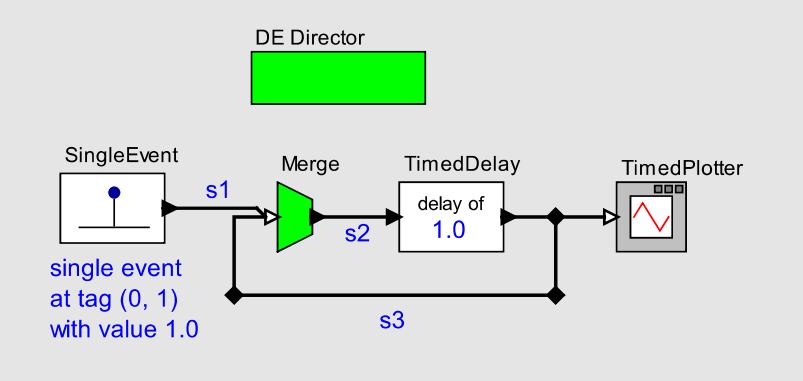

Figure 4.8. A simple DE model with feedback loop.

Similarly to the previous CT model with feedback, the behavior of this model is defined by the least fixed point solution of actor function F_a,

$$F_a(\mathbf{s}) = \mathbf{s}, \quad \mathbf{s} = (s_1, s_2, s_3) \in \mathcal{S}^3.$$

Following the iterative procedure given in Theroem (1), we construct the output signal tuple $\mathbf{s} = (s_1, s_2, s_3)$ from the empty signal tuple $\mathbf{s}^0 = (s_\perp, s_\perp, s_\perp)$.

With $\mathbf{s}^0$ as the input signal tuple, the first firing of the F_a actor generates $\mathbf{s}^1 = (s_1^1, s_2^1, s_3^1)$. The signal s_1^1 is the output signal from the `SingleEvent` source actor and the same as defined in Equation (4.3). The signal s_1^1 is complete and $\forall k \in \mathcal{N}, s_1^k = s_1^1$.

Assume that the `Merge` actor does *simple merge* as defined in Section 3.2.4 [2], then

$$s_2^1 = \texttt{Merge}(s_1^0, s_3^0) = \texttt{Merge}(s_\perp, s_\perp) = s_\perp.$$

The domain of signal s_2^1 does not change and remains as the empty set $\emptyset$.

For the output signal of the `TimedDelay` actor s_3^1, $\text{dom}(s_3^1) = [t_\perp, (1, 0))$ because of the time delay of 1.0. The domain of signal s_3^1 is open on the right because the event of signal s_2^0 at tag $(0, 0)$ is unknown.

[2] In this example, *lossless merge* will deliver the same result since the two input signals never have a presence event at the same time.

With $\mathbf{s}^1$ as the input signal tuple, the 2^{nd} firing of the F_a actor generates $\mathbf{s}^2 = (s_1^2, s_2^2, s_3^2)$. We have $s_1^2 = s_1^1$ as explained before. Now we have $s_2^2 = \texttt{Merge}(s_1^1, s_3^1)$. Therefore $\text{dom}(s_2^2) = \text{dom}(s_3^1) = [t_\perp, (1,0))$ as the merge result. Signal s_2^2 now has an event at tag $(0,1)$.

Finally, because s_2^1 does not change, s_3^2 does not change, i.e., $\text{dom}(s_3^2) = [t_\perp, (1,0)]$.

With $\mathbf{s}^2$ as the input signal tuple, the 3^{rd} firing of the F_a actor generates $\mathbf{s}^3 = (s_1^3, s_2^3, s_3^3)$, where $s_1^3 = s_1^1$. $s_2^3 = \texttt{Merge}(s_1^2, s_3^2)$, because $s_3^2 = s_3^1$ and $s_1^2 = s_1^1$, $s_2^3 = s_2^2$. Since now $\text{dom}(s_2^2) = [t_\perp, (1,0))$ has a bigger domain, the domain of signal s_3^3 expands to $[t_\perp, (2,0))$ because of the time delay of 1.0. s_3^3 now has an event at tag $(1,1)$, resulted from the event in s_2^2 at tag $(0,1)$ being delayed.

Keep repeating the above process and it is easy to see that the domains of all signals keep expanding. When the domains of all signals cover the whole tag set, we find the least fixed point solution.

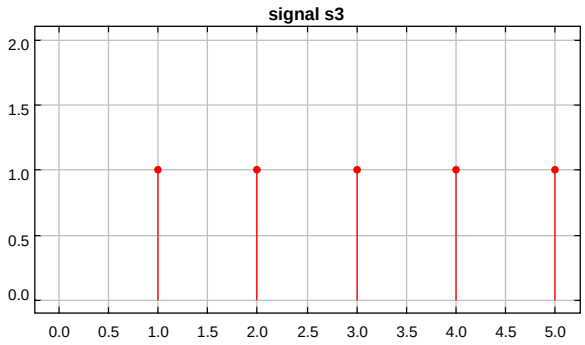

Figure 4.9. Signal s_3 as the result from an execution of the DE model in Figure 4.8.

A portion of the least fixed point solution of signal s_3 (until time 5.0) is shown in Figure 4.8. The dots in the figure indicate the delayed discrete events. The execution results comply with the analytic solution,

$$s_3(r,n) = \begin{cases} 1 & \text{if} \quad r \geq 1,\ r \in \mathcal{N} \text{ and } n = 1 \\ \varepsilon & \text{otherwise} \end{cases} \quad \text{and } \text{dom}(s_3) = \mathcal{T}.$$

4.4 Causality

In this section, we study the liveness problem. An actor is called *live* if its output signals are complete (defined over the whole tag set) given the input signals are complete [69]. We will use the *causality* property to decide whether a composition of actors is live or not.

Causality is about the relationship between causes and effects. Intuitively, for a causal actor, the effects cannot happen earlier than the causes. For a strictly causal actor, the effects must happen after the causes occur. Formally,

Definition 33 (Causality) An actor $a\colon \mathcal{S}^n \rightharpoonup \mathcal{S}^m$ is **causal** if for any signal $\mathbf{s} \in \mathcal{S}^n$,

$$\mathrm{dom}(\mathbf{s}) \subseteq \mathrm{dom}(a(\mathbf{s})).$$

An important conclusion about causal actors is that a causal actor is live [68]. All actors introduced in Section 3.2 are causal. Consequently, they are all live. However, in order to guarantee their compositions to be live, specifically feedback compositions, we need a more restrictive property defined below.

Definition 34 (Strict Causality) An actor $a\colon \mathcal{S}^n \rightharpoonup \mathcal{S}^m$ is **strictly causal** if for any signal $\mathbf{s} \in \mathcal{S}^n$,

$$\mathrm{dom}(\mathbf{s}) \subset \mathrm{dom}(a(\mathbf{s})).$$

The $\subset$ relation means strict subset.

A sufficient condition for reasoning about the liveness of a composition of actors is given as a theorem in [68]. The theorem states that given a totally ordered tag set and a network of causal and continuous actors where in every dependency loop in the network there is at least one *strictly causal* actor, then the network is a causal and continuous actor. Consequently, the network of actors is live.

Not all actors introduced in Section 3.2 are strictly causal. In particular, only source actors and a `TimedDelay` actor with a non-zero time delay are strictly causal. In fact, a source actor can be treated as a special `TimedDelay` actor with a time delay of *positive infinity*. Therefore, a direct conclusion from the above theorem is that for any composition of actors, if there is at least one `TimedDelay` actor with a non-zero time delay for each feedback loop, the composition actor is live.

It is easy to see why the `TimedDelay` actor is strictly causal. I will elaborate on explaining why some other actors are causal but not strictly causal.

The first actor is the `Integrator`. Claiming an `Integrator` to be not strictly causal may contradict with the common impression that a feedback loop with an `Integrator` actor is live. The reason is that we assume that the dynamical equations satisfy a global Lipschitz condition during the tag interval we are interested, specifically, the input CT signal as the derivative signal is bounded.

Under this assumption, the integrator can approximate the output signal value at a tag with a future time. This allows time to progress and in turn makes the `Integrator` actor and the feedback composition live. However, the amount of time increment depends on the value of input signal. In particular, if the value of the input CT signal keeps increasing without an upper bound, then the time increment will keep decreasing without a lower bound. As a result, the time may never exceed a certain time point. For example, let the input signal s be

$$s_i(r) = \frac{1}{1-r},\ r \in [0, 1.0) \quad \text{and} \quad s(t) = s_i(\texttt{time}(t)),\ t \in [(0,0),(1.0,0)).$$

As the integration proceeds, the least upper bound of the domain of the output signal gets closer to the tag $(1.0, 0)$, and the time increment gets infinitely smaller. As a consequence, the least upper bound of the domain of the output signal can never exceed $(1.0, 0)$. This integration process in fact corresponds to a genuinely Zeno behavior. In summary, the `Integrator` actor is causal but not strictly causal. If the ODE's to be solved satisfy a global Lipschitz condition along the interval for integration, then the `Integrator` is strictly causal.

The `TimedDelay` actor with a zero time delay is not strictly causal. Let the following signal s be the input signal,

$$s(r, n) = 1, r = 0.0 \text{ and } n \in \mathcal{N}.$$

Then the output signal s' will be

$$s'(r, n) = \begin{cases} 1 & \text{if} \quad n \geq 1 \\ 0 & \text{else} \end{cases},$$

where 0 is the default value. It is easy to tell that $\text{dom}(s') = \text{dom}(s)$. Therefore, the `TimedDelay` actor is not strictly causal. Consequently, the `VariableTimedDelay` is in general not strictly causal unless the amount of time delay has a lower bound $\delta' > 0$ such that $\delta(t) \geq \delta'$ for all tags.

The `Merge` actor defined above, whether the merge process is lossless or not, is not strictly causal. For example, let s_1 and s_2 be defined over a down set $[(0,0),(1,0)) \subset \mathcal{T}$, then the merged signal is defined over the same down set. According to Definition (34), the merge actor is only causal but not strictly causal.

4.4.1 Eventually Strict Causality

The strictly causality requires an actor to produce something from nothing as input. For example, if an actor a is strictly causal, then $\text{dom}(a(s_\perp))$ must not be an empty set, where $s_\perp$ is the empty signal. This is a fairly strong constraint.

We introduce an **eventually strict causality** below, which relaxes the strict causality.

Definition 35 (Eventually Strict Causality) An actor $a\colon \mathcal{S}^n \rightharpoonup \mathcal{S}^n$ is **eventually strictly causal** if for any signal $\mathbf{s} \in \mathcal{S}^n$, there exists an integer k, such that

$$\mathrm{dom}(\mathbf{s}) \subset \mathrm{dom}(a^k(\mathbf{s})),$$

where $a^k(\mathbf{s}) = \underbrace{a(a(\cdots a(\mathbf{s})))}_{k}$. We call the least of such k the **causality exponent** of the actor.

A strictly causal actor has its causality exponent $k = 1$.

The above definition of causality describes actors with *dynamic* causality properties. Specifically, an actor can *stutter* (being causal but not strictly causal) for a finite number of firings and then become strictly causal for next firing. This causality property is specially useful for reasoning about actors that produce simultaneous discrete events, such as a modal model with a finite number of transient states. We have the following conclusion.

Theorem 2 (Liveness of Composition) *Given a totally ordered tag set and a network of* causal *and* continuous *actors, if the transformed composition actor following Procedure (2) is eventually strictly causal, then the network of actors is live.*

Proof: Let a represent the composite actor. Suppose a is eventually strictly causal with a causality exponent k, then we can duplicate a for $k-1$ times and concatenate them together in sequence as shown in Figure 4.10 to form a single function a^k. The function a^k is strictly causal.

Following Theroem (1), since all actors in the composition are continuous, there is a unique behavior. Furthermore, the behavior can be constructively computed by iterating the function defined by the composition as shown in Equation (4.2).

Let $\mathbf{s}$ be the behavior constructed by iterating a and $\mathbf{s}'$ generated from iterating a^k. Then $\mathbf{s} = \mathbf{s}'$.

Following the theorem for the liveness of composition presented in [68], a^k is live. Thus, a is also live. □

A commonly used sufficient condition for a DE model to be non-Zeno is that if there is a lower bound $\delta > 0$ on the time between events, then there be no chattering Zeno conditions. Although this statement is rather obvious, the classical approach to proving it leverages some fairly sophisticated mathematics, constructing a metric space of signals using the so-called Cantor metric, and then invoking the Banach fixed point theorem [58].

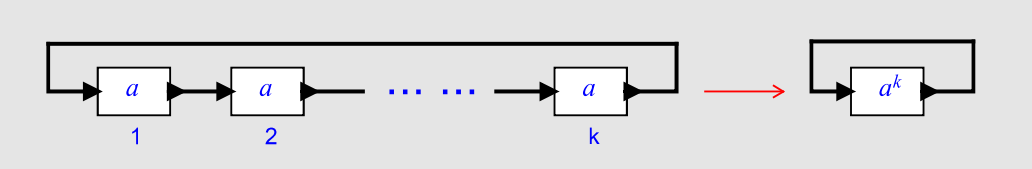

Figure 4.10. Concatenation of k identical function a results a single function a^k.

The theorem for the liveness of composition presented in [68] relaxed the above sufficient condition by taking into consideration the domain of signals. In particular, as long as the domains of all signals cover the whole tag set, which may not cover the whole time line or the whole universe, the model is not call Zeno. This makes sense in that if we are interested in a particular tag set, and signals are fully resolved on that set, then the computation can be terminated and we do not even bother to care about what happens outside the tag set.

Theroem (2) further relaxes the theorem for the liveness of composition presented in [68]. As an example, Figure 4.11 shows that a DE model containing a feedback composition is live with an eventually strictly causal actor. The modal model, called `EventuallyStrictlyCausalActor`, contains two modes of operation as shown in Figure 4.12, one mode delays the input signal for 1 time unit, and the other mode delays the input signal for 0 time. The modes switch based on the arrival of discrete events in the input signal and the `action` of an enabled transition sets the delay amount of the active mode.

Figure 4.13 shows the refinement shared by both `init` and `state` states, the input signal is delayed by the `TimedDelay` actor and the `Ramp` actor responds to each discrete event of the input signal by outputting its state and then increasing its state by 1. As we said before, the `TimedDelay` actor is strictly causal only if the amount of delay is a positive number. Therefore, the `EventuallyStrictlyCausalActor` actor will behave like a causal-only actor when the delay amount is 0. However, since the `init` and `state` states will be activated alternatively by the discrete events in the input signal, the `EventuallyStrictlyCausalActor` actor becomes *eventually strictly causal*.

Similar to the examples discussed in the Section 4.3, the behavior of this model is defined by the least fixed point solution of actor function F_a,

$$F_a(\mathbf{s}) = \mathbf{s}, \quad \mathbf{s} = (s_1, s_2, s_3) \in \mathcal{S}^3.$$

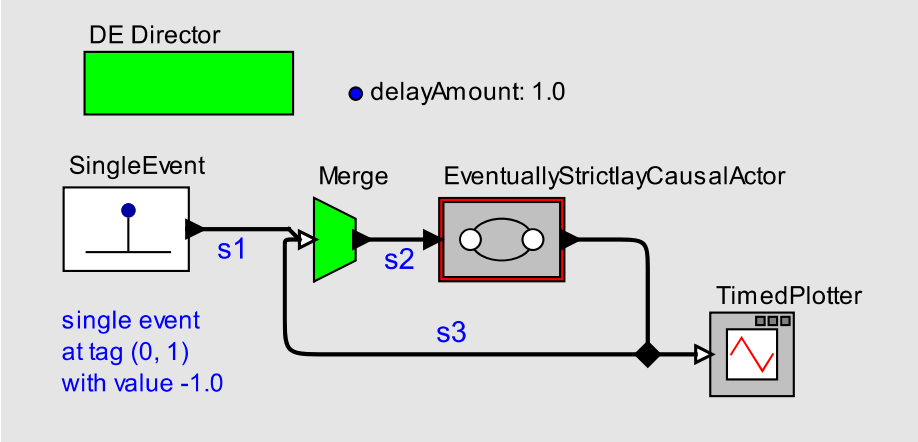

Figure 4.11. A model shows a feedback composition with an eventually strictly causal actor is live.

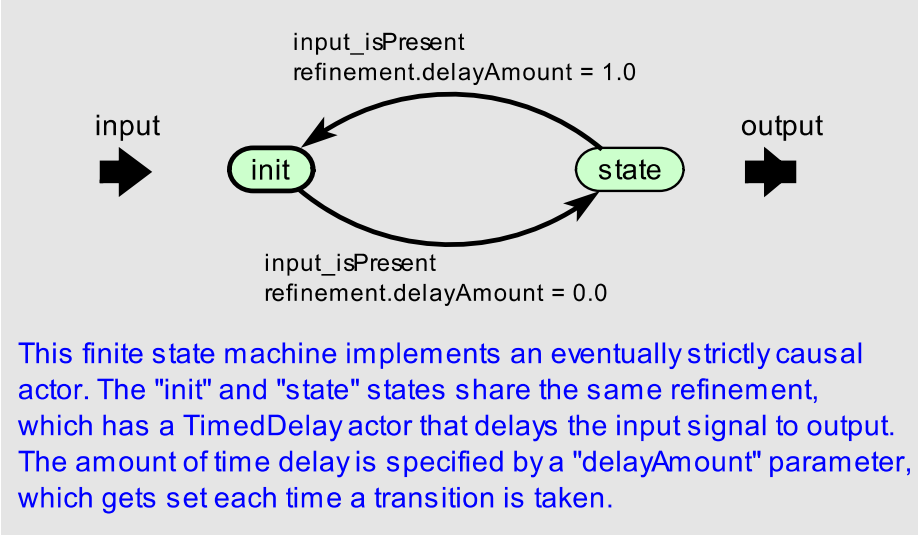

Figure 4.12. The inside details of the `EventuallyStrictlyCausalActor` actor in Figure 4.11.

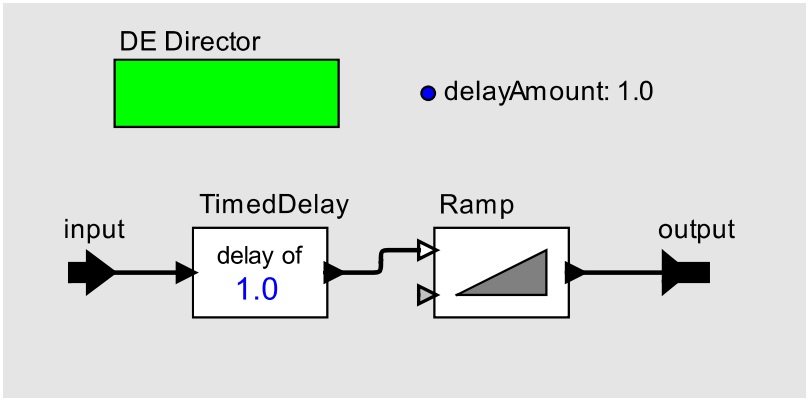

Figure 4.13. The refinement of the `init` and `state` states in Figure 4.12.

Following the iterative procedure given in Theroem (1) and notations given in Section 4.3, we construct the output signal tuple $\mathbf{s} = (s_1, s_2, s_3)$ from the empty signal tuple $\mathbf{s}^0 = (s_\perp, s_\perp, s_\perp)$. The construction process is very similar to the DE model with feedback example in Section 4.3.4.

With $\mathbf{s}^0$ as the input signal tuple, the first firing of the F_a actor generates $\mathbf{s}^1 = (s_1^1, s_2^1, s_3^1)$. The signal s_1^1 is the output signal from the `SingleEvent` source actor and the same as defined in Equation (4.3). The signal s_1^1 is complete and $\forall k \in \mathcal{N}, s_1^k = s_1^1$. The signal s_1^1 has a discrete event at tag $(0, 1)$.

Assume that the `Merge` actor does *simple merge* as defined in Section 3.2.4, then

$$s_2^1 = \texttt{Merge}(s_1^0, s_3^0) = \texttt{Merge}(s_\perp, s_\perp) = s_\perp.$$

The domain of signal s_2^1 does not change and remains as the empty set $\emptyset$.

For the output signal of the `EventuallyStrictlyCausalActor` actor s_3^1, $\text{dom}(s_3^1) = [t_\perp, (1, 0))$ because of the `EventuallyStrictlyCausalActor` is in the `init` mode with a time delay 1.0. The domain of signal s_3^1 is open on the right because the event of signal s_2^0 at tag $(0, 0)$ is unknown. Signal s_3^1 has no discrete events.

With $\mathbf{s}^1$ as the input signal tuple, the 2^{nd} firing of the F_a actor generates $\mathbf{s}^2 = (s_1^2, s_2^2, s_3^2)$. We have $s_1^2 = s_1^1$ as explained before.

Now we have $s_2^2 = \texttt{Merge}(s_1^1, s_3^1)$. Therefore $\text{dom}(s_2^2) = \text{dom}(s_3^1) = [t_\perp, (1, 0))$ as the merge result. The signal s_2^2 now has a discrete event at tag $(0, 1)$.

Finally, because s_2^1 does not change, s_3^2 does not change, i.e., $\text{dom}(s_3^2) = [t_\perp, (1, 0))$ and s_3^2 has no discrete events.

With $\mathbf{s}^2$ as the input signal tuple, the 3^{rd} firing of the F_a actor generates $\mathbf{s}^3 = (s_1^3, s_2^3, s_3^3)$, where $s_1^3 = s_1^1$. $s_2^3 = \texttt{Merge}(s_1^2, s_3^2)$, because $s_3^2 = s_3^1$ and $s_1^2 = s_1^1$, $s_2^3 = s_2^2$.

Since now $\text{dom}(s_2^2) = [t_\perp, (1, 0))$ has a bigger domain, the domain of signal s_3^3 also expands. For tags $t_\perp$ and $(0, 1)$, they are delayed to $(1, 0)$ and $(1, 1)$. Meanwhile, since the signal s_2^2 has one discrete event at tag $(0, 1)$, the output signal s_3^3 now has a discrete event at tag $(1, 1)$, delayed from tag $(0, 1)$. At tag $(0, 1)$, the `EventuallyStrictlyCausalActor` reacts to the input discrete event and changes its state from `init` to `state`, where the amount of delay of the `TimedDelay` actor now becomes 0.0. This state change causes the rest tags in $\text{dom}(s_2^2) = [t_\perp, (1, 0))$ to be delayed by 1 in indexes, resulting the set of tags $[(0, 3), (1, 0))$, where tag $(0, 3)$ is delayed from tag $(0, 2)$. In summary, the domain of output signal s_3^3 according to the definition of the `TimedDelay` actor in

Section 3.2.5,

$$\text{dom}(s_3^3) = [t_\perp, (1,0)) \cup [(0,3), (1,0)) \cup \{(1,0), (1,1)\} = [t_\perp, (1,1)].$$

With $\mathbf{s}^3$ as the input signal tuple, the 4^{th} firing of the F_a actor generates $\mathbf{s}^4 = (s_1^4, s_2^4, s_3^4)$, where $s_1^4 = s_1^1$. $s_2^4 = \texttt{Merge}(s_1^3, s_3^3)$, makes $\text{dom}(s_2^4) = [t_\perp, (1,1)]$ and s_2^4 now contains two events, at tags $(0,1)$ and $(1,1)$. Since s_2^3 does not change, s_3^4 does not change either.

Now we look at the 5^{th} firing of the F_a actor reacting to $\mathbf{s}^4$, which generates $\mathbf{s}^5 = (s_1^5, s_2^5, s_3^5)$. $s_1^5 = s_1^1$. $s_2^5 = s_2^4$ because $s_3^4 = s_3^3$.

Note that the `EventuallyStrictlyCausalActor` is defined as a function in denotational semantics, which is free of side effects. Therefore, all state transitions when the `EventuallyStrictlyCausalActor` reacts an output input signal will be discarded when analyzing the output signal generated from a new input signal. In particular, the initial state always remains `init`.

The `EventuallyStrictlyCausalActor` reacts to the input signal s_2^5 in the following way. Again, for tags $t_\perp$ and $(0,1)$, they are delayed to $(1,0)$ and $(1,1)$. Recall that s_2^5 has two events, at tags $(0,1)$ and $(1,1)$. At tag (0,1), the `EventuallyStrictlyCausalActor` switches state from `init` to `state` and the delay amount changes from 1.0 to 0.0. For the rest of tags in the domain of signal s_2^5, $\text{dom}(()s_2^5) = [t_\perp, (1,1)]$, they are delayed to $[(0,3), (1,2)]$. Note that at tag $(1,1)$, the `EventuallyStrictlyCausalActor` reacts to the second event in s_2^5 by switching state from `state` to `init`. In the end, the output signal s_3^5 has domain $[t_\perp, (1,2)]$ and contains two events at $(1,1)$ and $(1,2)$ respectively.

Keep repeating the above process and it is easy to see that the domains of all signals keeps expanding. When the domains of all signals cover the whole tag set, we find the least fixed point solution.

A portion of the least fixed point solution of signal s_3 (until time 5.0) is shown in Figure 4.14. The dots in the figure indicate the delayed discrete events. The execution results comply with the above analysis results.

The above model illustrates that a feedback composition of actors is live if there is an eventually strictly causal actor in the feedback loop. Note that Theroem (2) is still a sufficient but not necessary condition for answering the liveness problem of compositions of actors.

The above analysis for reasoning about the model behavior is a little bit tedious. One reason

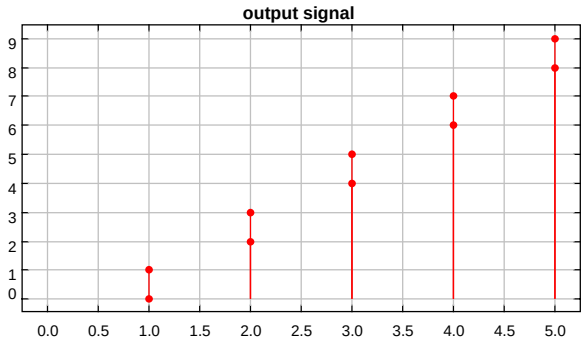

Figure 4.14. The output signal from the `EventuallyStrictlyCausalActor` actor in Figure 4.11.

for this is that our denotational semantics treats actors as function defined over signals. Therefore, we have to make an actor operate on the input signals from the very beginning in order to get the output signals. Note however that a *ModalModel* is intrinsically an FSM, where states are used to summarize the history of transitions. Denotational semantics does not leverage this facility effectively for reasoning model behaviors. When we define our operational semantics in the next chapter, we will provide a solution to avoid this omission, such that the analysis of models becomes more concise.

4.4.2 Causality Interfaces

In this subsection, we will provide a practical way to reason about the causality properties of compositions of actors by developing an interface theory called *causality interfaces* [64, 97].

The major conclusion is that if a composition of actors does not have causality loops, meaning essentially that each loop has a strictly causal actor, then the composition is live. In this subsection, we will develop interfaces for actors to carry causality information, and an algebra to compose them such that compositions of actors acquire causality interfaces that are inferred from the contained actor and their interconnections. By analyzing the causality interfaces and their compositions, we can tell whether a composition has a causality loop, and then can tell whether the composition is live.

A *causality interface* for an actor a with input ports P_i and output ports P_o is a function

$$\delta : P_i \times P_o \to D, \tag{4.6}$$

where D is an ordered set with two binary operations $\otimes$ and $\oplus$ that satisfy the axioms given below.

First, we require that the operators $\oplus$ and $\otimes$ be associative,

$$\forall d_1, d_2, d_3 \in D, \quad (d_1 \oplus d_2) \oplus d_3 = d_1 \oplus (d_2 \oplus d_3),$$
$$\forall d_1, d_2, d_3 \in D, \quad (d_1 \otimes d_2) \otimes d_3 = d_1 \otimes (d_2 \otimes d_3),$$

commutative,

$$\forall d_1, d_2 \in D, \quad d_1 \oplus d_2 = d_2 \oplus d_1,$$
$$\forall d_1, d_2 \in D, \quad d_1 \otimes d_2 = d_2 \otimes d_1,$$

and distributive,

$$\forall d_1, d_2, d_3 \in D, \quad d_1 \otimes (d_2 \oplus d_3) = (d_1 \otimes d_2) \oplus (d_1 \otimes d_3).$$

In addition, we require an additive and multiplicative identity, called $\mathbf{0}$ and $\mathbf{1}$, respectively. That is

$$\begin{aligned}
&\exists\ \mathbf{0} \in D \text{ such that} && \forall\ d \in D, \quad d \oplus \mathbf{0} = d \\
&\exists\ \mathbf{1} \in D \text{ such that} && \forall\ d \in D, \quad d \otimes \mathbf{1} = d \\
&\forall\ d \in D, \quad d \otimes \mathbf{0} = \mathbf{0} \\
&\forall\ d \in D, \quad d \oplus d = d
\end{aligned}$$

The ordering relation $<$ on the set D is a total order, meaning, as usual,

$$\begin{aligned}
&\forall d \in D, && d \not< d \\
&\forall d_1, d_2 \in D, && d_1 \not< d_2 \text{ and } d_2 \not< d_1 \ \Rightarrow\ d_1 = d_2 \\
&\forall d_1, d_2, d_3 \in D, && d_1 < d_2 \text{ and } d_2 < d_3 \ \Rightarrow\ d_1 < d_3.
\end{aligned}$$

As usual, the $=$ relation is ordinary equality, $d \not< d$ is shorthand for an assertion that $d < d$ is false, and $d_1 \leq d_2$ is shorthand for $(d_1 < d_2)$ or $(d_1 = d_2)$.

Finally, the key axioms of D relate the operators and the order as follows.

$$\begin{aligned}
&\forall d_1, d_2 \in D, && d_1 \leq (d_1 \otimes d_2) \\
&\forall d_1, d_2 \in D, && (d_1 \oplus d_2) \leq d_1
\end{aligned}$$

The elements of D are called *dependencies*, and $\delta(p_1, p_2)$ denotes the dependency that port p_2 has on p_1.

We will be interested in two specific cases. In the *boolean dependency* case, $D = \{\text{true}, \text{false}\}$, $\oplus$ is logical and, $\otimes$ is logical or, false $<$ true, $\mathbf{0}$ = true, and $\mathbf{1}$ = false. With these definitions, all of the above axioms are satisfied.

In the *weighted dependency* case, $D = \mathbb{R}_+ \cup \{\infty\}$, the non-negative real numbers plus infinity, $\oplus$ is the min function, $\otimes$ is addition, $<$ is ordinary numerical ordering, $\mathbf{0} = \infty$, and $\mathbf{1} = 0$[3]. Again, with these definitions, all of the above axioms are satisfied.

Composition Analysis

Given a set A of actors, a set C of connectors, and the causality interfaces for the actors, we can determine the causality interface of the composition. Consider the example in figure 4.1. To determine the causality interface of the composite actor a, we need to determine the function

$$\delta_a \colon \{p1, p2, p3\} \times \{p4, p5, p6\} \to D.$$

To do this, we form a graph of ports, and observe that the paths between ports traverse both actors and connectors. To determine the value of $\delta_a(p1, p4)$, for example, we need to consider all the paths between $p1$ and $p4$. A path consists of links provided by connectors and actors. The dependencies of the links and actors are combined using the $\otimes$ operator for series compositions and the $\oplus$ operator for parallel compositions.

Consider the example in Figure 4.1. From Figure 4.1(b) we can immediately conclude that

$$\delta_a(p1, p5) = \delta_1(p1, p5),$$

where δ_1 is the causality interface of actor a_1. Where there are no dependencies, the causality interface yields the additive identity, so

$$\delta_a(p1, p4) = \mathbf{0}.$$

Suppose for example that Figure 4.1 represents a synchronous program in, say, Lustre. For synchronous programs, we use boolean dependencies, where $D = \{\text{true}, \text{false}\}$. For ports p and p', $\delta(p, p') = \text{false}$ is interpreted to mean that at a tick of the clock, the value at p' depends on the value at p. A dependency value true is interpreted to mean that there is no such dependency. Suppose we annotate the diagram as shown in figure 4.15, where a solid line denotes a dependency with value false, a dashed line denotes a dependency with value true, and no line denotes a dependency with value $\mathbf{0} = \text{true}$. Hence, for instance, the dashed line between $p2$ and $p4$ might mean that there is a "pre" operator, which decouples the value at $p4$ from that at $p2$ in each tick of the clock. The dashed line, therefore, is equivalent to no line. Then the causality of the interface of the composite actor a becomes as shown in 4.15(c).

[3]In this case, the dependency set is also called a min-plus algebra [12].

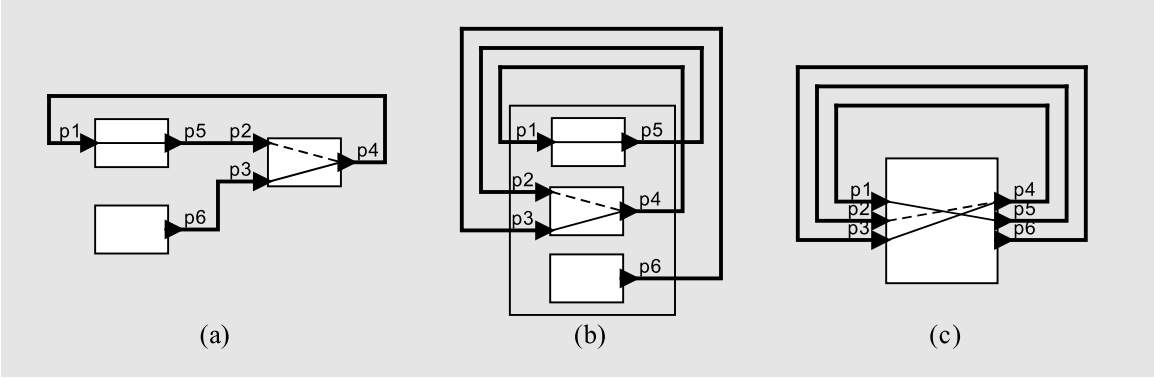

Figure 4.15. The composition of Figure 4.1 annotated with boolean dependencies. If the dependencies in (a) are given, the dependencies in (c) can be systematically inferred.

Connectors have particularly simple causality interfaces. A connector $c \in C$ linking output port p and input port p' yield the dependency

$$\delta(p, p') = \mathbf{1},$$

the multiplicative identity.

We can now analyze the feedback composition for causality loops. A causality loop exists if the dependency between any port and itself is false. Consider for example port $p4$. Using the fact that connectors yield the multiplicative identity for dependencies, we can write this as

$$\begin{aligned} \delta(p4, p4) &= \delta_1(p1, p5) \otimes \delta_2(p2, p4) \\ &= \text{false} \otimes \text{true} \\ &= \text{true} \end{aligned}$$

where we have used the fact that in boolean dependencies, $\otimes$ is logical or. We can similarly check every port to determine that this composition has no causality loops.

Consider the same diagram, but now representing a discrete-event model. For discrete-event models, we use a weighted dependency model, where $D = \mathbb{R}_+ \cup \{\infty\}$, the non-negative real numbers plus infinity, $\oplus$ is the min function, $\otimes$ is addition, $<$ is ordinary numerical ordering, $\mathbf{0} = \infty$, and $\mathbf{1} = 0$. Each dependency represents a time delay. In Figure 4.15, we now interpret solid lines to represent a delay of zero, so for example

$$\delta_1(p1, p5) = 0.$$

A causality loop occurs if the dependency from any port to itself is zero. A similar analysis again yields the fact that the model in Figure 4.15 has no causality loops.

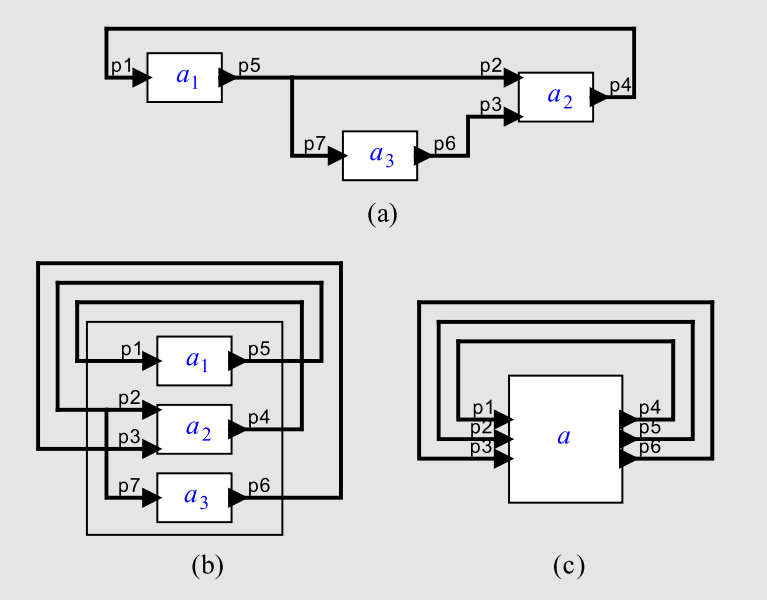

Figure 4.16. A more complicated composition.

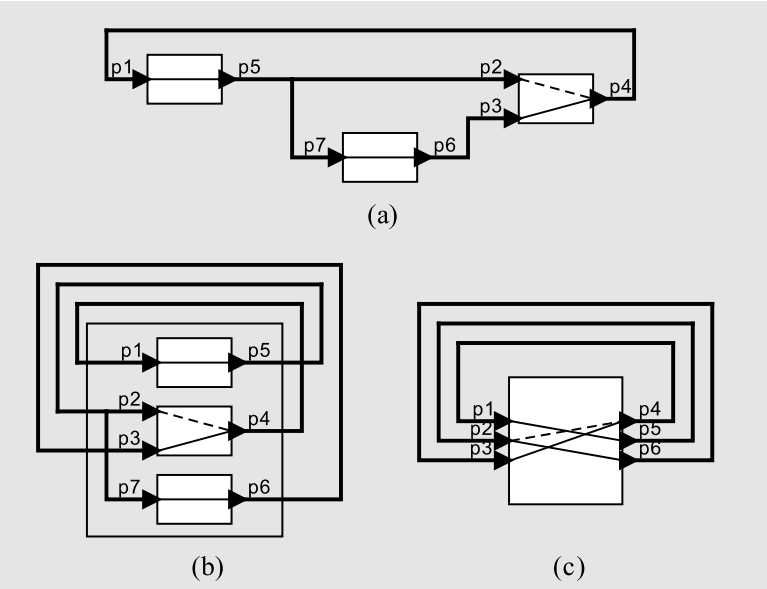

Figure 4.17. Dependencies for the composition in Figure 4.16. If the dependencies in (a) are given, the dependencies in (c) can be systematically inferred.

Consider the slightly more complicated composition shown in figure 4.16. This example has a more complicated link, joining ports $p5$, $p2$, and $p7$. It also has parallel paths. Suppose that the dependencies are as shown in Figure 4.17. Then we can perform the composition analysis to determine that port $p4$ has a causality loop. In particular, assuming a synchronous model,

$$\begin{aligned}\delta(p4, p4) &= \delta_1(p1, p5) \otimes (\delta_2(p2, p4) \oplus (\delta_3(p7, p6) \otimes \delta_2(p3, p4))) \\ &= \text{false} \otimes (\text{true} \oplus (\text{false} \otimes \text{false})) \\ &= \text{false}.\end{aligned}$$

Just as the $\otimes$ operator is used to compose causality of chained links, the $\oplus$ operator is used to compose causality of parallel links. The same analysis would reveal a causality loop in a discrete-event model.

Dynamic Causality Interfaces

In the above examples, the dependencies are static (they do not change during execution of the program). This situation is excessively restrictive in practice. For example, they do not describe well the actors that are eventually strictly causal. As presented in Section 3.3.2, one simple way to model dynamically changing dependencies is to use *modal models*. In a modal model, an actor is associated with a state machine, and its interface can depend on the state of the state machine. Therefore, an actor could have a different causality interface in each state of the state machine. In particular, let X denote the set of states of the state machine. Then the causality interfaces are given by a function

$$\delta' \colon P_i \times P_o \times X \to D.$$

A simple conservative analysis would combine the causality interfaces in all the states to get a conservative causality for the actor. Specifically, for an input port $p_i \in P_i$ and an output port $p_o \in P_o$,

$$\delta(p_i, p_o) = \bigoplus_{x \in X} \delta'(p_i, p_o, x).$$

This is conservative because causality analysis based on this interface may reveal a causality loop that is illusory, for example if the state in which the causality loop occurs is not reachable.

Depending on the model of computation and the semantics of modal models, the reachability of states in the state machine may be undecidable [42]. Hence, a more precise analysis may not always be possible. Nonetheless, it is easy to imagine circumstances in which a precise analysis could be carried out.

Dynamic analysis of causality interfaces at runtime offers more precise analysis results. This analysis technique essentially analyzes only those causality interfaces of the *active states* and their compositions, therefore offers more precise analysis results than conservative static analysis. This technique avoids the necessity of iterating all possible compositions of all states and therefore is computationally more practical. The limitation of the dynamic technique is that the analysis results may not be as comprehensive as those from extensive and thorough static analysis.

Figure 4.18 shows a model that will be treated as an algebraic loop if conservative analysis is performed, while in fact it is a live model as dynamic causality analysis shows that the causality loop is illusory.

This model contains two modal models `A` and `B`, each of which contains the same set of two states, `delay` and `noDelay`. For the `delay` state, there is a fixed amount of time delay, 1.0 time unit, from the input signal to the output signal; while the `noDelay` state simply connects the input signal to output signal.

The modal models `A` and `B` only differ in two places. The first difference is the initial state, modal model `A` has `delay` as the initial state, while modal model `B` has `noDelay` as the initial state. The second difference is the transients between the `delay` and `noDelay` states. These two modal models have exactly opposite settings.

The execution result is shown in Figure 4.23. The `SingleEvent` actor in Figure 4.18 outputs a single event at tag $(0, 1)$ with a value 5.0. The `Clock` actor outputs 1 and 0 alternatively at tags $(0.0 + 2k, 1)$ and $(1.0 + 2k, 1)$, $k \in \mathcal{N}$ respectively, which triggers the modal models to change states. With such configurations, the model shown in Figure 4.18 as a whole will have a non-zero time delay, 1.0, in any feedback loop at any time. In fact, the whole model just behaves like the model shown in Figure 3.20 but with the `TimedDelay` actor configured to have a time delay of 1.0. Therefore, the model is live according to Theroem (2). The execution results are shown in Figure 4.23.

Determining Causality Interfaces for Atomic Actors

The causality analysis technique we have given determines the causality interface of a composition based on causality interfaces of the components and their interconnections. An interesting question arises: how do we determine the causality interfaces of atomic actors? If the atomic actors are language primitives, as in the synchronous languages, then the causality interfaces of the

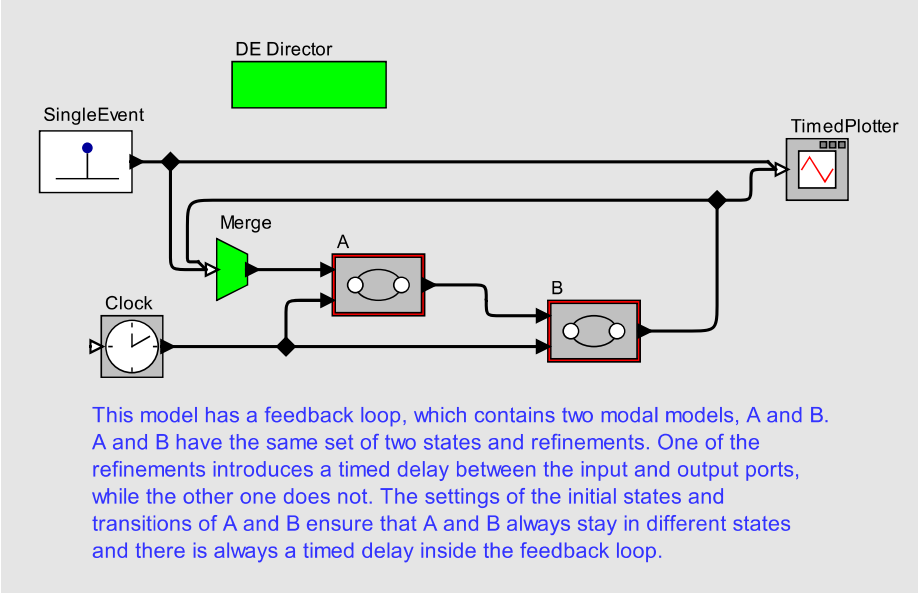

Figure 4.18. A live model contains two modal models, where both of them have dynamic causality interfaces.

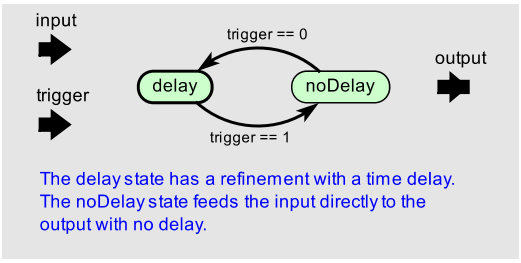

Figure 4.19. The details of the modal model `A` in figure Figure 4.18.

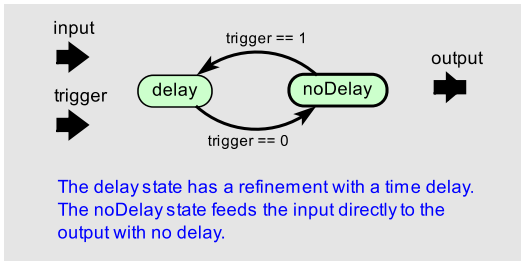

Figure 4.20. The details of the modal model `B` in figure Figure 4.18.

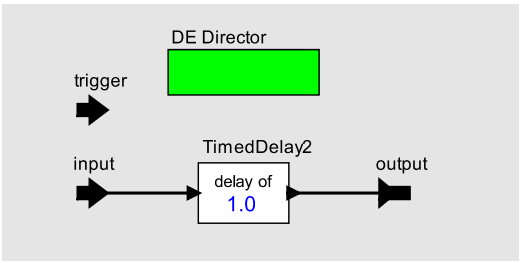

Figure 4.21. The refinement of the `delay` state.

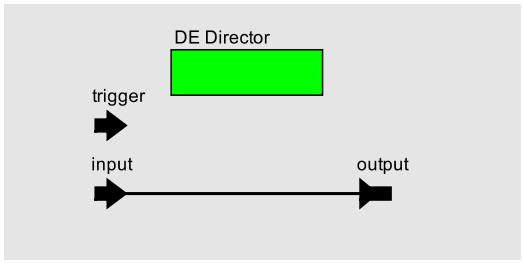

Figure 4.22. The refinement of the `noDelay` state.

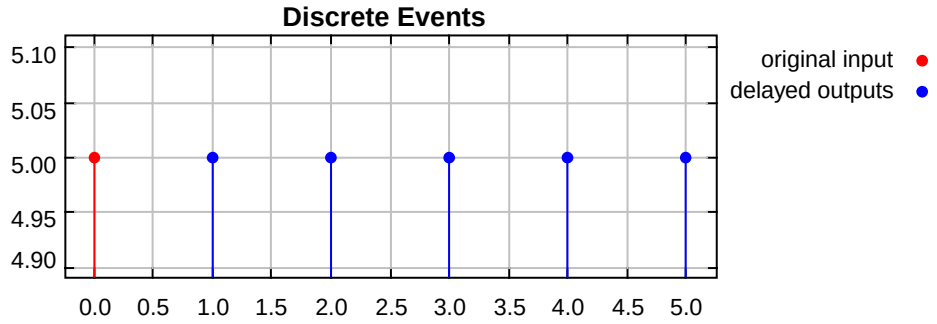

Figure 4.23. Concatenation of k identical function a results a single function a^k.

primitives are simply part of the language definition. They would be enumerated for use by a compiler.

However, in the case of coordination languages, the causality interfaces might be difficult to infer. If the atomic actors are defined in a conventional imperative language, then standard compiler techniques such as program dependence graphs (see for example [38, 52, 78]) might be usable. However, given the Turing completeness of such languages, such analysis is likely to have to be conservative.

Chapter 5

Operational Semantics

In this chapter, we give an operational semantics for hybrid systems. In computer science, operational semantics is a mathematically rigorous way to give meaning to computer programs by describing how to interpret a valid program as sequences of computational steps. These sequences then give the meaning of the program. In this dissertation, a program is a model with hybrid system semantics, whose meaning is given by the signals that reflect the interactions of actors in the model. Therefore, an operational semantics in our context is to define the rules for constructing signals.

5.1 Issues of Denotational Semantics

From the examples of the previous section, it is clear that the constructive procedure given in Theroem (1) focuses on the construction of signal domains. To be specific, the construction procedure relies on the causality properties of actors, specially those strictly causal actors, to make time diverge such that signal domains expand.

However, this denotational semantics cannot be directly used as an operational semantics for three reasons. First, this procedure treats actors as "functions" that transform a set of signals to another set of signals, which are also "functions." This makes it intuitive and formal to reason certain properties but not very convenient for software implementation, because software does not operate on abstract functions but instead on concrete data structures. Second, this procedure does not explicitly provide quantity information of how to advance time and expand signal domains. Lastly, but more importantly, the procedure does not tell how to compute signal values. We will suggest

solutions for all these issues in this section and formally define an operational semantics in Section 5.2.

5.1.1 Signal Domain Expansion

Recall that the challenging problem is that a computer execution of a hybrid system model is constrained to provide signal values on a discrete set. Therefore, one key question of our operational semantics is how to construct a discrete representation of signals.

The first step to construct a discrete representation for a signal is to determine the discrete subset where the discrete representation is defined. Recall that a discrete subset is a set of tags, which is a subset of the domain of the signal. The question of how to choose the discrete subset is equivalent to the question of how to control the intervals between consecutive tags in the discrete subset, i.e., how to control the signal domain expansion.

The denotational semantics in the previous chapter gives a constructive way to expand signal domains. In this subsection, we give an operational semantics that expands, instead of signal domains, the domains of discrete representations of signals in a constructive and more quantitative way. This domain expansion mechanism refines what is given in the denotational semantics. In this section, without ambiguity, we use "domain expansion" and "expansion of discrete subset" interchangeably.

An important observation from the examples in the previous chapter is that it is the *source actors* and the *eventually strictly causal actors* that initiate and continue domain expansion. Next, we discuss in detail how these actors expand signal domains in our operation semantics.

A source actor does not have any inputs, therefore its output signals are always complete. The discrete representation of the output signals, in theory, can be fully determined for any chosen discrete subset.

Take the `SingleEvent` actor in Figure 4.8 as an example. It produces the signal defined in Equation (4.3), which contains only one discrete event at tag $(0, 1)$. As we mentioned before in Section 2.4, the *minimum* discrete representation of a DE signal already captures all its dynamics. Therefore, our operational semantics only needs to construct the minimum discrete representation for the output signal of the `SingleEvent` actor, which contains three events at the following three tags, $(0, 0), (0, 1), (0, 2)$.

However, our operational semantics usually does not construct the whole discrete representation of the output signals from source actors at the very beginning of execution. In fact, in general, the construction is impractical for two reasons. First, the discrete subset of an output signal may be infinite. For example, the `DiscreteClock` signal defined in Example (7) contains an infinite number of discrete events. Its discrete subset is infinite.

Second, as introduced in Section 2.5.1, the discrete subset of a signal tuple depends on the discrete subsets of all component signals, and not all of them are known before execution. This reason particularly applies to CT signals. Recall that although the minimum discrete representation is sufficient for DE signals, a CT signal usually needs a practical discrete representation that has a bigger discrete subset and includes more events than the minimum discrete representation. As we explained in Section 2.4.2, which extra tags are included in the discrete subset depends on the dynamic equations and actual ODE solvers used to solve them. To make this explanation more concrete, we compare the output signals in Figure 4.3 and Figure 4.7. Both output signals are CCT signals, therefore, the dots in the figures are actually the discrete representations of the output signals. The tags where these dots happen are the discrete subset of the discrete representations and shared by all other signals in the same model including the output signal from the `Const` actor. It is obvious that the output signal of `Const` actor has two very distinct discrete subsets in two models with different dynamics.

Incremental and Lazy Domain Expansion

Now we explain how our operational semantics *incrementally* construct the discrete subset for output signals of source actors. We have two choices to expand the signal domain. The first choice is to expand the domain as far as possible, called *eager expansion* (or *optimistic* expansion). The second one is to choose the minimum expansion, called *lazy* expansion (or *conservative* expansion).

The denotational semantics introduced in the previous chapter embraces eager construction, which makes the analysis more compact. However, for operational semantics, we favor the lazy construction. Not only because it is impractical sometimes to deploy eager construction, such as for the aforementioned `DiscreteClock` signal, but also it is more efficient from the implementation point view, such as to minimize the memory usage for storing events and the overhead that arises from backtracking[1].

[1]Here we are not concerning distributed or parallel simulation strategies, which may have some other different arguments on whether an implementation is more efficient. Nevertheless, the concerns we discussed here are valid for both distributed and parallel simulation strategies.

Take the `SingleEvent` actor again as our example. Suppose our operational semantics starts at tag $t_\perp = (0, 0)$, the lazy expansion mechanism expands the discrete subset of the output signal incrementally, $\emptyset \rightarrow \{(0, 0)\}$, then $\{(0, 0)\} \rightarrow \{(0, 0), (0, 1)\}$, and then $\{(0, 0), (0, 1)\} \rightarrow \{(0, 0), (0, 1), (0, 2)\}$.

With the lazy construction mechanism, the amount of domain expansion is the interval between *consecutive tags* in the discrete subset. We call this domain expansion a **unit expansion**. The corresponding execution of the operational semantics to solve a subset of the discrete representation on this interval is called a **unit execution**.

After the final tag (defined at the end of Section 2.2.1) is covered inside the discrete subset, the operational semantics simply stops executing. This gives a **stop condition for execution**.

Another stop condition is that if no signal has more interesting events to produce, then there is no need to keep the execution running. A simple example that illustrates the stop condition is a DE model that just contains the `SingleEvent` actor example mentioned above. After tag $(0, 2)$, the output signal of the `SingleEvent` actor contains nothing interesting but absence events. If we constrain our operational semantics to explicitly represent discrete events only, then there is nothing interesting left for the operational semantics to do. Thus, the execution stops.

Coordination and Contract

So far, we have explained how operational semantics lazily expands domains incrementally given a discrete subset. However, we have not explained where this discrete subset comes from. We do this next.

It is important to note that our operational semantics only *coordinates* the interactions among actors. It is not the operational semantics but an actor that is responsible to report a domain expansion. This is because an actor defines the interactions of signals and has a better knowledge when an interesting event will happen. For example, the `DiscreteCock` actor knows discrete events will be generated at tags $\{(kp, 1) \mid k \in \mathcal{N}\}$ in its output signal; the `Integrator` actor knows how to approximate the truncation errors and adjust the step size accordingly.

As a coordinator, the operational semantics makes a *contract* with the actors under its coordination: Each actor faithfully reports the expected expansion of the discrete subset for its own output signals; The operational semantics controls domain expansion by honoring the smallest domain expansion request.

How much expansion to report, i.e., extending the discrete subset to which tag, is a local decision of each individual actor based on the best knowledge the actor has. Like the operational semantics, an actor is also conservative to expand the discrete subset of its output signals. In particular, it only requests the discrete subset to be extended to the tag of the next earliest event.

An eventually strictly causal actor also expands the domain of its output signals. Unlike source actors, the output signals are usually not complete. However, due to the lazy domain expansion mechanism chosen by the operational semantics, an eventually strictly causal actor is treated as exactly the same as a source actor. Here is the reason. Suppose an eventually strictly causal actor extends the domain of its output signal to $[t_\perp, t']$. Following the contract between the operational semantics and the actors, the final tag of the expanded domain picked by the operational semantics, denoted as t'', must satisfy $t_\perp < t'' \leq t'$. Note that the output signal s is completely defined during the interval $[t_\perp, t'']$, therefore the eventually strictly causal actor can be treated exactly like a source actor.

At the new tag where the discrete subset is extended, after the discrete representation of all signals are resolved (this will be discussed in detail later), the operational semantics awaits the expansion requests again and expands the signal domains further. This process will keep repeating until the final tag of the tag set is reached. This whole process illustrates how to construct the discrete subset of signals. The results comply with the definition of the discrete subset for signal tuples in Definition (30).

A complete procedure of how the operational semantics constructs the discrete subset for discrete of representations of signals is described as follows.

Procedure 3 *Discrete Subset Construction Procedure:*

Let $\boldsymbol{s}$ *be the tuple of all signals involved and* $\mathcal{T} = \mathrm{dom}(\boldsymbol{s})$. *Initially, the discrete subset is an empty set,* $\mathcal{D}_{\boldsymbol{s}} = \emptyset$.

1. *All source actors and eventually strictly causal actors request domain expansion by telling the final tags of their expected domains after expansion. Let* $T = \{t_i \mid i \in \{1, 2, \cdots, n\}\}$ *be the set of all these final tags.*

2. *Let* $t = \boldsymbol{min}(T)$ *be the smallest tag. The discrete subset expands to*

$$\mathcal{D}_{\boldsymbol{s}} = \mathcal{D}_{\boldsymbol{s}} \cup \{t\}.$$

3. *Resolve the discrete representation* $DiscreteRepresentations_{\boldsymbol{s}}$.

4. If tag t is not the final tag of $\mathrm{dom}(\boldsymbol{s})$*, go to 1 and repeat. Otherwise, stop.*

5.1.2 States

In the previous subsection, we resolved the issue that software does not operate on signals that are abstract functions. The solution we give is to compute discrete representation of signals instead. We give a procedure of how to constructively compute the discrete subsets. However, it is still cumbersome to utilize the discrete representation directly because a discrete representation carries too much information.

Let us revisit the CT model with feedback shown in Figure 4.6 to make this argument more concrete. If we elaborate on the 5^{th} firing of F_a a little bit, we will find that the whole signal s_2^4, which is defined on $\mathcal{T}'$, has to be fed into the `Integrator` for computing s_3^5 as defined in Section 3.2.3. Let $\mathcal{T}'' = \mathrm{dom}(s_3^5)$, then we have $\mathcal{T}' \subset \mathcal{T}''$. Note that this integration process starts from $t_\perp$ irrespective of the fact that a prefix of the signal s_3^5, specifically, s_3^4, has been computed. This is true even if we constrain the `Integrator` actor to compute discrete representation of signals only.

This contrasts to what an integration process is normally implemented in practice, where intermediate integration results may be used for future integration. Formally, to resolve an integration a function $\dot{x}(t)$ from 0 to b, if we have the integration result from 0 to a, where $a \leq b$, then

$$\begin{aligned} x(b) &= x(0) + \int_0^b \dot{x}(\tau, 0) d\tau \\ &= x(a) + \int_a^b \dot{x}(\tau, 0) d\tau. \end{aligned}$$

The key difference between the practical implementation and the one derived from the denotational semantics Theroem (1) is that the practical implementation introduces *states*, such as $x(a)$, which summarize the total effect of the history inputs on the initial state.

Next I will show how to introduce states into our operational semantics and use them for solving discrete representations of signals.

Procedure 4 *Discrete Subset Modification Procedure:*

Let $\boldsymbol{s}$ *be the tuple of all signals involved and* $\mathcal{T} = \mathrm{dom}(\boldsymbol{s})$*. Initially the discrete subset as an empty set,* $\mathcal{D}_{\boldsymbol{s}} = \emptyset$*.*

1. Following the steps in Procedure (3), extend the discrete subset to

$$\mathcal{D}_{\boldsymbol{s}} = \mathcal{D}_{\boldsymbol{s}} \cup \{t\}.$$

2. *After the discrete representation $\mathcal{DS}_s$ defined on $\mathcal{D}_s$ are resolved, save the states of actors.*

3. *Restrict the signal domain to a new set: $\mathcal{T}' = \mathcal{T} \setminus [t_\perp, t)$, where $t_\perp = \bigwedge \mathcal{T}$.*

4. *If tag t is the final tag of* $\mathrm{dom}(\boldsymbol{s})$*, stop. Otherwise, go to 3 and continue.*

One side effect resulted from step 3 of Procedure (4) is that the discrete subset is also modified such that all tags smaller than $t_\perp$ are discarded. Formally, $\mathcal{D}'_{\mathbf{s}} = \mathcal{D}_{\mathbf{s}} \cap \mathcal{T}'$. This effectively makes the discrete subset contain only one new element, the new tag t generated from Procedure (3) in step 1.

Consequently, the discrete representation of signals defined on those tags also disappear. We need a way to remember those events. Similarly to the intermediate states used for integration, we require some actors to have *states*, where the states summarize the effects on the *initial* states from the historical inputs until but not including the current tag. The states are saved in step 2. The saved states, together with the inputs at the current tag, will determine the future discrete representation and states.

We associate each actor a state. The state of a strict functional actor in Section 3.2 does not involve into the computation of the actor. Therefore is not important and can be anything.

The state of an `Integrator` summarizes the effects of the derivative input signal and the impulse and reset DE input signals. Equation (3.2) defines how the state is computed from the initial state and the input signals. Necessary integration-related information is also stored in the states.

The state of the `TimedDelay` actor, on the other hand, includes *future* discrete events. This is due to the lazy domain expansion of operational semantics. Suppose we have two `TimedDelay` actors a and b in a model. At tag t, based on the discrete representation of the input signals (and the states of actors), each actor generates a future event at tags t_a and t_b respectively. These two actors then request domain expansion: a requests to expand domain to tag t_a and b requests to tag t_b, where $t_a < t_b$. The operational semantics only grants the request from a and expands the discrete subset to t_a following Procedure (3), and then discards the prefix of discrete representation before tag t_a for all signals following Procedure (4). The actor b that has to remember that there is a future event at t_b, because after the operational semantics discards the prefix of discrete representation of input signals to actor b, there is no way for actor b to recompute the event happening at t_b again.

Similarly, the `DESource` actors, such as `SingleEvent` actor and the `DiscreteClock` actor all maintain states, which are future events. On the other hand, sink actors, such as the `TimedPlotter` actor, store all the *past* inputs have received.

The state of a `ModalModel` actor has two parts: one comes from the finite state machine (FSM), which models the state transitions, the other comes from the states of the refinements. The states of the FSM summarize the history of state transitions caused by input signals. The states of refinement are the states of the actors contained by the refinement, such as the `Integrator` or the `TimedDelay` actor or a mixture of these two.

5.1.3 Solving Signal Values

In this subsection, we solve the third issue of the denotational semantics by defining how to solve the signal values on the discrete subset.

Our solution leverages the lazy incremental domain expansion and states for actors introduced in the previous two subsections. In particular, given a model, at each new tag chosen for extending the discrete subset, the operational semantics keeps iterating all actors until all output signal values do not change. The signal values are computed from the states of actors. The set of resolved values is called a fixed point and defines the model's behavior at the new tag. After the signal values are resolved, the operational semantics chooses another tag to extend the discrete subset. By repeating the above process, when the discrete subset covers the whole tag set where the signals are defined, the operational semantics constructs the complete discrete representation of all signals.

Our solution is heavily inspired by the fixed-point semantics of synchronous languages [16, 17], particularly Lustre [47], Esterel [18], Signal [45], and Synchronous/Reactive (SR) language [36]. Next, we first introduce the mathematics background of this fixed point semantics that guarantees the success of the above solution. Then we will come back to give a more precise procedure of how to solve signal values.

The fixed-point semantics of synchronous languages also has roots in the least fixed point theorem given in Theroem (1). In our previous efforts on applying this theorem to reason about the uniqueness and existence of behaviors of compositions, all signals form a CPO under a *prefix order* relation. For synchronous languages, all **signal values** together with the `Unknown` value, denoted as $\mathcal{V}_\perp = \mathcal{V}_\varepsilon \cup \{\perp\}$ (see Section 2.1.2), form a CPO under the *information* order $\preceq$ (see Definition (4)).

We associate actors with functions defined on the value set $\mathcal{V}_\perp$. As will be seen later, these functions are called *output* functions. These output functions are closely related to the previous definitions of actors given in Section 3.2. We will further explain this later in Section 5.2.1. Next

we define *monotonicity* and *continuity* properties for these output functions defined on the value set with the information order relation.

Definition 36 (Monotonicity) A function $f\colon \mathcal{V}_\perp^n \rightharpoonup \mathcal{V}_\perp^m$ is **monotonic** if it is order-preserving,

$$\forall\, \mathbf{r}, \mathbf{s} \in \mathcal{V}_\perp^n,\ \mathbf{r} \preceq \mathbf{s} \implies f(\mathbf{r}) \sqsubseteq f(\mathbf{s}).$$

Definition 37 (Scott Continuity) A function $f\colon \mathcal{V}_\perp^n \rightharpoonup \mathcal{V}_\perp^m$ is **(Scott) continuous** if for any directed set $D \subseteq \mathcal{V}_\perp^n$, $f(D) \subseteq \mathcal{V}_\perp^m$ is a directed set, and

$$f(\bigvee D) = \bigvee f(D).$$

Recall that the value set $\mathcal{V}_\perp$ is a flat CPO, the longest chain has a length of 2. The longest chain in $\mathcal{V}_\perp^n$ has a length $n+1$. That is, all chains in $\mathcal{V}_\perp^n$ are *finite*. This special structure gives us very convenient properties for reasoning the functions defined on $\mathcal{V}_\perp$.

The first property is that a monotonic function defined on $\mathcal{V}_\perp$ is also a continuous function, besides the fact that continuity implies monotonicity [31].

The second property is that the constructive procedure for finding the least fixed point solution always terminates in a finite number of computation steps. In particular, given a function $f\colon \mathcal{V}_\perp^n \rightharpoonup \mathcal{V}_\perp^n$, the least fixed point solution of equation $f(\mathbf{s}) = \mathbf{s}$ is $f^n(\perp)$, where $\perp$ is the bottom element of $\mathcal{V}_\perp^n$. This property indicates that the least fixed point solution can always be found by consecutively evaluating the function f for a finite number of times, and the number of evaluations is the same as the size of value tuple. We extend this property to composition of actors and get the following conclusion.

Given a model consists of n (monotonic) actors and m signals. The upper bound of the execution cost to achieve the least fixed point solution is $n \times m$, where the unit is the normalized cost of one execution of an actor. Here one execution of an actor corresponds to one evaluation of the output function associated with the actor.

If an execution of an actor does not change any of its output signal values, then consecutive executions of the same actor won't change the outputs. Therefore, if an execution of all actors in sequence does not change any of the signal values, then consecutive executions of any actor or actors won't change any signal values either. This essentially means a fixed point has been reached. The worst case scenario is that each execution of all actors in sequence changes only one signal value and then it takes m executions to solve all signal values in order to reach a fixed point. Thus, the worst cost is $n \times m$.

With causality interface analysis, we can reduce the cost to $n \times c$, where $1 \leq c \leq m$. This is because causality interface analysis provides data dependency information among actors. Therefore we can adjust the execution order of actors, so that the outputs from one actor can be used by the other actors that need them in the same execution. The adjustment of execution order of actors allows more than one signal value to be resolved in one execution. Thus, the total number of executions required to reach a fixed point is reduced. However, there is a lower bound for the execution cost, which corresponds to the case $c = 1$, in that each actor has to been executed at least once to change their output signal values.

As a related work, in [36], Edwards gave some exact and heuristic algorithms for finding efficient schedules to compute the least fixed points. His algorithms are based on graph transformation and the Bekic's theorem [14].

Before we give the procedure of how to compute signal values, we need one more definition for the output functions of actors defined on the value set $\mathcal{V}_\perp$.

Definition 38 (Strictness) A function $f : \mathcal{V}_\perp \rightharpoonup \mathcal{V}_\perp$ is **strict** if

$$f(\perp) = \perp.$$

A function which is not strict is called **non-strict**.

A strict function is also monotonic but a monotonic function is not necessarily a strict function.

If the output function of an actor is strict (non-strict), the actor is called strict (non-strict). Intuitively, a strict actor does not produce any useful information unless all inputs are resolved to be known values[2]. While a non-strict actor can produce some `known` output value even part of its input values remain `unknown`.

Of all the actors introduced in Section 3.2, source actors, `Integrator`, the `TimedDelay` actor, and the `ZeroOrderHold` actor are non-strict actors. A `ModalModel` is non-strict if its active state is a transient state or the refinements of its active state contain at least one non-strict actor.

The importance of the strictness property for solving signal values is similar to that of the causality property for expanding signal domains. Both of them are used to break feedback loops, as illustrated in the following procedure.

Procedure 5 *Procedure For Solving Signal Values :*

[2]For simplicity, this definition does not consider *jointly strict* actors, such as actors implementing logic functions, which take more than one value as inputs. The strictness of these actors may be defined on individual input values.

Let $\boldsymbol{s}$ be the tuple of all signals involved and $\mathcal{T} = \mathrm{dom}(\boldsymbol{s})$.

1. *Let $t = \bigvee \mathcal{D}_{\boldsymbol{s}}$. At tag t, reset all signal values to $\perp$, i.e.,*

$$\boldsymbol{s}(t) = (\underbrace{\perp, \perp, \cdots, \perp}_{n}).$$

2. *Execute all non-strict actors.*

3. *Execute the rest of the actors according to a schedule to resolve the signal values $\boldsymbol{s}(t)$.*

4. *If the newly resolved signal values $\boldsymbol{s}(t)$ are the same as the results from last execution, a fixed point is reached and the procedure terminates. Otherwise, go to 3 and repeat.*

The step 2 in Procedure (5) is pulled out from the iteration loop from step 3 to step 4, because the outputs of non-strict actors do not depend on their inputs and are completely determined for the first execution. Therefore, to keep executing them in the iteration loop adds unnecessary computation cost and is wasteful.

In step 3, a *schedule* is used to compute the output signal values. The choice of schedule does not affect the fixed point results, as Theroem (1) tells us that the continuous function describing the model has a unique least fixed point solution. However, the choice of schedule does affect the computation cost for finding the fixed point.

5.2 Operational Semantics

In this section, we will formally define the operational semantics for hybrid systems. This operational semantics will be used as the guideline for us to implementing software tools for modeling and simulating hybrid systems.

We first summarize the procedures introduced in the previous section by aggregating them together. The result procedure gives us a big picture of what the operational semantics looks like.

Procedure 6 *Operational Semantics: (informal description)*

Let $\boldsymbol{s}$ be the tuple of all signals involved and $\mathcal{T} = \mathrm{dom}(\boldsymbol{s})$.

1. *Start with the discrete subset as an empty set, $\mathcal{D}_{\boldsymbol{s}} = \emptyset$. Initialize the states of actors.*

2. *All source actors and eventually strictly causal actors request domain expansion by telling the final tags of their expected domains after expansion. Let* $T = \{t_i \mid i \in \{1, 2, \cdots, n\}\}$ *be the set of all these final tags.*

3. *Let* $t = \textbf{\textit{min}}(T)$ *be the smallest tag. The discrete subset expands to*

$$\mathcal{D}_{\boldsymbol{s}} = \mathcal{D}_{\boldsymbol{s}} \cup \{t\}.$$

4. *At tag* t*, reset all signal values to* $\perp$*, i.e.,*

$$\boldsymbol{s}(t) = (\underbrace{\perp, \perp, \cdots, \perp}_{n}).$$

5. *Execute all non-strict actors to resolve the signal values* $\boldsymbol{s}(t)$ *based on their states and partial inputs.*

6. *Execute the rest of the actors according to a schedule to resolve the signal values* $\boldsymbol{s}(t)$*. The signal values are computed from the inputs and the actor states.*

7. *If the newly resolved signal values* $\boldsymbol{s}(t)$ *are the same as the results from last execution, a fixed point is reached. Otherwise, go to 6 and repeat.*

8. *Update the states of actors based on the fixed point.*

9. *Restrict the signal domain to a new set:* $\mathcal{T}' = \mathcal{T} \setminus [t_\perp, t)$*, where* $t_\perp = \bigwedge \mathcal{T}$*.*

10. *If tag* t *is the final tag of* $\mathrm{dom}(\boldsymbol{s})$*, stop. Otherwise, go to 2 and continue.*

5.2.1 Abstract Actor Semantics

In this subsection, we formalize the previously mentioned solutions for the issues of denotational semantics by associating actors with an **abstract actor semantics**. This semantics is called abstract in that it omits details of execution that are not relevant, such as how the actor actually performs what computation. In particular, this abstract actor semantics consists of three key functions: *output* function, *update* function, and *request* function. We use f, g, and h to denote these functions respectively. The output function f defines the operation of reading inputs and computing outputs based on the inputs and states. The update function g defines the operation of updating the states based on the current inputs and states. The request function h defines the operation of requesting domain expansion based on the inputs and states. These three functions together implement the actor function defined over signals.

Given an actor with n inputs and m outputs, let A denote its states, then the domain and codomain of the three functions of abstract actor semantics are defined as follows:

1. output function $\quad f\colon \mathcal{T} \times \mathcal{V}_\perp^n \times A \to \mathcal{V}_\perp^m$,
2. update function $\quad g\colon \mathcal{T} \times \mathcal{V}_\perp^n \times A \to A$,
3. request function $\quad h\colon \mathcal{T} \times \mathcal{V}_\perp^n \times A \to \mathcal{T}$.

Note that these functions are defined on values rather than signals.

The domain of the output function is a tuple including not only signal values but also the current tag and the states of actors. The indicates that the output function changes when states change.

The output function is closely related to the actor function defined over signals. Besides the state summarizing the history input signal, the output function takes the current input event (consists of the current tag and current input signal value) and generates the value of the output event at the current tag. Given the current tag and state, the output function needs to be monotonic as defined in Definition (36).

The update and request functions need to be strict functions. In other words, all inputs, includes the current input signal values and the states, must be known before their evaluations.

Sink actors have no output ports and their output functions simply discard inputs and return nothing. By default, the update function returns the old state. For some actors such as `Integrator` and `ModalModel`, new states are returned when their update functions are evaluated. The request function for an actor by default always returns the final tag. Some actors, such as the source actors and eventually strictly causal actors, can return a future tag that is not the final tag.

Our abstract actor semantics is similar to the S-function interface in Simulink but simpler, and it applies to not only CT models, but also other models of computation such as DE, SR, and FSM.

5.2.2 Abstract State Machine

A common way to rigorously define the operational semantics is the structural approach pioneered by Gordon Plotkin [29], which provides a state transition system for the language of interest. However, this approach is too detailed for the purposes of this dissertation. Instead, we use the abstract state machines (ASM), developed by Gurevichin [46], Huggins [53], and Borger [19], to formalize our operational semantics. The ASM keeps our operational semantics at a relatively high

abstraction level. In this way, our discussions will not be distracted by the a lot of notations and rules at very fine granularity.

In this subsection, we give some basics of ASM. The definitions introduced here are from [46, 19]. An ASM is defined as a set of **transition rules**. The basic rule is called **update rule**, which has the form,

$$f(x_1, x_2, \cdots, x_n) := x,$$

where f is an n-argument function and $x_1, x_2, \cdots, x_n,$ and x are expressions. Executing this rule changes the definition of the function f such that evaluating f with arguments $x_1, x_2, \cdots, x_n$ will yield the value specified by x. In other words, this rule specifies a **function update** and is just like an assignment.

Here is a common usage example. Let f be a *nullary* function, which takes no arguments and returns a value, then f is similar to a variable. In particular, suppose we have an update rule $f() := 2 + 3$, when this rule is executed, the value of the function f is updated to 5. In this dissertation, all the update rules are simple dates of nullary functions.

Another slightly complicated rule is called **conditional rule**, which has the form,

if `Condition` **then**

`Rule`

endif

where the guard `Condition` defines a precondition for the `Rule` to be applicable and it is a variable free first-order formula.

Rules are by default executed in sequence. In particular, they will be executed based on their appearance order. Take the following rule as an example, `Rule0` precedes `Rule1`.

if `Condition` **then**

`Rule0`

`Rule1`

endif

If the rules, `Rule0` and `Rule1`, are grouped into a single block with keywords **do-inparallel** and **enddo** as the following rule shows, then neither of these rules has priority over the other. This is

called **block rule**.

do-inparallel

`Rule0`

`Rule1`

enddo

If more than one rule is executed and their results contradict each other, then the machine will halt, because the machine is deterministic. We only consider deterministic ASMs in this dissertation.

All rules in a block rule will be applied to the machine simultaneously, while only those with satisfied guard conditions are applicable and executed. When these rules are executed, they will update the states of the machine. The new states will make other rules to be applicable. The sequence of the execution of rules gives the operational semantics of the machine.

5.2.3 Rules

In this section, we will treat a hybrid system as an ASM and define its rules. First, we define the states of a hybrid system and then the update rules for transforming the states.

A hybrid system is a network of actors interacting via signals. Given a hybrid system consisting of n actors and m signals, the set of its states, denoted as Σ, is defined as the following Cartesian product:

$$\Sigma = \mathcal{T} \times \Theta \times A^n \times \Delta,$$

where $\mathcal{T}$ is the tag set, $\Theta = \mathcal{V}_\perp^m$ is the set of m-value tuples, A^n is the Cartesian product of the states of all actors in the composition, Δ is $\mathcal{T}^n$, which stores the domain expansion requests from all actors.

The set of states Σ is called a *universe* in ASM. Let

$$\sigma = (t, s_1(t), s_2(t), \cdots, s_m(t), a_1, a_2, \cdots, a_n, t_1, t_2, \cdots, t_n) \in \Sigma, \tag{5.1}$$

where $a_i \in A, \forall i \in \{1, 2, \cdots, n\}$, be one distinguished state. Now we define a finite set of functions on Σ. The names of the functions in this set form a *signature* in ASM. The signature along with the interpretations of the functions on the universe is called a *static algebra*.

We extend the abstract functions defined for actors to be functions defined on Σ. For each actor $a_i, i \in \{1, 2, \cdots, n\}$, we redefine their abstract functions as follows:

1. output function $f_i' \colon \Sigma \to \Sigma$,
2. update function $g_i' \colon \Sigma \to \Sigma$,
3. request function $h_i' \colon \Sigma \to \Sigma$.

The new functions have the property that the elements of an input tuple σ that do not appear in the original definition of the function are unchanged in the output tuple σ'.

For a hybrid system model which is a network of actors, we define the following functions:

1. group-output $F \colon \Sigma \to \Sigma$,
2. group-state-update $G \colon \Sigma \to \Sigma$,
3. group-request $H \colon \Sigma \to \Sigma$,
4. advance-tag $AT \colon \Sigma \to \Sigma$,
5. fixed-point check $FP \colon \Sigma \to \{\texttt{TRUE}, \texttt{FALSE}\}$.

The group-output function F is defined as follows:

$$\forall \sigma \in \Sigma, F(\sigma) = f_n'(f_{n-1}'(\cdots(f_1'(\sigma)))). \tag{5.2}$$

That is, the output functions of all actors are composed in sequence, where the order is specified by a schedule. Similarly, we have the group-state-update function G and group-request function H defined as follows: $\forall \sigma \in \Sigma$,

$$G(\sigma) = g_n'(g_{n-1}'(\cdots(g_1'(\sigma)))), \tag{5.3}$$

$$H(\sigma) = h_n'(h_{n-1}'(\cdots(h_1'(\sigma)))). \tag{5.4}$$

Given a state tuple σ as shown in Equation (5.1), the advance-tag function AT generates the following state tuple,

$$\sigma' = AT(\sigma) = (\texttt{min}(t_1, t_2, \cdots, t_n), \underbrace{\bot, \bot, \cdots, \bot}_{m}, a_1, a_2, \cdots, a_n, t_1, t_2, \cdots, t_n), \tag{5.5}$$

where `min` function is defined in Section 2.1.1. The new state tuple only contains unknown signal values.

The fixed-point check function FP is a predicate on the state space Σ. The sole purpose of this function is to test for convergence of fixed point. Given a state σ, we have $FP(\sigma) = \texttt{TRUE}$ if σ is a fixed point for function F, that is,

$$F(\sigma) = \sigma.$$

Now we define the state transition rules for the ASM of hybrid systems.

`Update Rules:`

$\sigma := F(\sigma)$

$\sigma := G(\sigma)$

$\sigma := H(\sigma)$

$\sigma := AT(\sigma)$

`Conditional Rules:`

`Rule 0:` **if** $FP(\sigma) = \texttt{FALSE}$ **then**

$\sigma := F(\sigma)$

endif

`Rule 1:` **if** $FP(\sigma) = \texttt{TRUE}$ **then**

$\sigma := G(\sigma)$

$\sigma := H(\sigma)$

$\sigma := AT(\sigma)$

endif

`Block Rule:`

`Rule 2:` **do-inparallel**

`Rule 0`

`Rule 1`

enddo

The basic update rules are essentially assignment statements. For example, the rule $\sigma := F(\sigma)$ means that given a state σ, evaluate the group-output function with σ as the argument and update the state with the evaluation result $F(\sigma)$.

Rule 0 and 1 are conditional rules. Rule 0 says that given a state σ, if the fixed point has not

been reached, keep applying the update rule $\sigma := F(\sigma)$. This corresponds the inner loop starting from step 6 in Procedure (6). Rule 1 says that given a state σ, if the fixed point has been reached, the apply the update rules $\sigma := G(\sigma)$, $\sigma := H(\sigma)$, and $\sigma := AT(\sigma)$ in sequence.

The block rule specifies that Rule 0 and Rule 1 are executed in parallel. Because their guard conditions are mutually exclusive, only one rule is applicable at any time. After Rule 1 finishes, Rule 0 will be enabled to solve the fixed point at the new tag. This process corresponds to the outer loop starting from step 2 in Procedure (6).

The above operational semantics defined with the ASM is very similar to the operational semantics of synchronous languages, but has a metric associated with the interval between two clicks. Thus, actors interact not only via signals but also through time synchronization. This is one key characteristic for timed models of computation.

DE Example

We will use the DE with feedback example shown in Figure 4.8 as an example to illustrate how the rules defined above transform the states of the model. This model has four actors and three signals, therefore, the set of states is

$$\begin{aligned} \Sigma &= \mathcal{T} \times \Theta \times A^4 \times \Delta \\ &= \mathcal{T} \times \mathcal{V}_\perp^3 \times A^4 \times \mathcal{T}^4 \end{aligned}$$

A particular state $\sigma \in \Sigma$ is

$$\sigma = (t, s_1(t), s_2(t), s_3(t), a_1, a_2, a_3, a_4, t_1, t_2, t_3, t_4).$$

We use 1, 2, 3, 4 to index the four actors, `SingleEvent`, `Merge`, `TimedDelay`, and `TimedPlotter` respectively. The functions of these actors are also indexed correspondingly. Then we have the functions $F, G, H,$ and FP defined following Equation (5.2) to Equation (5.5).

Figure 5.1 shows how the operational semantics executes the DE model by applying the rules specified before. This figure only shows the first two unit executions after initialization. An unit execution is defined as the execution of the operational semantics over a *unit domain expansion*, which is the minimum domain expansion the operational semantics grants.

When a rule is applied, some elements in the state tuple σ are updated, indicated by the red color (or grep color in black-and-white printouts). When a rule is applied, some functions of actors

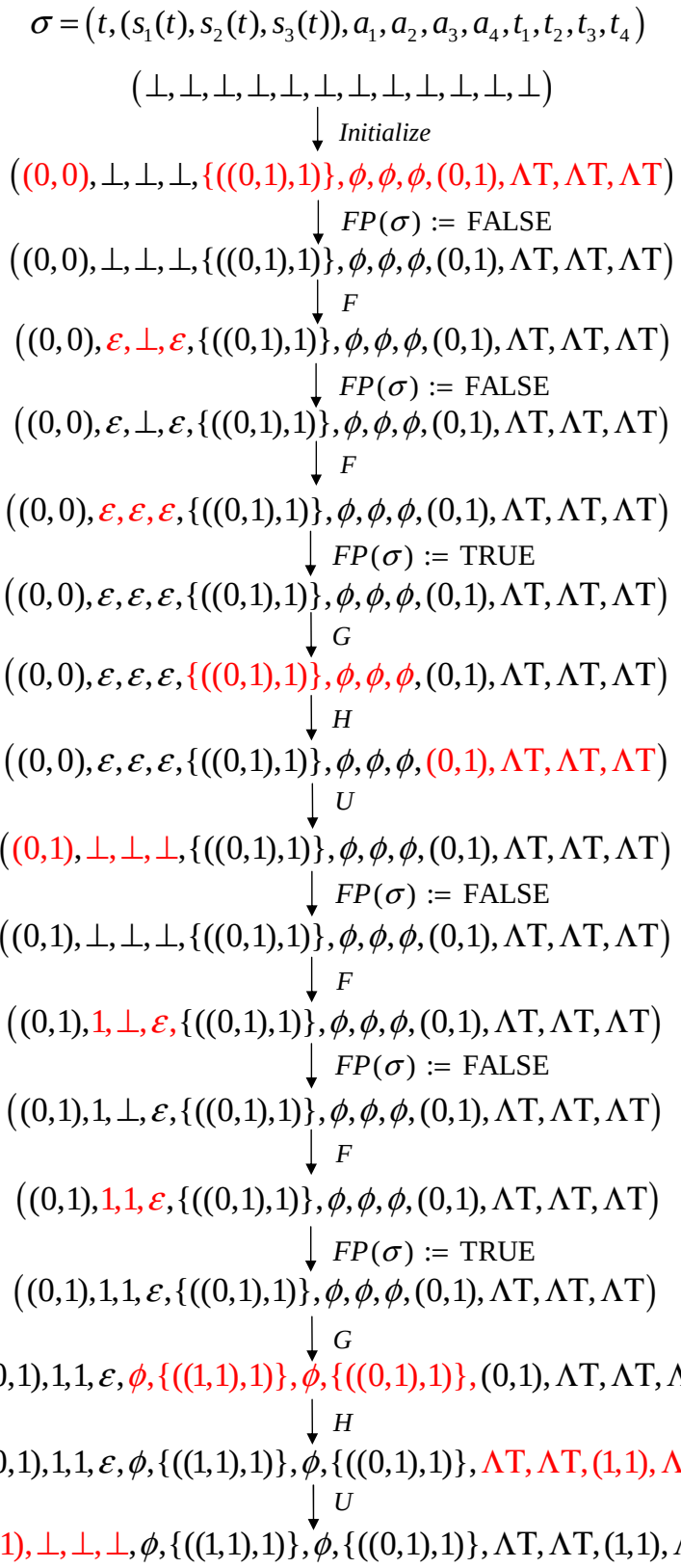

Figure 5.1. Illustration of how the operational semantics solves the DE model in Figure 4.8. This figure only shows the first two unit executions after initialization.

will be evaluated. If the new state resulting from the evaluations of a function is the same as the old state, then the evaluation is essentially a waste of computation. The number of such useless evaluations during an execution heavily depends on the quality of the execution schedule. Useless evaluations should be eliminated as much as possible to improve the efficiency of the operational semantics. An optimal schedule has the least number of useless evaluations.

The execution starts from tag $(0, 0)$, which is set by initialization. Rule 2 is first applied. Since the fixed point is not reached yet, Rule 0 is enabled. It take two applications of Rule 0 to find the fixed point. However, if we switch the execution order of the `Merge` actor and the `TimedDelay` actor, applying Rule 0 only once will be sufficient for finding the fixed point. In fact, the modified execution order has no useless evaluations during the process of finding fixed point solution.

After the fixed point is reached, Rule 1 is enabled and applied. The `Merge` and `TimedPlotter` actors always request to expand domain to the end of the tag set $\bigwedge \mathcal{T}$. Since the `TimedDelay` actor has not received any discrete events, it also requests domain expansion to the final tag. The `SingleEvent` actor requests domains to expand to $(0, 1)$, where it generate a discrete event. Therefore, after Rule 1 is applied, the new state contains a new current tag of $(0, 1)$.

After this, Rule 0 becomes enabled again and Rule 1 is disabled. Then the above process repeats. The second part of Figure 5.1 shows how the rules are applied at tag (0,1).

Note that except the `Merge` actor, all the other actors have states, which are discrete events. For example, the `TimedDelay` actor has a future discrete event $((1, 1), 1)$ at the end of the second unit execution. The `TimedPlotter` actor saves all discrete events it has received, such as the discrete event $((0, 1), 1)$.

5.2.4 Extra Rules for CT and Hybrid Systems Models

In the previous subsection, two rules have been defined for domain expansion and solving signal values. In this section, with a little bit more effort, we extend those rules further for CT models, especially for support nontrivial ODE solvers, event detection, and backtracking, such that we achieve a complete operational semantics for hybrid systems.

As we pointed out earlier in Section 2.4.2, it usually take more than one (intermediate) step for practical ODE solvers to complete an integration. During the integration process, the integration results from one intermediate step are used by the immediately following step. This essentially

requires the current tag and actor states to be updated at intermediate steps. However, an integration is not always guaranteed to succeed due to the error tolerance control and possible level crossing detections. Therefore, we need to support backtracking.

We define two more update functions:

1. integration-complete check $IC\colon \Sigma \to \{\texttt{TRUE}, \texttt{FALSE}\}$,
2. step-size-accepted check $SA\colon \Sigma \to \{\texttt{TRUE}, \texttt{FALSE}\}$.

These update functions, like the fixed-point check function FP, do not affect the states directly but rather return a boolean value to enable or disable certain rules. In particular, the integration-complete check function IC returns `TRUE` only if all intermediate integration steps have finished. The number of intermediate integration steps may be fixed if the solver is an explicit solver (such as RK-23 solver) or just bounded if the solver is an implicit one and needs to perform convergence check at run time (such as Backwards Euler solver). The step-size-accepted check function returns `TRUE` if *all* actors accept the integration results. An integration *succeeds* if it is complete and accepted by all actors.

Most actor accept the integration results by default. Only `Integrator`, `LevelCrossingDetector`, and `ModalModel` may have different opinions about the integration results. These actors can reject the domain expansion choice made by the previous advance-tag update function AT. If this happens, the integration needs to be retried with another *smaller* domain expansion. This is called backtracking.

Note that the actor states and current tag are modified during integration. Therefore, we need to introduce one more nullary function, $C : \emptyset \to \Sigma$, called **checkpoint**, to save the integration results and current tag after an integration succeeds and overwrites the current tag and actor states from the saved data if backtracking is necessary.

We add two extra update rules for checkpoint C as follows,

`Extra Update Rules:`

$$\sigma := C()$$
$$C() := \sigma$$

We modify Rule 1 and Rule 2 of the previously given operational semantics for supporting CT

and hybrid systems models:

```
Conditional Rule:
Rule 1:          if    FP(σ) = TRUE    then
                  σ := G(σ)
                  if    IC(σ) = TRUE    then
                      if    SA(σ) = TRUE    then
                          C() := σ
                      else
                          σ := C()
                      endif
                  endif
                  σ := H(σ)
                  σ := AT(σ)
                 endif
Block Rule:
Rule 2:          C() := σ
                 do-inparallel
                     Rule 0
                     Rule 1
                 enddo
```

This new Rule 1 adds two additional checks between the group-state update and group-request update. If the current integration completes and the integration results are accepted, then the newly resolved states are saved into the checkpoint. If the integration completes while the integration results are not rejected (e.g. by a `LevelCrossingDetector` actor), then the states of machine are updated with the previously saved states in the checkpoint. After this, all actors request domain expansion again. Some actors will request smaller domain expansions than what they used to request. Then the smallest request is chosen and all signal values are reset to unknown.

The modified Rule 2 adds an initialization update to the nullary function C.

Hybrid System Example

We will use a simple hybrid system example, shown in Figure 5.2, to illustrate how the newly introduced and modified rules defined in this subsection, together with the rules in the previous subsection, transform the states of the model.

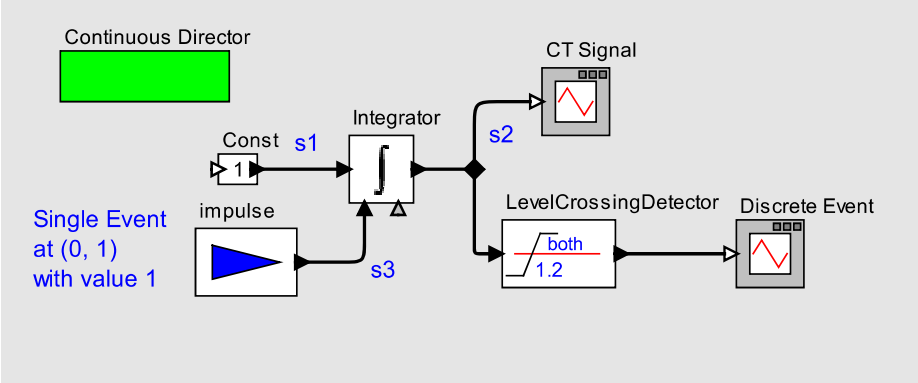

Figure 5.2. A simple hybrid system model.

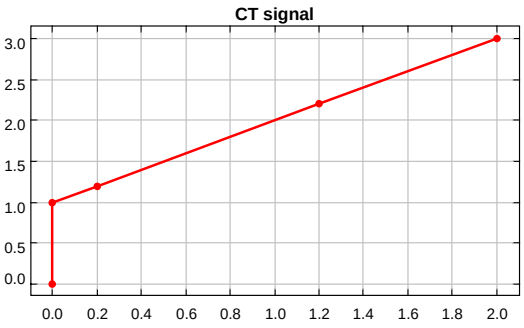

Figure 5.3. The output signal from the `Integrator` actor.

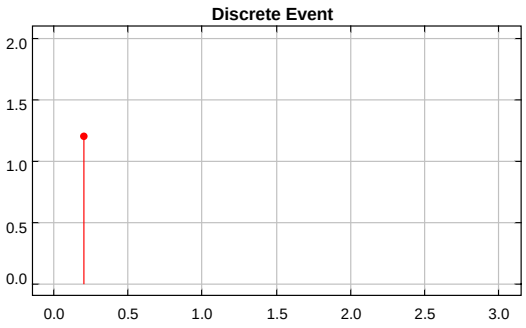

Figure 5.4. The output signal from the `LevelCrossingDetector` actor.

This model has 6 actors and 4 signals. We ignore the two `TimedPlotter` actors since they are simple and have been studied in the previous example. We use 1, 2, 3, 4 to index these four actors, `Const`, `impulse`, `Integrator`, and `LevelCrossingDetector` respectively. This order also reflects the execution order of these actions in schedule.

The `impulse` actor sends a DE *impulse* signal to the `Integrator`, which contains only one discrete event at tag $(0, 1)$ with a value of 1. The `Integrator` has an initial state of 0. The `LevelCrossingDetector` detects level crossings at level 1.2.

We use the RK-23 solver introduced in Section 2.4.2 as an example for solving the dynamic equations. The discussion here also applies to other solvers. The RK-23 solver requires two intermediate integration results to complete an integration. We require a state of the `Integrator` to be a tuple of two elements, the integration result and an integer number indicating which intermediate integration step is performed. Note that for different solvers, the states of an `Integrator` may be different.

One subtly about the integration is that if the step size is 0, then one integration is sufficient to give the output values. For example, according to the equations for RK-23 solver Equation (2.6) to Equation (2.9), the intermediate integrations have zero effect and can be ignored.

Recall that we need an extra state to checkpoints. The set of states is

$$\begin{aligned} \Sigma &= \mathcal{T} \times \Theta \times A^4 \times \Delta \\ &= \mathcal{T} \times \mathcal{V}_{\perp}^4 \times A^4 \times \mathcal{T}^4 \end{aligned}$$

A particular state $\sigma \in \Sigma$ is

$$\sigma = (t, s_1(t), s_2(t), s_3(t), s_4(t), a_1, a_2, a_3, a_4, t_1, t_2, t_3, t_4).$$

Figure 5.5 shows how the operational semantics solves the hybrid system model in Figure 5.2 by applying the transition rules. This figure only shows the first unit execution after initialization.

At the very beginning, Rule 2 is applied. After the checkpoint $C()$ is initialized, Rule 0 is applied because the fixed point is not reached. In this example, the fixed point happens to be reached after one application of Rule 0. The fixed-point check function FP returns `True`, therefore Rule 1 is applied. After the states are updated and before actors request for domain expansion, the integration-complete function IC checks whether the integration process completes. Since the integration step size is 0, the integration completes. No discrete events are detected also, therefore, the current states, including the current tag and all states of actors, are stored into the checkpoint ω.

Note that the states of the `LevelCrossingDetector`, which stores the previous received inputs, are not updated during its update functions. Instead, they are updated (from $\perp$ to 0) just before the checkpoint is updated. This is also true for sink actors such as `TimedPlotter`. The reason is that an integration is not guaranteed to succeed and may be rejected later. To avoid saving the integration results that might be abandoned later, these actors need to save *confirmed* states, which are those used to update the checkpoint.

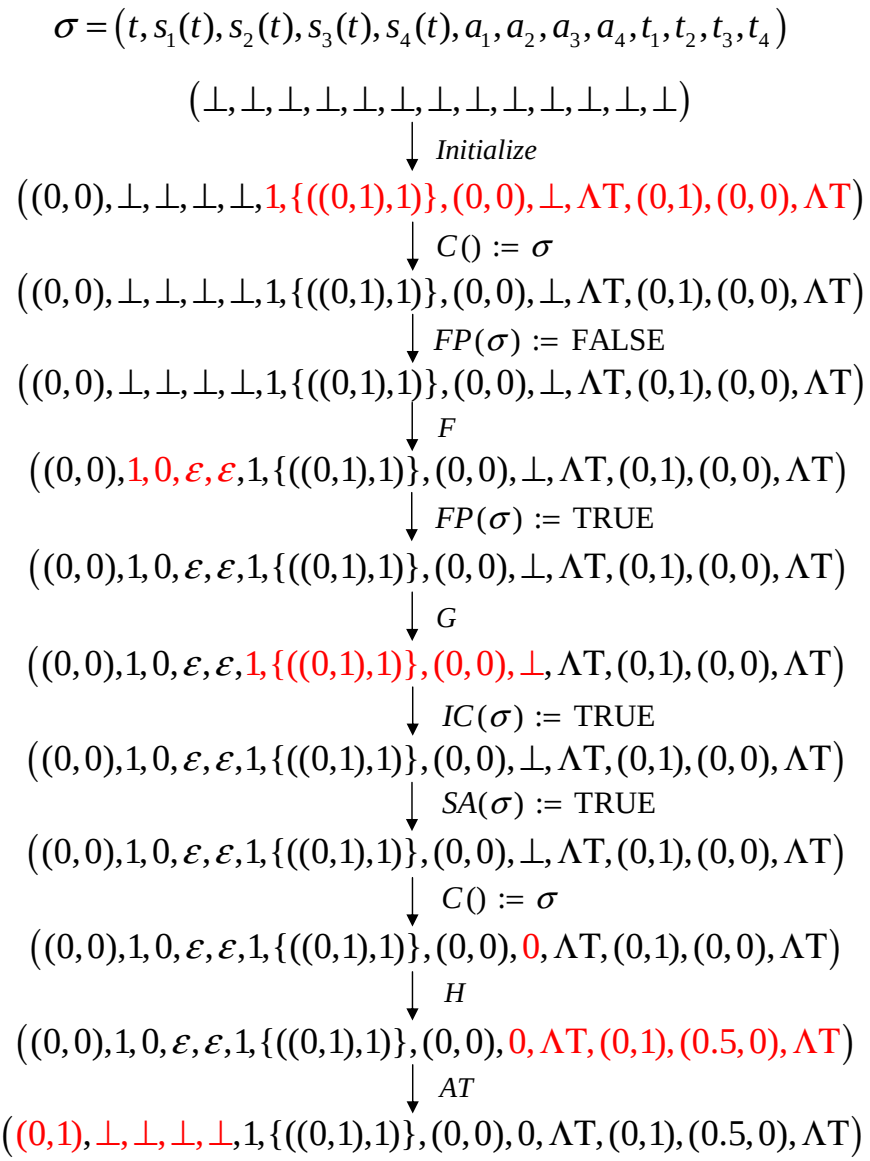

Figure 5.5. Illustration of how the operational semantics solves the hybrid system model in Figure 5.2. This figure only shows the first unit execution at tag $(0, 0)$.

At this point, the initial values of all signals at time 0.0 have been resolved.

Next, the `Integrator` requests to expand domain to $(0.5, 0)$ (assuming the integration step size is 1.0), while the `impulse` actor requests to expand domain to $(0, 1)$. The request from the `impulse` actor is smaller and the current tag is updated to $(0, 1)$, where the `impulse` actor produces a discrete event. Figure 5.6 shows how the operational semantics handles this discrete event.

$$
\begin{array}{c}
\big((0,1), \bot, \bot, \bot, \bot, 1, \{((0,1),1)\}, (0,0), 0, \Lambda T, (0,1), (0.5,0), \Lambda T\big) \\
\downarrow \; FP(\sigma) := \text{FALSE} \\
\big((0,1), \bot, \bot, \bot, \bot, 1, \{((0,1),1)\}, (0,0), 0, \Lambda T, (0,1), (0.5,0), \Lambda T\big) \\
\downarrow \; F \\
\big((0,1), 1, 1, 1, \varepsilon, 1, \{((0,1),1)\}, (0,0), 0, \Lambda T, (0,1), (0.5,0), \Lambda T\big) \\
\downarrow \; FP(\sigma) := \text{TRUE} \\
\big((0,1), 1, 1, 1, \varepsilon, 1, \{((0,1),1)\}, (0,0), 0, \Lambda T, (0,1), (0.5,0), \Lambda T\big) \\
\downarrow \; G \\
\big((0,1), 1, 1, 1, \varepsilon, 1, \phi, (1,0), 0, \Lambda T, (0,1), (0.5,0), \Lambda T\big) \\
\downarrow \; IC(\sigma) := \text{TRUE} \\
\big((0,1), 1, 1, 1, \varepsilon, 1, \phi, (1,0), 0, \Lambda T, (0,1), (0.5,0), \Lambda T\big) \\
\downarrow \; SA(\sigma) := \text{TRUE} \\
\big((0,1), 1, 1, 1, \varepsilon, 1, \phi, (1,0), 0, \Lambda T, (0,1), (0.5,0), \Lambda T\big) \\
\downarrow \; C() := \sigma \\
\big((0,1), 1, 1, 1, \varepsilon, 1, \phi, (1,0), 1, \Lambda T, (0,1), (0.5,0), \Lambda T\big) \\
\downarrow \; H \\
\big((0,0), 1, 1, 1, \varepsilon, 1, \phi, (1,0), 1, \Lambda T, \Lambda T, (0.5,0), \Lambda T\big) \\
\downarrow \; AT \\
\big((0.5,0), \bot, \bot, \bot, \bot, 1, \phi, (1,0), 1, \Lambda T, \Lambda T, (0.5,0), \Lambda T\big)
\end{array}
$$

Figure 5.6. This figure shows the *second* unit execution of the operational semantics at tag $(0, 1)$, which handles a discrete event at tag $(0, 1)$.

Again, Rule 0 is first applied. When the group-output function F is evaluated, the output function of the `impulse` actor generates an output signal value $s_3(0, 1) = 1$. The evaluation of the output function of the `Integrator` generates an output signal value $s_2(0, 1) = 1$.

The fixed point is reached after Rule 0 is applied once. Then Rule 1 is applied. The update function of the `impulse` actor resets its state to an empty set, indicating no future events to be produced. The `Integrator` updates its integration result to 1 as the result of the discrete event from the `impulse` actor.

Since the integration step size is still 0, from tag $(0, 0)$ to $(0, 1)$, the integration completes after one execution and the fixed point has reached. The fixed point solution at this tag gives the *final* values of all signals at time 0, because there are no more discrete events at this time point. The checkpoint is updated.

Then, the `impulse` actor requests domain expansion to the final tag and the `Integrator` actor still requests an expansion to tag $(0.5, 0)$. The request from the *Integrator* is accepted and Rule 0 is applied to try to resolve the discrete representation of signals at tag $(0.5, 0)$.

Figure 5.7 shows how the operational semantics tries to solve a discrete representation of continuum dynamics during a continuous interval, and how to backtrack due to a level crossing reported from the `LevelCrossingDetector` at tag $(0.2, 0)$.

It takes three intermediate steps to complete the whole integration according to the RK-23 solver. The last element in the state tuple of the `Integrator` indicates which intermediate step is being performed. For example, $(1, 0)$ shows the first step and $(1.5, 1)$ shows the second step.

The update of the current tag also depends on which intermediate step is to be performed. In particular, for a step size of 1, the current tag is updated to 0.5, 0.75, and 1 at intermediate step 0, 1, and 2. An integration completes when the second element in the state of the `Integrator` reaches 3 (This number is particular for RK-23 solver. For other solvers, the number may be different. For implicit ODE solvers, the criteria for completion of integration usually depend on the convergence of integrated results instead of a fixed number.).

The output signal value of signal s_2 is resolved to be 2. Therefore, the `LevelCrossingDetector` detects a level crossing and rejects the current integration results and domain expansion. The checkpoint was used to update the current states. Then all actors request a more reasonable domain expansion. The `LevelCrossingDetector` requests to expand domain to $0.2, 0$ based on its newly received inputs and previous state.

After the continuous dynamics from tag $(0, 1)$ to $(0.2, 0)$ is resolved, meaning the fixed point of all signal values at tag $(0.2, 0)$ are solved, we get the initial values of all signals at time 0.2. These states update the checkpoint again.

Then, the `LevelCrossingDetector` requests to expand domain to tag $(0.2, 1)$, where this actor will generate a discrete event. The process of how to handle this discrete event is similar to Figure 5.6 shows. After this discrete event is handled, the final values of all signals at time 0.2 are resolved and states are saved into the checkpoint. Then the domains get expanded further.

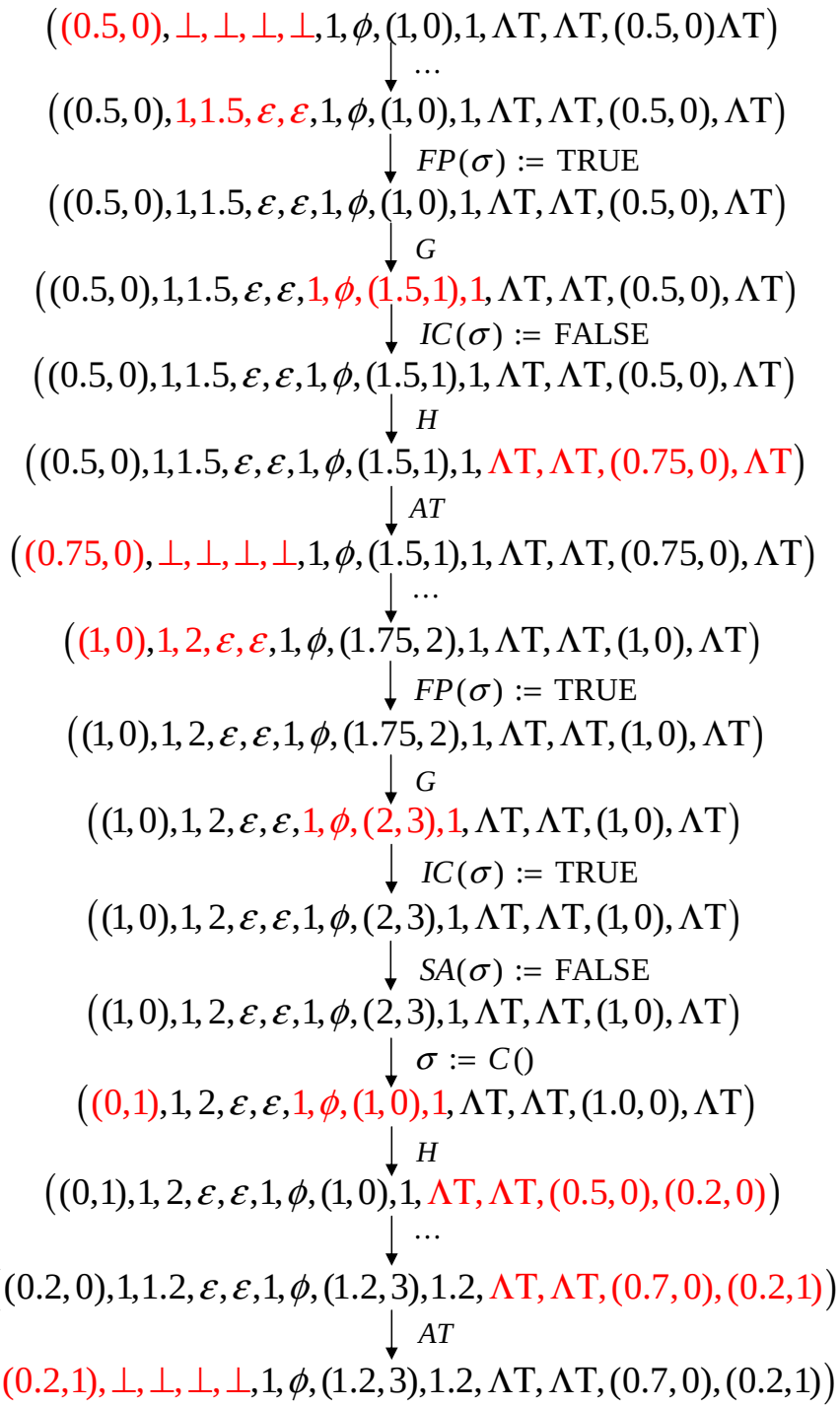

Figure 5.7. This figure shows the *third* unit execution, where the integration is first tried from tag $(0,1)$ to tag $(1,0)$, and was rejected by the `LevelCrossingDetector`. Then the integration was retried from tag $(0,1)$ to tag $(0.2,0)$. The $(0.2,0)$ was requested by the `LevelCrossingDetector`.

The whole process keeps repeating until the expanded domain covers the whole tag set. In which case, the operational semantics finishes the construction of the discrete representation of all signal values and the execution terminates.

A portion of the execution results, from $(0, 0)$ to $(3, 0)$, of the hybrid system model in Figure 5.2 is shown in Figure 5.4. The dots in the figure are the actual computed values, which reflect the discrete subset and the representation. These results comply with the above analysis results.

5.2.5 Optimization

As we saw in the process of how our operational semantics solves the DE example, useless evaluations may happen when solving fixed point solutions. A good execution order can help reduce the number of useless evaluations. For example, the modified execution order for the DE model and the execution order for the hybrid system model. The key for reducing useless evaluations is (with the best effort) to ensure that when an actor is executed, its inputs are ready. That is, the execution order should honor the data dependencies between actors. Thus, causality interfaces, which basically construct a dependency graph for actors, can help to achieve this purpose.

Some operational semantics classify actors in a hybrid systems into clusters, a *state-transition cluster*, an *output cluster*, and an *event-generation cluster* [66, 30, 62]. When executing hybrid systems, actors in the state-transition cluster are first executed, followed by actors in the output cluster, and then the event-generation clusters. This operational semantics tries to minimize the invocations of each individual actor, based on the following observations:

1. The intermediate integration results are not final and not needed for actors to generate outputs. Therefore, *all* actors in the output cluster can be invoked after integrators resolve the states.

2. Similar to actors producing outputs, an event-generator actor only operates on the final integration results. Therefore, they are executed after integration results are ready. What is more, they are executed after the output cluster because usually the actors in the output cluster provide inputs for the event-generation cluster. This execution order reflects data dependency.

3. When discrete events interact with each other, time does not advance. Therefore, continuous dynamics is not involved. In other words, there is no need to invoke state-transition actors.

Consequently, an execution of hybrid systems can be treated as a sequence of phase executions, *continuous phase execution* and *discrete phase execution*. The discrete phase execution goes first,

and then these two phase executions are invoked alternatively [62]. At the beginning of an execution (for example, at tag $t_\perp = (0, 0)$), all signal values are treated as the *initial* values at time 0. The *first* discrete phase execution essentially establishes the *final* values of the signals at time 0. Then, the operational semantics invokes the continuous phase execution to solve the continuum over the interval until the next discrete event happens. Assume the next discrete event is at tag $(r, 1)$, then the continuous phase execution resolves the initial values at time r, to be specific, the values at tag $(r, 0)$. After that, a discrete phase execution is invoked to handle the discrete events at time r.

A discrete phase execution at a time $r \in \mathcal{R}_0$ can be treated as a fixed point iteration, where the fixed point is reached when there is no more *future* discrete event happening at time r. The process of finding this fixed point is identical to what we did for the hybrid system model in Figure 5.5 and Figure 5.6, where the first figure solves the initial signal values and the second figure solves the final values. A discrete phase execution always start with the initial values of signal and find the final values of signal, which is the fixed point. It always takes a finite number of executions to find the fixed point due to the finite change constraint on signals. For example, it takes two executions to find the fixed point of the discrete phase of execution at time 0 for the hybrid system example.

After the final values at time r are calculated, a continuous phase execution proceeds. The process keeps repeating until the discrete phase execution at the time where the final tag is completes.

5.3 Modularity

The optimization technique described in Section 5.2.5 has so deep an impact on executing hybrid systems that sometimes they are treated as an integrated part of the operational semantics. However, an optimization is always a trade-off among several objectives, it and this one does not come for free. In fact, the aforementioned optimization technique causes problems for another objective of the operational semantics, which is to support modularity. This section is dedicated to examining and solving those problems.

Recall that the foundation of optimized operational semantics relies on the classification of actors into clusters, such that during different phases of executions, only relevant actor clusters will be invoked. In this way, the number of evaluations of actor functions are reduced. However, when modularity is required, in particular, when hierarchy is used to group a subset of a system into

an *opaque* actor, we do not have a clean and clear way to classify actors into clusters. Thus, the optimized technique does not work any more.

We use the model in Figure 5.8 as an illustration. This model has three parts: the upper part is a simple second-order ODE; the lower part is almost identical to the upper part except that the second `Integrator` is hidden inside the next level of hierarchy as Figure 5.9 shows; the right part is a test facility to monitor the difference between the two integration results from the upper and lower models.

If we examine the upper model only, we can immediately tell that the two `Integrator` actors along with the Const and AddSubtract consist of a state-transition cluster, and there is no output cluster nor event-generation cluster. We can perform the optimized operational semantics based on this classification.

However, if we examine the lower model only, we will see the problem. How do we classify the composite actor called `Embedded CT Model`? Without looking inside the actor for more details, we cannot tell. In order to get correct execution results, in practice, we have to include this actor into all three actor clusters, i.e., this composite actor is a state-transition actor, an output actor, and an event-generation actor. Apparently, this classification *increases* the number of function evaluations instead of reducing it.

What is more, when we look inside the composite actor, the difference between a state-transition actor and an output actor is blurred. For example, if we insert a `Scale` actor with a scale factor of unity into the `Embedded CT Model` as shown in Figure 5.13, the execution results should not be changed, because the `Scale` actor does not change signal values and is semantically equivalent to a wire. If we treated the model as an isolated entity, since the output from the `Scale` actor does not affect the input of `Integrator`, it is treated as an output actor. However, when used in composite actor, the `Scale` actor cannot be treated as an output actor any more, because it in fact belongs to the state-transition actor cluster. In order to guarantee the correctness of execution, every actor inside a composite actor basically has to be included into all three clusters. Again, this adds computation cost instead of lowering it.

What is more, classifying almost every single actor into any of the three clusters makes the software implementation rather complicated and error prone. Figure 5.14 shows the execution of the same model in Figure 5.8 in Ptolemy 4.0.1. Unlike Figure 5.10, which shows that the difference between the two integration results is (almost) identical to 0 (because they are under the control of

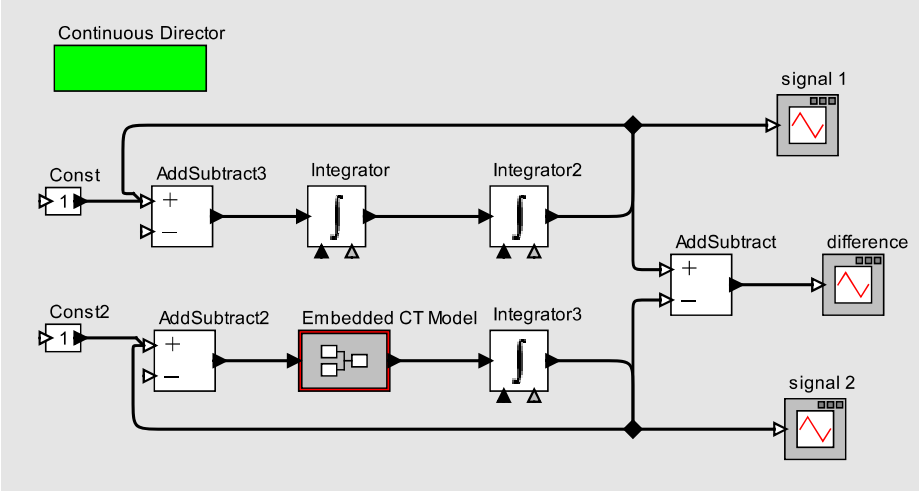

Figure 5.8. A model tests the invariant that introducing hierarchy does not alter the model's semantics.

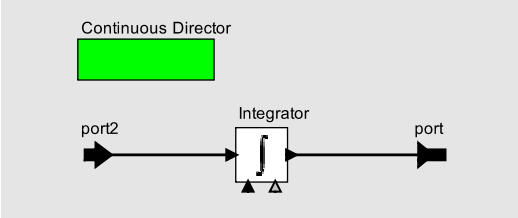

Figure 5.9. The inside of the `Embedded CT Model`.

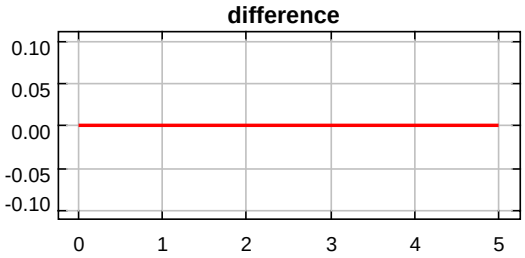

Figure 5.10. This plotter plots the difference of the integration results.

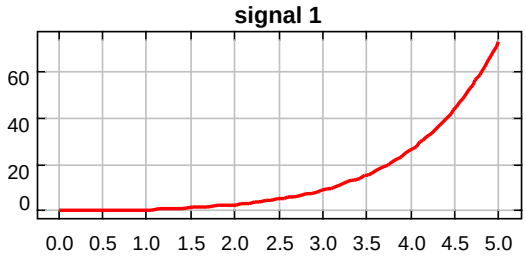

Figure 5.11. This plotter plots the integration result from the model without hierarchy.

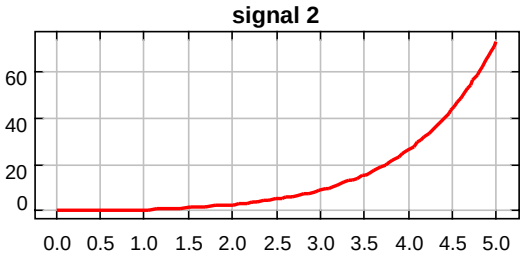

Figure 5.12. This plotter plots the integration result from the model with hierarchy.

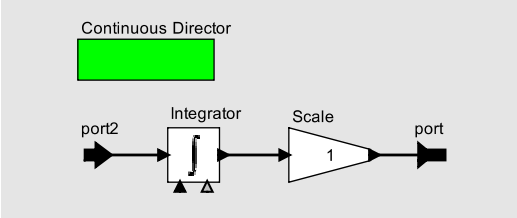

Figure 5.13. A modified `Embedded CT Model` with a `Scale` actor.

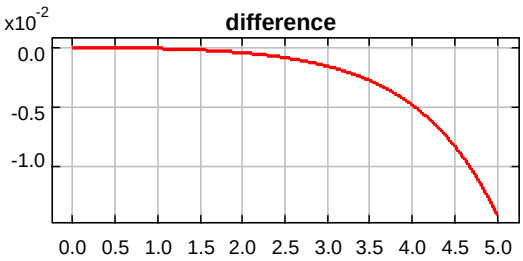

Figure 5.14. This figure shows the incorrect execution results generated from an incorrect software implementation.

the same ODE solver, RK-23 solver introduced before), this figure shows a non-negligible difference. This difference is by no means caused by truncation errors or numerical error (because they are not at the same magnitude (10^{-2} vs. 10^{-5}), but by an incorrect software implementation. In fact, simulating the model in Ptolemy 4.0.1 takes substantial time, because the ODE solver tries to reduce the step size to compensate for the incorrect software implementation.

5.3.1 Arbitrary Composition

As we pointed out earlier, it is challenging to model, design, and simulate heterogeneous systems that mix digital controllers realized in hardware and software with the continuous dynamics of physical systems. Our operational semantics, which extends the synchronous language principles by associating time semantics between clicks, and introduce mechanisms for integration of continuous dynamics, effectively unifies synchronous/reactive, discrete-event, and continuous-time concurrent models of computation. Because of this commonality, we achieve arbitrary compositions of CT, DE, and synchronous language models and modular executions of such compositions. This opens up very interesting possibilities for joint modeling of software control systems and controlled physical plants.

Note that we are not proposing to use our operational semantics to replace all the aforementioned individual models of computation, in that execution efficiency, modeling convenience, and synthesizability all may be compromised. In fact, there are good reasons to use all three of them.

5.3.2 Newton's Cradle Model with Distributed Control

Now we give an example to illustrate the power of our operational semantics for supporting modular execution. This example is modified version of the Newton's cradle model shown in the

Section 1.3. Instead of having a central controller, this model requires each ball to control their behaviors. The model is shown in Figure 5.15 and the mode controller is in Figure 5.16.

Recall that we assume that balls are placed in a straight line $x - dimension$ and the collisions happen in the $x - y$ dimensions only. Therefore, a ball can at most have two neighbors, and this is the reason for each ball to have extra ports than the original ball definition in Figure 1.11. In particular, a ball in this model needs to monitor the x, y positions and velocities from both side, which adds 6 more ports, shown in Figure 5.17, where the `Ball Dynamics` actor is the same as the original model and defines the second-order ODE for the ball dynamics.

Inside the mode controller of each individual ball, there are three states, `init`, `state`, and `transient` sharing the same refinement as shown in Figure 5.17. There are three outgoing transitions of the `state` state. The transition with guard "`guard1 && !(guard2)`" models the collision with the ball on the left side; the transition with guard "`guard2 && !(guard1)`" models the collision with the ball on the right side; and the transition with guard "`guard1 && guard2`" models the *simultaneous* collisions from both sides.

The guard expressions `guard1` and `guard2` are specified as string parameters:[3]

```
guard1: x_left_isPresent && y_left_isPresent && theta_left_dot_isPresent
        && (distanceFunction(x, x_left, y, y_left) <= diameter)
        && (theta_left_dot > theta_dot),

guard2: x_right_isPresent && y_right_isPresent && theta_right_dot_isPresent
        && (distanceFunction(x, x_right, y, y_right) <= diameter)
        && (theta_dot > theta_right_dot).
```

When the left ball and right ball hit the middle ball at the same time, to be accurate, say at tag $(r, 0)$, the transition with guard "`guard1 && guard2`" of the middle ball is enabled. We need to examine this scenario more carefully. Let us use two DE signals, called s_l and s_r, to model the interactions of this middle ball with the left and right ball respectively, then both s_l and s_d will have a discrete event at tag $(r, 1)$ (Recall the definition of the `LevelCrossingDetector` in Section 3.2.8).

Note that meanwhile, the right ball will receive a DE signal s'_r which is the same as s_r, containing one discrete event at $(r, 1)$, because the mode controller of the right ball has the transition with guard "guard1 && !(guard2)"" enabled too. Similarly, the left ball receives a DE signal $s'_l = s_l$ too.

At tag $(r, 1)$, recall that we allow only two balls to be involved in a collision due to the lack of

[3]Since the guard expressions are pretty complicated, using a string parameter will greatly improve the readability of the finite state machine. However, when I write this dissertation, this facility is not fully functioning, and the execution results are achieved by substituting the content of the string parameters into the guard expressions.

a general analytic solution for multi-body collision problems, only one transition (corresponding to one of the discrete events) can take place. Suppose the middle ball collides with the left ball, then they exchange their velocities (assuming perfectly elastic collisions). At the same tag, both the left ball and right ball take transitions because of the discrete events in signal s'_r and s'_l, and update their velocities.

Then at the next tag $(r, 2)$, in contrast to our guess that the middle ball will collide with the right ball, there is no collision. That is because the velocity of the right ball was modified in the previous tag and the new state does not cause a collision. In other words, the discrete event in s_r and s'_r is gone. Apparently, this result does not comply with our intuition.

The solution is to introduce a temporary storage to save the discrete event and transient states to trigger simultaneous interactions between balls. In this model, we introduce a parameter `TemporaryState` to store the velocity of one ball and the `transient` state. The dynamics works as follows.

1. When the middle ball collides with both sides at tag $(t, 0)$, transition with guard "`guard1 && guard2`" is enabled.
2. At tag $(r, 1)$, the transition is taken, where the middle ball takes the velocity of the left ball and the velocity of the right ball is stored into `TemporaryState`. The `transient` state is activated.
3. Because the `transient` state is a transient state, its outgoing transient is enabled immediately.
4. At tag $(r, 2)$, the outgoing transition of the `transient` state is taken, which sets velocity the middle ball to the velocity saved in parameter `TemporaryState`.

One execution of such case is shown in Figure 5.18.

Beside illustrating that our operational semantics can handle this model correctly, another more practical motivation of the distributed control is to make the model more scalable. In particular, if we want to add more balls, instead of modifying the central mode controller, we only need to configure the topology of the layout of the balls' positions, while the inside details of balls, such as the mode controller, are untouched. Note this benefit is made possible by the operational semantics we gave before.

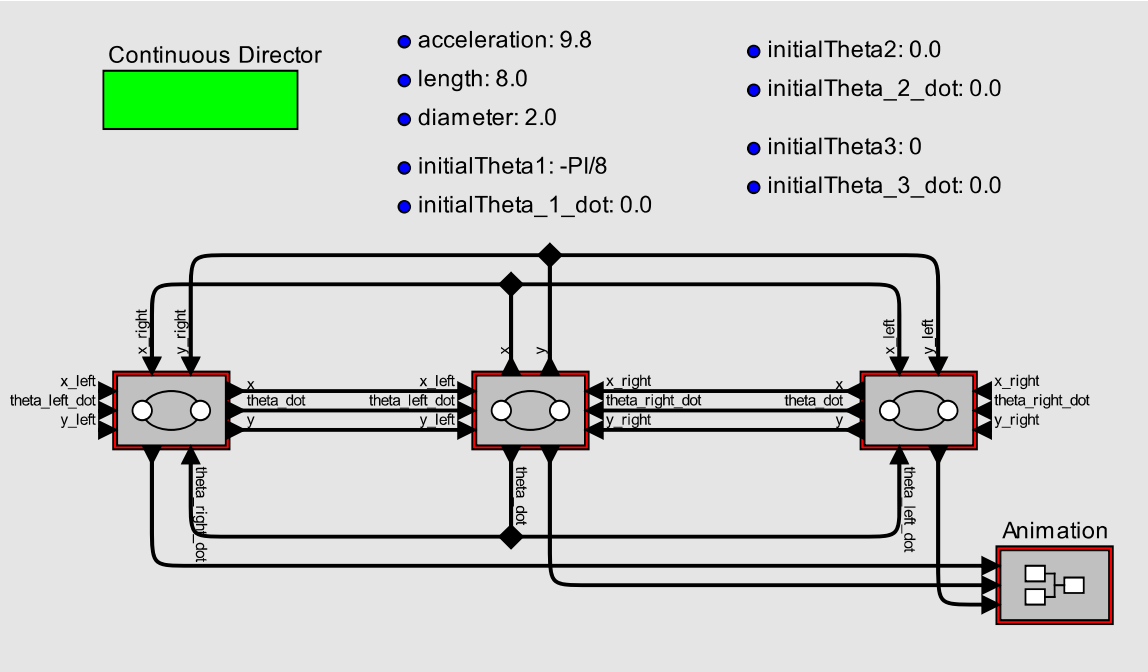

Figure 5.15. A modified Newton's cradle model with distributed control, comparing to the original model in Figure 1.10.

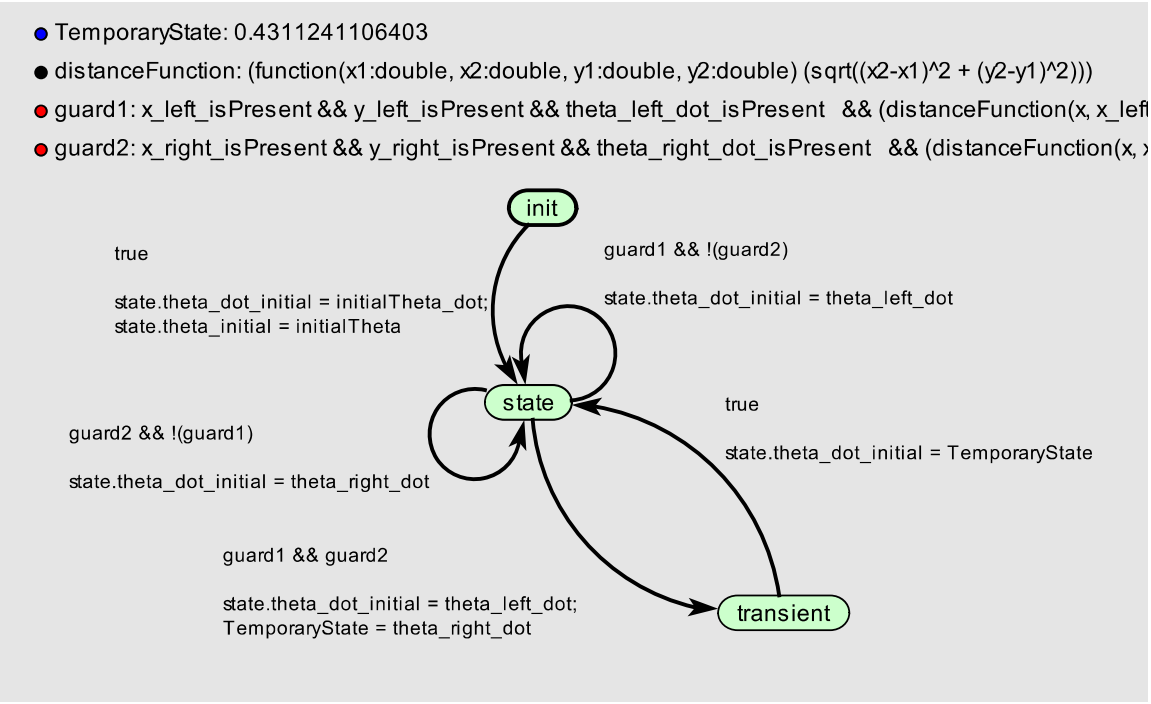

Figure 5.16. The mode controller of individual balls.

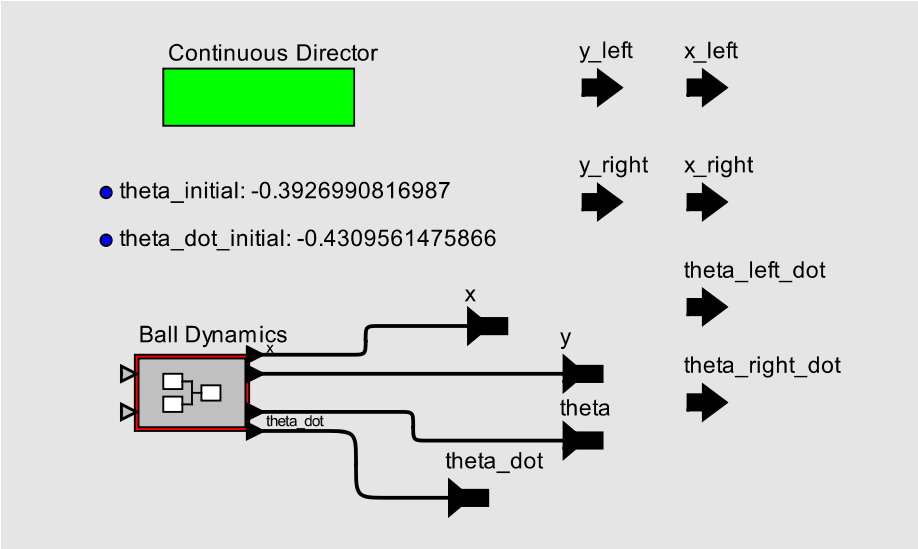

Figure 5.17. The refinement of the states in Figure 5.16.

5.4 Software Implementation

The operational semantics proposed in this section has been implemented in HyVisual [28, 21], which is a software tool for modeling and simulating hybrid systems. HyVisual is part of the Ptolemy II software framework [59, 22], which is written in Java and available in open-source form (BSD-style license) at http://ptolemy.org.

Output and Update Functions

The output and update functions in the abstract actor semantics are abstracted by the *Executable* interface. Figure 5.20 listed a few important methods specified in the executable interfaces. The *postfire* method implements the update function, and the *prefire* and *fire* methods together implement to the output function. The *prefire* method tests the precondition for an actor to be able to fire, and the fire method does real computation. A typical usage of the prefire and fire methods is as follows,

```
if (prefire()) {
   fire();
}
```

The prefire method can be incorporated as part of the fire method. Separating the prefire method from the fire method is an optimization.

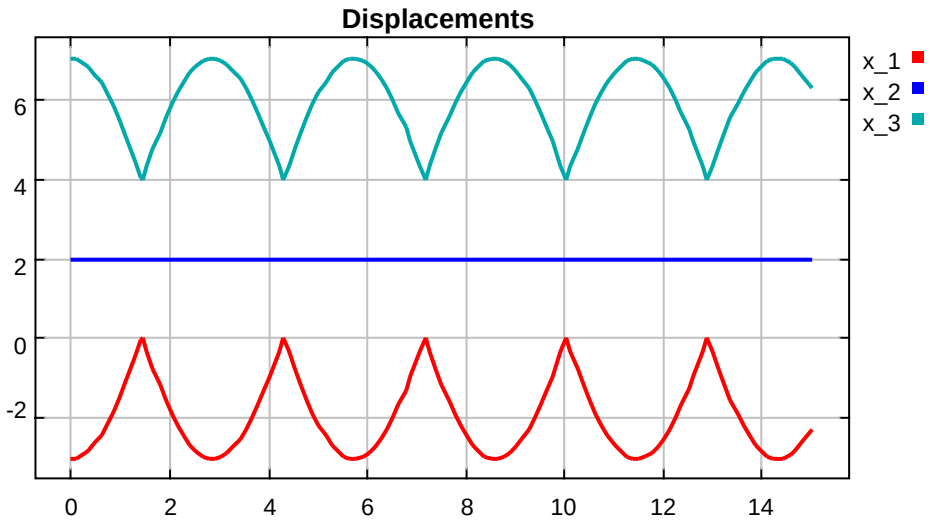

Figure 5.18. The displacements balls with an initial configuration that ball #1 and ball #3 are moved away in opposite direction from their equilibrium positions with the same angels.

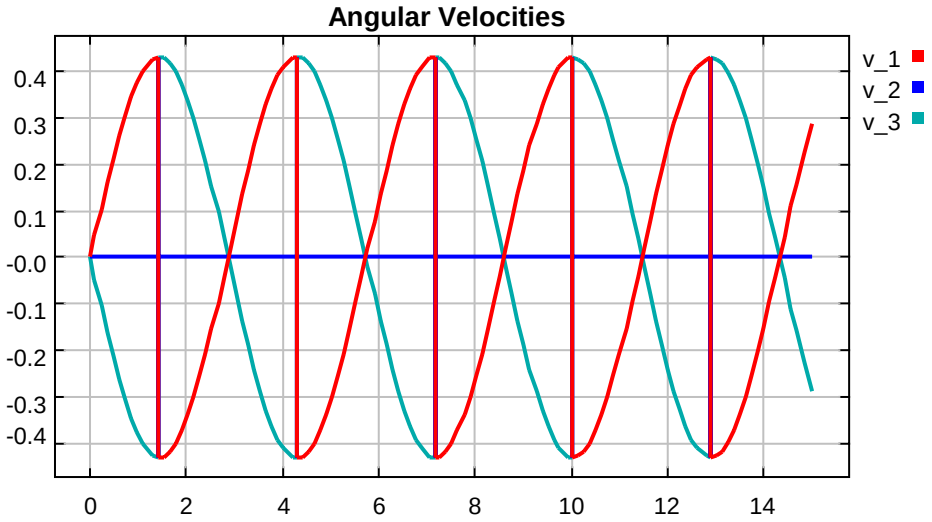

Figure 5.19. The velocities of balls with an initial configuration that ball #1 and ball #3 are moved away in opposite direction from their equilibrium positions with the same angels.

initialize	Initialize the actor.
prefire	Test preconditions for firing.
fire	Read inputs and produce outputs.
postfire	Update the state.
wrapup	End execution of the actor.

Figure 5.20. The key flow of control operations in the Ptolemy II abstract semantics.

The prefire and fire methods may be executed multiple times before a fixed point is reached, while the postfire method can only be invoked once after the fixed point is reached.

Director

The rules for the operational semantics given in the previous section are implemented by a *Director* class, which decides the evaluation order of the output and update functions.

Instead of associating each actor a request function, a director implements a *fireAt* method, which is shared by all actors under the control of the same director. A typical usage of the fireAt method is as follows,

```
director.fireAt(someFutureTime);
```

If an actor needs to request domain expansion, it calls the fireAt method to register a future tag into the director.

According the update rule for the advance-tag function, the director is responsible to collect all requests from actors, to grant the smallest request, and to discard the other requests. The data structure for storing requests can be as simple as a set. In practice, the requests are stored into a linked list where a smaller request precedes a bigger one. This makes the job of finding the smallest request easy.

Chapter 6

Zeno Hybrid Systems

The dynamics of physical systems at the macro scale level (not considering effects at the quantum level) are continuous in general. Even in a digital computer that performs computation in a discrete fashion, its fundamental computing elements (transistors) have continuous dynamics. Therefore, it is a natural choice to model the dynamics of physical systems with ordinary differential equations (ODEs) or partial differential equations (PDEs). However, modeling a physical system with only continuous dynamics may generate a stiff model, because the system dynamics might have several time scales of different magnitudes. Simulating such stiff models in general is difficult in that it takes a lot of computation time to get a reasonably accurate simulation result.

Hybrid system modeling offers one way to resolve the above problem by introducing abstractions on dynamics. In particular, slow dynamics are modeled as piecewise constant while fast dynamics are modeled as instantaneous changes, i.e., discretely. In this way, the remaining dynamics will have time scales of about the same magnitude and the efficiency of simulation, especially the simulation speed, is greatly improved. However, special attention must be devoted to hybrid system models because Zeno hybrid system models may arise from the abstractions.

An execution of a Zeno hybrid system generates Zeno signals with an infinite number of discontinuities during a finite time interval. The limit of the set of discontinuous time points of a Zeno signal is called the *Zeno time point*. The signal value at the Zeno time point is called the *Zeno state*. Because handling each discontinuity takes a non-zero and finite computation time, the execution of a Zeno hybrid system inevitably halts near the Zeno time point.

Some researchers have treated Zeno hybrid system models as over abstractions of the physical

systems and tried to rule them out by developing theories to detect Zeno models [95, 4, 7]. However, because of the intrinsic complexity of interactions between continuous and discrete dynamics of hybrid systems, a general theory, which can give the sufficient and necessary conditions for the existence of Zeno behaviors of hybrid system models with nontrivial dynamics, is still not available (and does not appear to be anywhere on the horizon).

Some researchers have tried to extend the simulation of Zeno systems beyond the Zeno time point by regularizing the original system [55, 5] or by using a sliding mode simulation algorithm [75]. The regularization method requires modification of the model structure by introducing some lower bound on the interval between consecutive discrete transitions. However, the newly introduced lower bound invalidates the abstractions and assumptions of the instantaneity of discrete transitions. Consequently, the simulation performance might suffer from the resulting stiff models. Furthermore, different behaviors after the Zeno time may be generated depending on the choices of regularizations. This may not be desirable because the physical system being modeled typically has a unique behavior. The sliding mode algorithm tends to be more promising in simulation efficiency and uniqueness of behaviors, but it only applies to special classes of hybrid system models.

A new technique to extend simulations beyond the Zeno time point is presented in [8], where a special class of hybrid systems called *Lagrangian* hybrid systems are considered. Rather than using regularizations or a sliding mode algorithm, the dynamics of a Lagrangian hybrid system before and after the Zeno time point are derived under different constraints. In this chapter, we extend the results in [8] to more general hybrid system models.

In this chapter, we argue that the presence of Zeno behavior indicates that the hybrid system model is incomplete by considering some classical Zeno models that incompletely describe the dynamics of the system being modeled. This motivates the systematic development of a systematic method for *completing* hybrid system models through the introduction of new *post-Zeno* states, where the completed hybrid system transitions to these post-Zeno states at the Zeno time point. The post-Zeno states of the completed systems describe the system dynamics after Zeno time points.

Then we propose a practical simulation strategy for constructing approximations of Zeno signals by leveraging the completed hybrid system model. This technique allows executions to *jump over* Zeno time points and proceed with the dynamics defined in the post-Zeno states by discarding an infinite number of discrete events when the approximations are close enough to the Zeno signals. The approximations of Zeno signals contain a *finite* number of discrete events and can be simulated

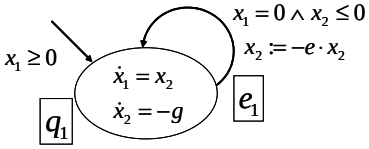

Figure 6.1. A hybrid system model of a simple bouncing ball.

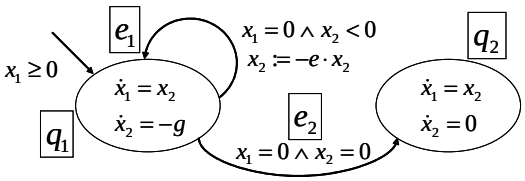

Figure 6.2. A more complete hybrid system model of the bouncing ball.

in a finite number of computation steps according to the operational semantics given in Chapter 5. In this way, we can simulate a Zeno hybrid system model beyond its Zeno time point and reveal the complete dynamics of the system being modeled.

Partial work presented in this chapter was published in [96].

6.1 Example Zeno Hybrid Systems

Before we get into the technique details of the algorithm on extending simulation beyond Zeno time points, we would like to investigate some classical Zeno hybrid system models including the bouncing ball model [93] and the water tank model [55], and show that they do not completely describe the behavior of the original physical systems.

6.1.1 Bouncing Ball

Considering a ball bouncing on the ground, where bounces happen instantaneously with a restitution coefficient $e \in [0, 1]$. A hybrid system model for this system is shown in Fig. 6.1. This model has only one state q_1 associated with a second-order differential equation modeling the continuous dynamics, where the variables x_1 and x_2 represent the ball's position and velocity respectively, and $\dot{x_1} = x_2$, $\dot{x_2} = -g$. From this one state, there is a transition e_1 that goes back to itself. The transition has a guard expression, $x_1 = 0 \wedge x_2 \leq 0$, and a reset map, $x_2 := -e \cdot x_2$.[1]

Note that the above guard expression declares that a bounce happens when the ball touches the ground and its velocity x_2 is non-positive, meaning either it is still or it is moving towards the ground. However, further analysis of the model reveals that when the following condition holds, $x_1 = 0 \wedge x_2 = 0$, meaning that the ball is at reset on the ground, the supporting force from the ground cancels out the gravity force. Therefore, the acceleration of the ball should be 0 rather than

[1] Identity reset maps, such as $x_1 := x_1$, are not explicitly shown.

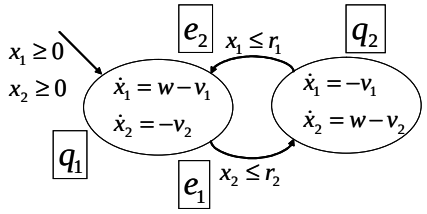

Figure 6.3. A hybrid system model of a water tank system.

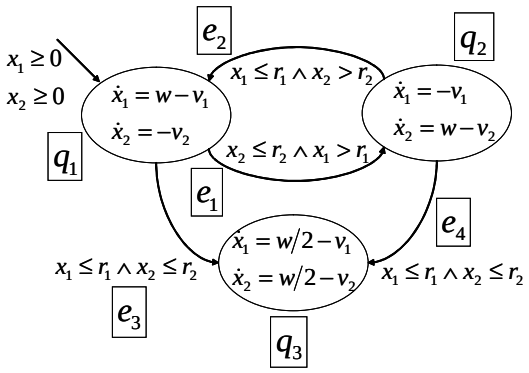

Figure 6.4. A more complete hybrid system model of the water tank system.

the acceleration of gravity. Under this circumstance, the ball in fact has a rather different dynamics given by $\dot{x_1} = x_2,\ \dot{x_2} = 0$.

This suggests that new dynamics might be necessary to describe the model's behavior. Consequently, the complete description of the dynamics of the bouncing ball system should include both an extra state associated with the new dynamics and a transition that drives the model into that state. One design of such hybrid system models is shown in Figure 6.2, where q_2 and e_2 are the new state and transition.

6.1.2 Water Tank

The second model that we will consider is the water tank system consisting of two tanks. We use x_1 and x_2 for the water levels, r_1 and r_2 for the critical water level thresholds, and v_1 and v_2 for the constant flow of water out of the tanks. There is a constant input flow of water w, which goes through a pipe and into either tank at any particular time point. We assume that $(v_1 + v_2) > w$, meaning that the sum of the output flow in both tanks is greater than the input flow. Therefore, the water levels of both tanks keep dropping. If the water level of any tank drops below its critical threshold, the input water gets delivered into that tank. The process of switching the pipe from one tank to the other takes zero time.

One hybrid system model that describes such system is shown in Fig. 6.3. This model has two states q_1 and q_2 corresponding to the different dynamics of the system when the input water flows into either of the two tanks. Transitions e_1 and e_2 specify switching conditions between states.

Note that the guard expressions between those two states are not mutually exclusive, meaning

that the guards $x_1 \leq r_1$ and $x_2 \leq r_2$ may be enabled at the same time. A trivial example will be that the two tanks have initial water levels $x_1 = r_1$ and $x_2 = r_2$. If the two tanks have initial water levels $x_1 > r_1$ and $x_2 > r_2$, then the water levels of both tanks will drop and the water pipe will switch between the two tanks. As more and more water flows out of tanks, we will see that the frequency of the pipe switching becomes higher and higher. In the limit, when this frequency reaches infinity, both guards become enabled at the same time.

When both guards are enabled, the water tank system will have a different dynamics. Recall the assumption that the switching speed of the water pipe is infinitely fast, the pipe should inject water into both tanks at the same time. In other words, there are virtually two *identical* pipes injecting water into both tanks. Also note that the input water flow is a constant and the pipe cannot hold water, therefore one possible scenario will be that each tank gets *half* of the input water. Therefore, at this time point, the whole system will have a rather different dynamics given by

$$\dot{x_1} = w/2 - v_1, \quad \dot{x_2} = w/2 - v_2. \tag{6.1}$$

We introduce a new state associated with the above dynamics and complete the transitions going from the existing states to the newly added state. The new design of the complete hybrid system model for the water tank system is shown in Figure 6.4, where q_3 is the new state, e_3 and e_4 are the newly added transitions. Note that for simplicity we allow the water levels to have negative values. Otherwise, we will need some other discrete states to show that once a tank is empty, it is always empty.

The hybrid system model in Figure 6.4 is similar to the *temporal* regularization results proposed in [55]. One of the key differences is that the temporal regularization solution requires the process of switching pipe to take some positive time ε. The amount of this ε affects the resulting behaviors. In fact, when ε goes to 0, the temporal regularization result is the same as what we have derived in (6.1).

In the next section, we will propose a systematical way to complete the specification of hybrid system models. In particular, we will discuss how to introduce new states called *post-Zeno* states, to modify the existing transitions, and to construct new transitions to these new states for model behaviors before and after potential Zeno time points. In Section 6.3, we will develop a practical and feasible simulation algorithm to approximate the exact behaviors of Zeno hybrid system models.

6.2 Completing Hybrid System Models

The purpose of this section is to introduce an algorithm for completing hybrid system models with the goal of carrying executions past the Zeno time point. This algorithm can be thought of as a combination of the currently known conditions for the existence (or nonexistence) of Zeno behavior in hybrid systems. Of course, the characterization of Zeno behavior in the literature is by no means complete, so we cannot claim that the procedure outlined here is the only way to complete a hybrid system, nor that the resulting hybrid system is the canonical completed hybrid system. We only claim that, given the current understanding of Zeno behavior, this method provides a reasonably satisfying method for completing hybrid systems. We dedicate the latter half of this section to examples, where we carry out the completion process.

6.2.1 Hybrid System Completion

Define a *hybrid system* as a tuple,

$$\mathcal{H} = (\Gamma, D, G, R, F),$$

where

- $\Gamma = (Q, E)$ is a finite directed graph, where Q represents the set of discrete states and E represents the set of edges connecting these states. There are two maps $\mathfrak{s} : E \to Q$ and $\mathfrak{t} : E \to Q$, which are the source and target maps respectively. That is $\mathfrak{s}(e)$ is the source of the edge e and $\mathfrak{t}(e)$ is its target.
- $D = \{D_q \subseteq \mathcal{R}^n_\infty \mid q \in Q\}$ is a set of *domains*, one for each state $q \in Q$. While the hybrid system is in state q, the dynamics of the hybrid system is a trajectory in D_q. Note that $\mathcal{R}^n_\infty = \mathcal{R}^n \cup \{\infty\}$ and $\mathcal{R}^n_\infty$ is compact.
- $G = \{G_e \subseteq D_{\mathfrak{s}(e)} \mid e \in E\}$ is a set of *guards*, where G_e is a set associated with the edge e and determines the switching behavior of the hybrid system at state $\mathfrak{s}(e)$. When the trajectory intersects with the guard set G_e, a transition is triggered and the discrete state of the hybrid system changes to $\mathfrak{t}(e)$. $G_q = \bigcup_{\mathfrak{s}(e)=q} G_e$ is the union of the guards associated with the outgoing edges from the same state q. We assume that G_q is closed, i.e., that every Cauchy sequence converges to an element in G_q.

- $R = \{R_e : G_e \to D_{\mathfrak{t}(e)} \mid e \in E\}$ is a set of *reset maps*. We write the image of R_e as $R_e(G_e) \subseteq D_{\mathfrak{t}(e)}$. These reset maps specify the initial continuous states of trajectories in the target discrete states.

- $F = \{f_q : D_q \to \mathcal{R}^n_\infty \mid q \in Q\}$ is a set of *vector fields*, which specify the dynamics of the hybrid system when it is in a discrete state q. We assume f_q is Lipschitz when restricted to D_q.

We will not explicitly define hybrid system behavior and Zeno behavior, as these definitions are well-known and can be found in a number of references [95, 4, 55, 70].

The goal of this section is to complete a hybrid system $\mathcal{H}$, i.e., we want to form a new hybrid system $\overline{\mathcal{H}}$ in which executions are carried beyond the Zeno time point. We begin by constructing this system theoretically and then discuss how to implement it practically. The theoretical completion of a hybrid system is carried out using the following process:

1. Augment the graph Γ of $\mathcal{H}$, based on the existence of *higher order cycles*, to include *post-Zeno* states, and edges to these post-Zeno states.

2. Specify the domains of the post-Zeno states.

3. Specify the guards on the edges to the post-Zeno states.

4. Specify the reset maps on the edges to the post-Zeno states.

5. Specify the vector fields on the post-Zeno states, based on the vector fields on the pre-Zeno states.

6. Check the existence of higher order cycles again on the completed hybrid systems. If there is any, go the step 1 and repeat the completing process. Otherwise, the process terminates.

Before carrying out this process, it is necessary to introduce the notion of *higher order cycles* in Γ. We call a finite string consisting of alternating states and edges in Γ a *finite path*,

$$q_1 \xrightarrow{e_1} q_2 \xrightarrow{e_2} q_3 \xrightarrow{e_3} \cdots \xrightarrow{e_{k-1}} q_k,$$

with $e_i \in E$ and $q_i \in Q$, s.t., $s(e_i) = q_i$ and $t(e_i) = q_{i+1}$. We denote such a path by $\langle q_1;\ e_1, e_2, \ldots, e_{k-1};\ q_k \rangle$.

For simplicity, we only consider paths with distinct edges. We could have considered paths with repeated edges, but that will result in an unbounded number of paths, each of which is arbitrarily

long. This makes the problem intractable. The number and length of paths with distinct edges are finite. In the worst case scenario, suppose there are N edges in the graph, the number of paths is $\sum_{m=1}^{N}(P_m^N)$, where P_m^N gives the number of permutations of choosing m edges from a set of N edges.

Although we only consider paths with distinct edges, we do not require a path to contain distinct states. In particular, if the starting state is the same as the ending state, such as $\langle q_1;\ e_1, e_2, \ldots, e_{k-1};\ q_1\rangle$, we call such a path a *finite cyclic path*. The set of all finite cyclic paths is called the *higher order cycles* in Γ and denoted by C. Formally,

$$C = \{\langle q;\ e_1, e_2, \ldots, e_{k-1};\ q\rangle \mid \forall i, j,\ i \neq j \Rightarrow e_i \neq e_j,\ e_i, e_j \in E,\ q \in Q\}. \tag{6.2}$$

To ease future discussion, we define $c_0 = \langle q;\ e_1, e_2, \ldots, e_{k-1};\ q\rangle \in C$. We will use this example cyclic path through the rest of this section.

We also define two operators, π_Q and π_E, on a cyclic path $c \in C$, where $\pi_Q(c)$ gives the starting and ending state of the path and $\pi_E(c)$ gives the first edge appearing in the path. When applied to the path c_0, $\pi_Q(c_0) = q$ and $\pi_E(c_0) = e_1$.

Consider any two edges $e_i, e_j \in E$ appearing a cyclic path such as c_0, where e_j immediately follows e_i, meaning $\mathfrak{s}(e_j) = \mathfrak{t}(e_i)$. We define a *restricted reset map* for the first edge e_i, $W_{e_i} : G_{e_i} \to G_{e_j}$ by

$$W_{e_i}(G_{e_i}) = R_{e_i}(G_{e_i}) \cap G_{e_j},$$

where G_{e_i} and G_{e_j} are guard sets associated with edges e_i and e_j respectively. The word "restricted" means that the image of the function W_{e_i} is constrained by the guard set of the destination state G_{e_j}.

For the last edge e_{k-1} appearing in the path c_0, we define the function $W_{e_{k-1}}$ by

$$W_{e_{k-1}}(G_{e_{k-1}}) = R_{e_{k-1}}(G_{e_{k-1}}) \cap G_{e_1},$$

where G_{e_1} is the guard set associated with the first edge e_1 appearing in the path c_0.

Then for a cyclic path $c \in C$, we define a function $R_c : G_{\pi_E(c)} \to G_{\pi_E(c)}$, where $\pi_E(c)$ is the first edge appearing in the path c. The function R_c is the composition of the reset maps along the path c, called the *reset map along path c*. Choosing the path c_0 as an example, we define the function $R_{c_0} : G_{e_1} \to G_{e_1}$ by

$$R_{c_0} = R_{\langle q;\ e_1, e_2, \ldots, e_{k-1};\ q\rangle} = W_{e_{k-1}} \circ W_{e_{k-2}} \circ \cdots \circ W_{e_2} \circ W_{e_1}. \tag{6.3}$$

We write the image of R_c as $R_c(G_{\pi_E(c)})$. From the definition of R_c, we know $R_c(G_{\pi_E(c)})$ must be a subset of $G_{\pi_E(c)}$ because it is resulted from intersection of sets. If we apply R_c again to $R_c(G_{\pi_E(c)})$, let $R_c^2(G_{\pi_E(c)})$ denote $R_c(R_c(G_{\pi_E(c)}))$, we have $R_c^2(G_{\pi_E(c)}) \subseteq R_c(G_{\pi_E(c)})$. Keeping applying the function R_c will result in a sequence of sets,

$$G_{\pi_E(c)} \supseteq R_c(G_{\pi_E(c)}) \supseteq R_c^2(G_{\pi_E(c)}) \supseteq \cdots \supseteq R_c^k(G_{\pi_E(c)}) \supseteq \cdots .$$

A question naturally arises: does the sequence always converge?

Let $\wp(X)$ be the power set of X. It is well known that for any set X, the ordered set $(\wp(X);\ \subseteq)$ with a set inclusion order is a complete lattice, where the *least upper bound (LUB)* is given by the union of subsets and the *greatest lower bound (GLB)* by the intersection of subsets. Note that the ordered set $(\wp(X);\ \supseteq)$ with a *reverse* set inclusion order is also a complete lattice, where the LUB is given by the intersection and the GLB by the union of subsets. By this definition, $(\wp(\mathcal{R}_\infty^n);\ \supseteq)$ is a complete lattice.

A complete lattice is a CPO. Therefore, $(\wp(\mathcal{R}_\infty^n);\ \supseteq)$ is a CPO. A CPO has a bottom element. For $(\wp(\mathcal{R}_\infty^n);\ \supseteq)$, the bottom element is $\mathcal{R}_\infty^n$.

Every chain in a CPO converges, therefore the sequence of sets,

$$G_{\pi_E(c)} \supseteq R_c(G_{\pi_E(c)}) \supseteq R_c^2(G_{\pi_E(c)}) \supseteq \cdots \supseteq R_c^k(G_{\pi_E(c)}) \supseteq \cdots ,$$

which forms a chain, always converges.

Now our question is whether we can constructively find the set that this sequence converges to. We will answer this question by applying Theroem (1).

We first define a function for a cyclic path c, $\mathbf{F}_c : \wp(\mathcal{R}_\infty^n) \to \wp(\mathcal{R}_\infty^n)$, by

$$\mathbf{F}_c(S) = R_c(S \cap G_{\pi_E(c)}),\ \text{where } S \in \wp(\mathcal{R}_\infty^n). \tag{6.4}$$

Theorem 3 (Monotonicity) *Function* $\mathbf{F}_c$ *is monotonic.*

Proof: Recall that $\mathbf{F}_c$ is monotonic if

$$\forall\, S_1, S_2 \in \wp(\mathcal{R}_\infty^n), S_1 \supseteq S_2 \Rightarrow \mathbf{F}_c(S_1) \supseteq \mathbf{F}_c(S_2).$$

Let $e = \pi_E(c)$. Suppose $S_1 \supseteq S_2$, then $G_e \cap S_1 \supseteq G_e \cap S_2$, and $R_e(G_e \cap S_1) \supseteq R_e(G_e \cap S_2)$. We have $W_e(G_e \cap S_1) \supseteq W_e(G_e \cap S_2)$, and therefore $R_c(G_e \cap S_1) \supseteq R_c(G_e \cap S_2)$, which gives $\mathbf{F}_c(S_1) \supseteq \mathbf{F}_c(S_2)$. □

In fact, a more general conclusion is that function $\mathbf{F}_c$ is a monotonic function as long as function R_c is a total function.

Theorem 4 (Scott Continuity) *Function* $\mathbf{F}_c$ *is continuous.*

Proof: Recall that $\mathbf{F}_c$ is continuous if for any chain $L \subseteq \wp(\mathcal{R}_\infty^n)$, where $L = \{S_1, S_2, S_3, \cdots\}$ and $S_1 \supseteq S_2 \supseteq S_3 \supseteq \cdots$,

$$\mathbf{F}_c(\wedge L) = \wedge\hat{\mathbf{F}}_c(L) = \wedge\{\mathbf{F}_c(S_i) \mid S_i \in L\}.$$

Note that $\forall\, S_i, S_i \supseteq \wedge L$. Because $\mathbf{F}_c$ is monotonic, $\mathbf{F}_c(S_i) \supseteq \mathbf{F}_c(\wedge L)$. Therefore,

$$\mathbf{F}_c(\wedge L) \subseteq \wedge\hat{\mathbf{F}}_c(L).$$

Now we want to show

$$\wedge\hat{\mathbf{F}}_c(L) \subseteq \mathbf{F}_c(\wedge L).$$

For any $x \in \wedge\hat{\mathbf{F}}_c(L)$, there must exist $y \in S_i \cap G_{\pi_E(c)}$ for all S_i, s.t., $x = R_c(y)$. Also note that $\mathbf{F}_c(\wedge L) = R_c(\bigcap_i S_i \cap G_{\pi_E(c)})$ and $y \in \bigcap_i S_i \cap G_{\pi_E(c)}$, therefore $x \in \mathbf{F}_c(\wedge L)$. In other words, $\wedge\hat{\mathbf{F}}_c(L) \subseteq \mathbf{F}_c(\wedge L)$.

In the end,

$$\wedge\hat{\mathbf{F}}_c(L) \subseteq \mathbf{F}_c(\wedge L) \text{ and } \mathbf{F}_c(\wedge L) \subseteq \wedge\hat{\mathbf{F}}_c(L) \Rightarrow \mathbf{F}_c(\wedge L) = \wedge\hat{\mathbf{F}}_c(L).$$

□

Alternatively, we can prove theorem (4) according to 8.8(2) in [31] (page 178). In particular, because $(\wp(X);\ \supseteq)$ satisfies the *ascending chain condition* (ACC) defined in 2.37(v) in [31] (page 51), and function $\mathbf{F}_c$ is monotonic according to theorem (3), $\mathbf{F}_c$ is a continuous function.

By applying The important result is that the sequence of sets,

$$G_{\pi_E(c)} \supseteq R_c(G_{\pi_E(c)}) \supseteq R_c^2(G_{\pi_E(c)}) \supseteq \cdots \supseteq R_c^k(G_{\pi_E(c)}) \supseteq \cdots$$

converges to a set, which is a least fixed point of $\mathbf{F}_c$.

Theroem (1) also gives us a constructive procedure to find such a fixed point. That is, start by evaluating $\mathbf{F}_c$ with $\mathcal{R}_\infty^n$, and then keep applying $\mathbf{F}_c$ on the evaluated results until they converge to a fixed point.

However we cannot guarantee that the fixed point can be computed with the above procedure in a *finite* number of steps of computation. In fact, example 6.2.2 in the next section will show that

finding the fixed point may take an infinite number of computation steps. In this case, our algorithm breaks down and this is one of the limitations.

We denote such a fixed point as Z_c, called *Zeno set* of cyclic path c, where

$$Z_c = \mathbf{F}_c(Z_c). \tag{6.5}$$

Equation (6.5) states that if a trajectory intersects with the guard set $G_{\pi_E(c)}$ at an element $z \in Z_c \subseteq G_{\pi_E(c)}$, then after a series of reset maps, R_c, the initial continuous state of the new trajectory z' is again in Zeno set Z_c, which is a subset of $G_{\pi_E(c)}$. Note that the new initial state z' does not have to be the same as the starting state z (c.f. [95]). However, the new state z' will nevertheless trigger the first transition appearing first in the path to be taken anyway. With the constraint that transitions happen instantaneously, there will be an infinite number of transitions happening at the same time point. Therefore, the existence of a nonempty Zeno set Z_c indicates the possible existence of Zeno equilibria [95]. This motivates the construction of the completed hybrid system based on a subset of cyclic paths, $C' = \{c \in C \mid Z_c \neq \emptyset\}$.

For a hybrid system $\mathcal{H}$, define the corresponding *completed hybrid system* $\overline{\mathcal{H}}$ by

$$\overline{\mathcal{H}} = (\overline{\Gamma}, \overline{D}, \overline{G}, \overline{R}, \overline{F}),$$

where

1. $\overline{\Gamma} = (\overline{Q}, \overline{E})$, where $\overline{\Gamma}$ has more discrete states and edges than Γ. The set of extra states is $Q' = \overline{Q} \setminus Q$, where Q' is called the set of *post-Zeno states*. The set of extra edges is $E' = \overline{E} \setminus E$. We pick the extra states and edges to be in bijective correspondence with Q', i.e., there exist bijections $g : Q' \to C'$ and $h : E' \to C'$. Consequently, $\forall c \in C'$, there always exist a unique $q \in Q'$ and a unique $e \in E'$.

   We define the source and target maps, $\bar{\mathfrak{s}} : \overline{E} \to \overline{Q}$ and $\bar{\mathfrak{t}} : \overline{E} \to \overline{Q}$ for $e \in \overline{E}$ by

$$\bar{\mathfrak{s}}(e) = \begin{cases} \mathfrak{s}(e) & \text{if} \quad e \in E \\ \pi_Q(h(e)) & \text{if} \quad e \in E' \end{cases}, \quad \text{and} \quad \bar{\mathfrak{t}}(e) = \begin{cases} \mathfrak{t}(e) & \text{if} \quad e \in E \\ g^{-1}(h(e)) & \text{if} \quad e \in E' \end{cases}.$$

   Intuitively, for each cyclic path $c \in C'$ found in Γ, we can find a new discrete state $q = g^{-1}(c) \in Q'$ and a new edge $e = h^{-1}(c) \in E'$ that goes from $\pi_Q(c)$ to q in $\overline{\Gamma}$.

   The new discrete state q, which corresponds to the cyclic path $c = g(q)$, copies all edges of the original state $\pi_Q(c)$ except the first edge appearing in the path, $\pi_E(c)$. In fact, we can think that the newly introduced edge $e = h^{-1}(c)$ replaces the edge $\pi_E(c)$.

2. Define $\overline{D} = D \cup D'$, where D' is the set of domains of post-Zeno states, defined as $D' = \{D'_q \subseteq \mathcal{R}^n_\infty \mid q \in Q'\}$. For each $c \in C'$, D'_q is defined by

$$D'_q = Z_c, \quad \text{where} \quad q = g^{-1}(c) \in Q'. \tag{6.6}$$

Note that D'_q is not only the domain for post-Zeno state q but also the guard set that triggers the transition from the pre-Zeno state $\pi_Q(c)$ to the post-Zeno state q.

3. In order to define $\overline{G}$, we first modify the guard G_e in G by subtracting Z_c from G_e, where $c \in C'$ with $\pi_Q(c) = \bar{\mathfrak{s}}(e)$. Define, for all $e \in E$,

$$\widetilde{G_e} = G_e \setminus \bigcup_{c \in C' \text{ s.t. } \pi_Q(c) = \bar{\mathfrak{s}}(e)} Z_c, \tag{6.7}$$

and define, for all $e \in E'$,

$$G'_e = D'_q, \quad \text{where} \quad q = \bar{\mathfrak{t}}(e). \tag{6.8}$$

Then the complete definition of $\overline{G}$ is $\overline{G} = \{\widetilde{G_e} \mid e \in E\} \cup \{G'_e \mid e \in E'\}$.

4. $\overline{R} = \{R_e : \widetilde{G_e} \to D_{\bar{\mathfrak{t}}(e)} \mid e \in E\} \cup \{R'_e : G'_e \to D_{\bar{\mathfrak{t}}(e)} \mid e \in E'\}$, where the reset map R'_e is the identity map.

5. $\overline{F} = F \cup \{f'_q : D'_q \to \mathcal{R}^n_\infty \mid q \in Q'\}$, where f'_q is the vector field on D'_q. This vector field may be application-dependent, but in some circumstances, it can be obtained from the vector field $f_{q'}$ on $D_{q'}$, where $q' = \pi_Q(g(q)) \in Q$.

6. Check the existence of higher order cycles again on the completed hybrid systems. If there is any, go the step 1 and repeat the completing process. Otherwise, the process terminates. This step is necessary for completing hybrid systems with a state having multiple outgoing edges pointing back to itself.

Upon inspection of the definition of the completed hybrid system, it is evident that we have explicitly given a method for computing every part of this system except for the vector fields on the post-Zeno states. We do not claim to have an explicit method for generally computing f'_q, because this would depend on the constraints imposed by D'_q which we do not assume are of any specific form.

However, in some special cases, it is possible to find such a vector field [8]. This special class of hybrid systems are called *Lagrangian* hybrid systems. The dynamics of a Lagrangian hybrid system before and after the Zeno time point are derived under different constraints.

In the next subsection, we will demonstrate how to carry out the process of completing hybrid systems by revisiting the examples discussed in Section 6.1.

6.2.2 Examples Revisited

Example 1: Bouncing Ball.

We first revisit the bouncing ball example shown in Fig. 6.1. Write this example hybrid system as a tuple, $\mathcal{H} = ((Q, E), D, G, R, F)$. We have the discrete state set $Q = \{q_1\}$, the edge set $E = \{e_1\}$, the set of guards $G = \{G_{e_1}\}$, where $G_{e_1} = \{(x_1, x_2) \in \mathcal{R}^2_\infty \mid x_1 = 0 \wedge x_2 \leq 0\}$, and the set of the reset maps $R = \{R_{e_1}\}$, where R_{e_1} is defined by $R_{e_1}(x_1, x_2) = (x_1, -e \cdot x_2)$, $\forall (x_1, x_2) \in G_{e_1}$.

There is only one element, $c = \langle q_1;\ e_1;\ q_1 \rangle$, in the set C of cyclic paths. For path c, the composition of reset maps along c is $R_c = W_{e_1}$ and $\pi_E(c) = e_1$. Evaluating equation (6.5) with Z_c as the guard G_{e_1}, we get

$$\begin{aligned}
Z'_c &= \mathbf{F}_c(\mathcal{R}^n_\infty) \\
&= R_c(\mathcal{R}^n_\infty \cap G_{e_1}) \\
&= W_{e_1}(G_{e_1}) \\
&= R_{e_1}(G_{e_1}) \cap G_{e_1} \\
&= \{(x_1, x_2) \mid x_1 = 0 \wedge x_2 \geq 0\} \cap \{(x_1, x_2) \mid x_1 = 0 \wedge x_2 \leq 0\} \\
&= \{(x_1, x_2) \mid x_1 = 0 \wedge x_2 = 0\} \\
&= \{(0, 0)\}.
\end{aligned}$$

Keep evaluating equation (6.5) with Z'_c, and we get

$$\begin{aligned}
Z''_c &= \mathbf{F}_c(Z'_c) \\
&= R_c(Z'_c \cap G_{e_1}) \\
&= W_{e_1}(Z'_c) \\
&= R_{e_1}(Z'_c) \cap Z'_c \\
&= \{(0, 0)\} \cap \{(0, 0)\} \\
&= \{(0, 0)\}.
\end{aligned}$$

Since Z''_c is the same as Z'_c, we have got a fixed point of $\mathbf{F}_c$, denoted as $Z_c = \{(0, 0)\}$. Since Z_c is nonempty, we introduce a new state q_2 and a new edge e_2 such that $\overline{Q} = \{q_1, q_2\}$ and $\overline{E} = \{e_1, e_2\}$.

The source and target maps are

$$\bar{\mathfrak{s}}(e) = \quad q_1 \quad , \quad \forall e \in \overline{E} \ , \qquad \text{and} \qquad \bar{\mathfrak{t}}(e) = \begin{cases} q_1 & \text{if} \quad e = e_1 \\ q_2 & \text{if} \quad e = e_2 \end{cases} \quad .$$

Because there is only one outgoing edge e_1 from the state q_1 and this edge appears in the cyclic path c, the new state q_2 copies no edge from q_1.

The domain for discrete state q_2 is $D'_{q_2} = Z_c$. Then $\overline{D} = D \cup \{D'_{q_2}\}$. Since the set D'_{q_2} only contains one element, the dynamics (vector fields) of the hybrid system is trivial, where $\dot{x_1}(t) = 0$, $\dot{x_2}(t) = 0$. This simply means that the ball cannot move at all, which is exactly the same as what we got in the introduction.

We must point out that the domain for a post-Zeno state may contain more than one element. In this case, the dynamics in general cannot be computed without a model designer's expertise. However, in some special cases such as mechanical systems, the vector fields describe the equations of motion for these systems. If in addition, the guards are derived from unilateral constraints on the configuration space, then the vector fields on the post-Zeno states can be obtained from the vector fields on the pre-Zeno states via holonomic constraints. In fact, the vector fields on the post-Zeno state of the above example can be obtained from a *hybrid Lagrangian* [8]. A detailed explanation of the process for computing vector fields and more examples can be found in [8].

Note that D'_{q_2} is also the guard set of e_2 that specifies the switching condition from q_1 to q_2, meaning $G'_{e_2} = \{(x_1, x_2) \in \mathcal{R}^2_\infty \mid x_1 = 0 \wedge x_2 = 0\}$. Following (6.7), we get a modified $\widetilde{G_{e_1}} = \{(x_1, x_2) \in \mathcal{R}^2_\infty \mid x_1 = 0 \wedge x_2 < 0\}$. The set of these two guard sets gives $\overline{G} = \{\widetilde{G_{e_1}}, G'_{e_2}\}$.

Finally, $\overline{R} = \{R_{e_1}, R'_{e_2}\}$, where R'_{e_2} is just the identity map.

Now we need to check the completed hybrid system again for possible non-empty Zeno sets. There is still only one cyclic path c. However by repeating the above procedure, we get $Z_c = \emptyset$. The detailed derivation process is the same as what described the above and is omitted here. Therefore the completing process is finished.

In summary we get the completed hybrid system $\overline{\mathcal{H}} = ((\overline{Q}, \overline{E}), \overline{D}, \overline{G}, \overline{R}, \overline{F})$, which is the same as the model shown in Figure 6.2.

Example 2: Water Tank.

Now let us revisit the water tank example shown in Fig. 6.3. Write this example hybrid system as a tuple, $\mathcal{H} = ((Q, E), D, G, R, F)$. We have the discrete state set $Q = \{q_1, q_2\}$, the edge set $E = \{e_1, e_2\}$, the set of guards $G = \{G_{e_1}, G_{e_2}\}$, where $G_{e_1} = \{(x_1, x_2) \in \mathcal{R}^2_\infty \mid x_2 \leq r_2\}$ and $G_{e_2} = \{(x_1, x_2) \in \mathcal{R}^2_\infty \mid x_1 \leq r_1\}$, and the set of the reset maps $R = \{R_{e_1}, R_{e_2}\}$, where both reset maps are identity maps.

There are two elements, $c_1 = \langle q_1;\ e_1, e_2;\ q_1\rangle$ and $c_2 = \langle q_2;\ e_2, e_1;\ q_2\rangle$, in the set C that contains cyclic paths. For path c_1, the composition of reset maps along c_1 is $R_{c_1} = W_{e_2} \circ W_{e_1}$ and $\pi_E(c_1) = e_1$.

We first compute the fixed point Z_{c_1} for $\mathbf{F}_{c_1}$ by evaluating (6.5) with Z_{c_1} as the guard G_{e_1}, we get

$$\begin{aligned}
Z'_{c_1} &= \mathbf{F}_{c_1}(\mathcal{R}^n_\infty) \\
&= R_{c_1}(\mathcal{R}^n_\infty \cap G_{e_1}) \\
&= (W_{e_2} \circ W_{e_1})(G_{e_1}) \\
&= W_{e_2}(W_{e_1}(G_{e_1})) \\
&= W_{e_2}(R_{e_1}(G_{e_1}) \cap G_{e_2}) \\
&= W_{e_2}(G_{e_1} \cap G_{e_2}) \\
&= R_{e_2}(G_{e_1} \cap G_{e_2}) \cap G_{e_1} \\
&= G_{e_1} \cap G_{e_2} \cap G_{e_1} \\
&= G_{e_1} \cap G_{e_2} \\
&= \{(x_1, x_2) \mid x_2 \leq r_2\} \cap \{(x_1, x_2) \mid x_1 \leq r_1\} \\
&= \{(x_1, x_2) \mid x_1 \leq r_1 \wedge x_2 \leq r_2\}.
\end{aligned}$$

Keep evaluating equation (6.5) with Z'_{c_1}, we get

$$\begin{aligned}
Z''_{c_1} &= \mathbf{F}_{c_1}(Z'_{c_1}) \\
&= R_{c_1}(Z'_{c_1} \cap G_{e_1}) \\
&= (W_{e_2} \circ W_{e_1})(Z'_c) \\
&= \cdots \quad \text{(steps omitted)} \\
&= \{(x_1, x_2) \mid x_1 \leq r_1 \wedge x_2 \leq r_2\}. \qquad (6.9)
\end{aligned}$$

Since Z''_{c_1} is the same as Z'_{c_1}, we have got a fixed point of $\mathbf{F}_{c_1}$, denoted as $Z_{c_1} = \{(x_1, x_2) \mid x_1 \leq$

$r_1 \wedge x_2 \leq r_2\}$. Similarly, for path c_2, we get $Z_{c_2} = \{(x_1, x_2) \mid x_1 \leq r_1 \wedge x_2 \leq r_2\}$, which is the same as Z_{c_1}.

Since both Z_{c_1} and Z_{c_2} are nonempty, we introduce two new states q_3 and q_4 and two new edges e_3 and e_4 such that $\overline{Q} = \{q_1, q_2, q_3, q_4\}$ and $\overline{E} = \{e_1, e_2, e_3, e_4\}$. The source and target maps are

$$\bar{\mathfrak{s}}(e) = \begin{cases} q_1 & \text{if} \quad e = e_1 \vee e = e_3 \\ q_2 & \text{if} \quad e = e_2 \vee e = e_4 \end{cases}, \quad \text{and} \quad \bar{\mathfrak{t}}(e) = \begin{cases} q_2 & \text{if} \quad e = e_1 \\ q_1 & \text{if} \quad e = e_2 \\ q_3 & \text{if} \quad e = e_3 \\ q_4 & \text{if} \quad e = e_4 \end{cases}.$$

Because there is only one outgoing edge e_1 from the state q_1 and this edge appears in the cyclic path c, the new state q_3 copies no edge from q_1. Similarly, the new state q_4 does not copy edges from q_2.

The domain for discrete state q_3 is $D'_{q_3} = Z_{c_1}$, and the domain for discrete state q_4 is $D'_{q_4} = Z_{c_2}$. Then $\overline{D} = D \cup \{D'_{q_3}, D'_{q_4}\}$.

As we pointed out earlier in the previous example, in order to derive the dynamics for post-Zeno states, a careful analysis has to be performed by model designers, and the resulting dynamics may not be unique. For example, one might think that 3/4 of the input flow goes into the first tank and the rest goes into the second tank. This dynamics is different from what we had in the introduction chapter. We do not (in fact, we cannot) determine which result is better.

Note that D'_{q_3} is also the guard set of e_3 that specifies the switching condition from q_1 to q_3, meaning $G'_{e_3} = \{(x_1, x_2) \in \mathcal{R}^2_\infty \mid x_1 \leq r_1 \wedge x_2 \leq r_2\}$. Following (6.7), we get a modified $\widetilde{G_{e_1}} = \{(x_1, x_2) \in \mathcal{R}^2_\infty \mid x_1 \leq r_1 \wedge x_2 > r_2\}$. Similarly, we get $G'_{e_4} = \{(x_1, x_2) \in \mathcal{R}^2_\infty \mid x_1 \leq r_1 \wedge x_2 \leq r_2\}$, and a modified $\widetilde{G_{e_2}} = \{(x_1, x_2) \in \mathcal{R}^2_\infty \mid x_2 \leq r_2 \wedge x_1 > r_1\}$. The set of these two guard sets gives $\overline{G} = \{\widetilde{G_{e_1}}, \widetilde{G_{e_2}}, G'_{e_3}, G'_{e_4}\}$.

Finally, $\overline{R} = \{R_{e_1}, R_{e_2}, R'_{e_3}, R'_{e_4}\}$, where all reset maps are identity maps.

Now we need to check the completed hybrid system again for possible non-empty Zeno sets. There are still only two cyclic paths c_1 and c_2. However by repeating the above procedure, we get $Z_{c_1} = Z_{c_2} = \emptyset$. The detailed derivation process is the same as described the above and is omitted here. Therefore the completing process is finished.

In summary we get the completed hybrid system $\overline{\mathcal{H}} = ((\overline{Q}, \overline{E}), \overline{D}, \overline{G}, \overline{R}, \overline{F})$, which is slightly different from the model shown in Figure 6.4 in that $\overline{\mathcal{H}}$ contains 4 discrete states. However, if we

choose the same dynamics such as (6.1) for discrete states q_3 and q_4, then q_3 and q_4 are the same. Thus we get a model with the same dynamics as that of the model in Fig. 6.4.

Example 3: A Non-Zeno Hybrid System.

Now let us consider a hybrid system simpler than the bouncing ball example shown in Figure 6.1. Similar to the bouncing ball example, this hybrid system has a single state, $Q = \{q_1\}$, and a single edge, $E = \{e_1\}$. The set of guards is $G = \{G_{e_1}\}$, where $G_{e_1} = \{x \in R \mid 0 \leq x \leq 10\}$, and the set of the reset maps is $R = \{R_{e_1}\}$, where R_{e_1} is defined by $R_{e_1}(x) = x + 1$, $\forall x \in G_{e_1}$.

There is only one element, $c = \langle q_1;\ e_1;\ q_1 \rangle$, in the set C of cyclic paths. For path c, the composition of reset maps along c is $R_c = W_{e_1}$ and $\pi_E(c) = e_1$. Evaluating equation (6.5) with Z_c as the guard G_{e_1}, we get

$$\begin{aligned} Z'_c &= \mathbf{F}_c(\mathcal{R}^n_\infty) \\ &= R_c(\mathcal{R}^n_\infty \cap G_{e_1}) \\ &= W_{e_1}(G_{e_1}) \\ &= R_{e_1}(G_{e_1}) \cap G_{e_1} \\ &= \{x \in R \mid 1 \leq x \leq 11\} \cap \{x \in R \mid 0 \leq x \leq 10\} \\ &= \{x \in R \mid 1 \leq x \leq 10\}. \end{aligned}$$

Keep evaluating equation (6.5) with Z'_c, and we get

$$\begin{aligned} Z''_c &= \mathbf{F}_c(Z'_c) \\ &= R_c(Z'_c \cap G_{e_1}) \\ &= W_{e_1}(Z'_c) \\ &= R_{e_1}(Z'_c) \cap Z'_c \\ &= \{x \in R \mid 2 \leq x \leq 10\}. \end{aligned}$$

Keep evaluating equation (6.5) with the new results for 10 times, and we will get

$$\begin{aligned} Z_c^{11} &= \mathbf{F}_c(Z_c^{10}) \\ &= R_c(Z_c^{10} \cap G_{e_1}) \\ &= W_{e_1}(Z_c^{10}) \\ &= R_{e_1}(Z_c^{10}) \cap Z_c^{10} \\ &= R_{e_1}(\{x \in R \mid x = 10\}) \cap \{x \in R \mid x = 10\} \\ &= \{x \in R \mid x = 11\} \cap \{x \in R \mid x = 10\} \\ &= \emptyset, \end{aligned}$$

where Z_c^n indicates the n-th result from evaluating equation (6.5).

The result set Z_c^{11} is an empty set, any set intersecting with an empty set will result in an empty set. The empty set is the least fixed point of $\mathbf{F}_c$. Because this fixed point is empty, there is no Zeno time point. Therefore, we conclude this model is non-Zeno and it is already complete.

Example 4: A Zeno Hybrid System.

In this example, we use the same model in Example 6.2.2 but with a different reset map, $R = \{R_{e_1}\}$, where R_{e_1} is defined by $R_{e_1}(x) = x/2$, $\forall x \in G_{e_1}$.

There is only one element, $c = \langle q_1;\ e_1;\ q_1 \rangle$, in the set C of cyclic paths. For path c, the composition of reset maps along c is $R_c = W_{e_1}$ and $\pi_E(c) = e_1$. Evaluating equation (6.5) with Z_c as the guard G_{e_1}, we get

$$\begin{aligned} Z_c' &= \mathbf{F}_c(\mathcal{R}_\infty^n) \\ &= R_c(\mathcal{R}_\infty^n \cap G_{e_1}) \\ &= W_{e_1}(G_{e_1}) \\ &= R_{e_1}(G_{e_1}) \cap G_{e_1} \\ &= \{x \in R \mid 0 \le x \le 5\} \cap \{x \in R \mid 0 \le x \le 10\} \\ &= \{x \in R \mid 0 \le x \le 5\}. \end{aligned}$$

Keep evaluating equation (6.5) with Z'_c, and we get

$$\begin{aligned} Z''_c &= \mathbf{F}_c(Z'_c) \\ &= R_c(Z'_c \cap G_{e_1}) \\ &= W_{e_1}(Z'_c) \\ &= R_{e_1}(Z'_c) \cap Z'_c \\ &= \{x \in R \mid 0 \leq x \leq 2.5\}. \end{aligned}$$

Keep evaluating equation (6.5) with the new results for n times, and we will get

$$\begin{aligned} Z_c^{n+1} &= \mathbf{F}_c(Z_c^n) \\ &= R_c(Z_c^n \cap G_{e_1}) \\ &= W_{e_1}(Z_c^n) \\ &= R_{e_1}(Z_c^n) \cap Z_c^n \\ &= R_{e_1}(\{x \in R \mid 0 \leq x \leq (1/2)^n\}) \cap \{x \in R \mid 0 \leq x \leq (1/2)^n\} \\ &= \{x \in R \mid 0 \leq x \leq (1/2)^{n+1}\} \cap \{x \in R \mid 0 \leq x \leq (1/2)^n\} \\ &= \{x \in R \mid 0 \leq x \leq (1/2)^{n+1}\}, \end{aligned}$$

where Z_c^n indicates the n-th result from evaluating equation (6.5).

It is obvious that the more times we evaluate the equation (6.5), the bigger n will be. However, there will not be a limit for n.

Analytically, we can conclude that the least fixed point of $\mathbf{F}_c$ is a non-empty set containing only one element, $\{0\}$. However, this result cannot be computed in a finite number steps. Therefore, our constructive procedure does not work for this kind of hybrid systems.

Example 5: Another Zeno Hybrid System.

Here is another Zeno hybrid system example, we use the same model in Example 6.2.2 but with a different guard set, $G_{e_1} = \{x \in R \mid x \geq 0\}$.

There is only one element, $c = \langle q_1;\ e_1;\ q_1 \rangle$, in the set C of cyclic paths. For path c, the composition of reset maps along c is $R_c = W_{e_1}$ and $\pi_E(c) = e_1$. Evaluating equation (6.5) with Z_c

as the guard G_{e_1}, we get

$$\begin{aligned} Z'_c &= \mathbf{F}_c(\mathcal{R}^n_\infty) \\ &= R_c(\mathcal{R}^n_\infty \cap G_{e_1}) \\ &= W_{e_1}(G_{e_1}) \\ &= R_{e_1}(G_{e_1}) \cap G_{e_1} \\ &= \{x \in R \mid x \geq 1\} \cap \{x \in R \mid x \geq 0\} \\ &= \{x \in R \mid x \geq 1\}. \end{aligned}$$

Keep evaluating equation (6.5) with Z'_c, we get

$$\begin{aligned} Z''_c &= \mathbf{F}_c(Z'_c) \\ &= R_c(Z'_c \cap G_{e_1}) \\ &= W_{e_1}(Z'_c) \\ &= R_{e_1}(Z'_c) \cap Z'_c \\ &= \{x \in R \mid x \geq 2\}. \end{aligned}$$

Keep evaluating equation (6.5) with the new results for n times, we will get

$$\begin{aligned} Z^{n+1}_c &= \mathbf{F}_c(Z^n_c) \\ &= R_c(Z^n_c \cap G_{e_1}) \\ &= W_{e_1}(Z^n_c) \\ &= R_{e_1}(Z^n_c) \cap Z^n_c \\ &= R_{e_1}(\{x \in R \mid x \geq n\}) \cap \{x \in R \mid x \geq n\} \\ &= \{x \in R \mid x \geq n+1\} \cap \{x \in R \mid x \geq n\} \\ &= \{x \in R \mid x \geq n+1\}, \end{aligned}$$

where Z^n_c indicates the n-th result from evaluating equation (6.5).

It is obvious that the more times we evaluate the equation (6.5), the bigger n will be. However, there will not be a limit for n.

Analytically, noting that $\mathcal{R}_\infty$ contains ∞, we can conclude that the least fixed point of $\mathbf{F}_c$ is a set containing only one element $\{\infty\}$. Therefore this system may have a Zeno behavior, we need to introduce a post-Zeno state for describing the dynamics after the Zeno time point. The completion

process is omitted in this report. However, we have to point out that this fixed point set cannot be computed in a finite number steps. Therefore, our constructive procedure does not work for this kind of hybrid systems.

6.3 Approximate Simulation

When simulating a Zeno hybrid system model, we meet more challenging practical issues. The first difficulty is that before the Zeno time point, there will be an infinite number of discrete transitions (events). A discrete phase of execution needs to be performed for each time point when a discrete event occurs, which takes a non-zero time. So it is impossible to handle all discrete transitions in a finite time interval. In other words, the simulation gets stuck near the Zeno time point. The second difficulty is caused by numerical errors, which make it impractical to get an exact simulation. We will first elaborate on the second issue, and then we will come back to the first issue in subsection 6.3.3.

6.3.1 Numerical Errors

There are two sources of numerical errors: round-off error and truncation error[2]. Round-off error arises from using a finite number of bits in a computer to represent a real value. We denote this kind of difference as η. Then we can say that each integration operation will incur a round-off error of order η, denoted as $O(\eta)$. Round-off error accumulates. Suppose we integrate with a fixed step-size solver with a integration step size as h. In order to simulate over a unit time interval, we need h^{-1} integration steps, then the total round-off error is $O(\eta/h)$. Clearly, the bigger the step size, the fewer integration steps, the smaller the total round-off error. Similar results can be drawn for variable step-size solvers.

Truncation error comes from the integration algorithms used by practical ODE solvers. For example, an nth-order explicit Runge-Kutta method, which is derived to match the first $n+1$ terms of Taylor's expansion, has a *local* truncation error of $O(h^{n+1})$ and an *accumulated* truncation error of $O(h^n)$. Note that both truncation errors decrease as h decreases. Ideally we will get no truncation errors as $h \to 0$.

The total numerical error ε for an ODE solver using an nth-order explicit Runge-Kutta method

[2]We will not give a thorough discussion of numerical errors, which have been extensively studied, e.g. in [24]. We would rather briefly review and explain the important trade-offs when choosing integration step sizes.

is the sum of the round-off error and truncation error,

$$\varepsilon \sim \eta/h + h^n. \tag{6.10}$$

We can see that with a big integration step size h, the total error is dominated by truncation error, whereas round-off error dominates with a small step size. Therefore, although it is desirable to choose a small step size to reduce truncation error, the accuracy of a calculation result may not be increased due to the accumulation of round-off error. If we take the derivative of (6.10) with respect to h, then we get that when $h \sim \eta^{1/(n+1)}$ the total error ε reaches its minimum $O(\eta^{n/(n+1)})$. Therefore, in practice, we need to set a lower bound for both the integration step size and error tolerance (or value resolution) of integration results. We denote them as h_0 and ε_0 respectively, where

$$h_0 \sim \eta^{1/(n+1)}, \quad \varepsilon_0 \sim \eta^{n/(n+1)}.$$

For a good simulation, accuracy is one concern and efficiency is another objective. Efficiency for numerical integration is usually measured in terms of computation time or the number of computing operations. Using a big integration step size is an effective way to improve efficiency but with the penalty of loss of accuracy. So there is a trade-off. Furthermore, step sizes have upper bounds that are enforced by the consistency, convergence, and stability requirements when deploying practical integration methods on concrete ODEs [24]. Therefore, most practical *adaptive* ODE solvers embed a mechanism inside the integration process to adjust the step size according to the changing speed (derivative) of integration results, so that efficiency gets improved while maintaining the required accuracy at the same time.

In summary, a practical ODE solver usually specifies a minimum integration step size h_0, some small error tolerance ε_0, and an algorithm to adapt step size to meet requirements on both efficiency and accuracy.

6.3.2 Computation Difficulties

It is well-known that numerical integration in general can only deliver an approximation to the exact solution of an initial value ODE. However, the distance of the approximation from the exact solution is controllable for certain kinds of vector fields. For example, if a vector field satisfies a Lipschitz condition along the time interval where it is defined, we can constrain the integration results to reside within a neighborhood of the exact solution by introducing more bits for representing values to get better precision and integrating with a small step size.

The same difficulties that arise in numerical integration also appear in event detection. A few algorithms have been developed to solve this problem [84, 79, 37]. However, there is still a fundamental unsolvable difficulty: we can only get the simulation time close to the time point where an event occurs, but we are not assured of being able to determine that point precisely.

Simulating a Zeno hybrid system poses another fundamental difficulty. We will first explain it through a simple continuous-time example with dynamics

$$\dot{x}(t) = 1/(t-1),\ x(0) = 0,\ t \in [0, 2]. \tag{6.11}$$

We can analytically find the solution for this example, $x(t) = \ln \mid t-1 \mid$. However, getting the same result through simulation is difficult. Suppose the simulation starts with $t = 0$. As t approaches 1, the derivative $\dot{x}(t)$ keeps decreasing without bound. To satisfy the convergence and stability requirements, the step size h has to be decreased. When the step size becomes smaller than h_0, round-off error is not neglectable any more and the simulation results become unreliable. Trying to reduce the step size further doesn't help, because the disturbance from round-off error will dominate.

A similar problem arises when simulating Zeno hybrid system models. Recall that Zeno executions have an infinite number of discrete events (transitions) before reaching the Zeno time point, and the time intervals between two consecutive transitions shrink to 0. When the time interval becomes less than h_0, round-off errors again dominate.

In summary, it is impractical to precisely simulate the behavior of a Zeno model. Therefore, similar to numerical integration, we need to develop a computationally feasible way to approximate the exact model behavior. The objective is to give a close approximation under the limits enforced by numerical errors. We will do this in the next subsection.

6.3.3 Approximating Zeno Behaviors

In Section 6.2, we have described how to specify the behaviors of a Zeno hybrid system before and after the Zeno time point and how to develop transitions from pre-Zeno states to post-Zeno states. The construction procedure works for guards which are arbitrary sets. However, assuming that each guard is the sub-levelset of a function (or collection of functions) simplifies the framework for studying transitions to post-Zeno states. Therefore, we assume that a transition going from a pre-Zeno state to a post-Zeno state has a guard expression of form,

$$G_{e_c} = \{x \in \mathcal{R}^n \mid g_{e_c}(x) \leq 0\}, \tag{6.12}$$

for every $c \in C$, where $e_c = h^{-1}(c)$ and $g_{e_c} : \mathcal{R}^n \to \mathcal{R}^k$. Furthermore, we assume that $g_{e_c}(x)$ is continuously differentiable.

In this section, we will develop an algorithm such that the complete model behavior can be simulated. As the previous subsection pointed out, we can only approximate the model behaviors before the Zeno time point. Therefore, the first issue is to be able to tell how close the simulation results are to the exact solutions before the Zeno time point. This will decide when the transitions from pre-Zeno states to post-Zeno states are taken. The second issue is how to establish the initial conditions of the dynamics after the Zeno time point from the approximated simulation results.

Issue 1: Relaxing Guard Expressions.

To solve the first issue, we first relax the guard conditions defining the transitions from the pre-Zeno states to the post-Zeno states; if the current states fall into a neighborhood of the Zeno states (the states at the Zeno time point), the guard is enabled and transition is taken. Note that when the transition is taken, the system has a new dynamics and the rest of the events before the Zeno time point, which are infinite in number, are discarded. Therefore the computation before the approximated Zeno time point can be finished in finite time.

A practical problem now is to define a good neighborhood such that the approximation is "close enough" to the exact Zeno behavior. We propose two criteria. The first criterion is based on the error tolerance ε_0[3]. We rewrite (6.12) as

$$G_{e_c}^{\varepsilon_0} = \{x \in \mathcal{R}^n \mid g_{e_c}(x) \leq \varepsilon_0\}, \tag{6.13}$$

meaning if $x(t)$ is the solution of $\dot{x} = f_q(x)$ with $q = \overline{\mathfrak{s}}(e_c)$, and if the evaluation result of $g_{e_c}(x(t))$ falls inside $[0, \varepsilon_0]$, the simulation results of $x(t)$ will be thought as close enough to the exact solution at the Zeno time point, and the transition will be taken. In fact, because ε_0 is the smallest amount that can be reliably distinguished, any value in $[0, \varepsilon_0]$ will be treated the same.

The second criterion is based on the minimum step size h_0. Suppose the evaluation result of $g_{e_c}(x(t))$ is outside of the range $[0, \varepsilon_0]$. If it takes less than h_0 time for the dynamics to drive the value of $g_{e_c}(x(t))$ down to 0, then we will treat the current states as close enough to the Zeno states. This criterion prevents the numerical integration from failing with a step size smaller than h_0, which may be caused by some rapidly changing dynamics, such as those in (6.11).

[3]If $g_{e_c}(x)$ is a vector valued function, then ε_0 is a vector with ε_0 as the elements.

We first get a linear approximation to function $g_{e_c}(x(t))$ around t_0 (cf. [84],[37]),

$$g_{e_c}(x(t_0+h)) = g_{e_c}(x(t_0)) + \frac{\partial g_{e_c}(x)}{\partial x} \cdot f_q(x) \mid_{x=x(t_0)} \cdot h + O(h^2), \tag{6.14}$$

where h is the integration step size. Because we are interested in the model's behavior when h is close to h_0, where h is very small, we can discard the $O(h^2)$ term in (6.14). We are interested in how long it takes for the value of function $g_{e_c}(x(t_0))$ to go to 0, so we calculate the required step size by solving (6.14),

$$h = -\frac{g_{e_c}(x(t_0))}{\frac{\partial g_{e_c}(x)}{\partial x} \cdot f_q(x) \mid_{x=x(t_0)}}. \tag{6.15}$$

Now we say that if $h < h_0$, the states are close enough to the Zeno time point. So we rewrite the boolean expression (6.12) as

$$G_{e_c}^{h_0} = \left\{ x \in \mathcal{R}^n \mid -\frac{g_{e_c}(x)}{\frac{\partial g_{e_c}(x)}{\partial x} \cdot f_q(x)} \leq h_0 \right\}. \tag{6.16}$$

In the end, we give a complete approximated guard expression of the transition e_c from a pre-Zeno state to a post-Zeno state:

$$G_{e_c}^{\text{approx}} = G_{e_c}^{\varepsilon_0} \cup G_{e_c}^{h_0}.$$

This means that if either guard expression in (6.13) and (6.16) evaluates to be true, the transition will be taken. Performing this process on each guard in the set $\{G_{e_c} \mid c \in C\}$ we obtain the set $\{G_{e_c}^{\text{approx}} \mid c \in C\}$. Note that to ensure deterministic transitions, we also subtract the same set from the original guard sets defined in (6.7). Replacing the guard expressions given in Section 6.2 with these approximated ones, we obtain an approximation to the completed hybrid system $\overline{\mathcal{H}}$, $\overline{\mathcal{H}}^{\text{approx}}$. This is the completed hybrid system that is implemented for simulation.

Issue 2: Reinitialization.

The other issue is how to reinitialize the initial continuous states of the new dynamics defined in a post-Zeno state. Theoretically, these initial continuous states are just the states at the Zeno time point, meaning that they satisfy the guard expression in (6.12). This is guaranteed by the identity reset maps associated with the transitions.

In some circumstances, like the examples discussed in this chapter, the initial continuous states can be explicitly and precisely calculated. However, in general, if there are more variables involved in guard expressions than the constraints enforced by guard expressions, we cannot resolve all initial states. In this case, we have to use the simulation results as part of the initial states. Clearly, since in

simulation we do not actually reach the Zeno time point, the initial states are just approximations. Consequently, the simulation of the dynamics of post-Zeno states will be approximation too.

We applied the technique given in this section to the Newton's cradle model with collision restitution $k = 0$ and the execution results are shown in Figure 1.17.

Chapter 7

Conclusions

The purpose of this dissertation is to interpret hybrid systems as executable programs written in a programming language with a hybrid system semantics. In order to achieve this, an operational semantics needs to be given to that programming language for executing the programs. This dissertation proposes such an operational semantics with focuses on a few semantic issues. The objective is to make the operational semantics complete, precise, and understandable.

7.1 Summary of Results

We start with the question of how to precisely and unambiguously represent the discontinuities in continuous-time signals and simultaneous discrete events in discrete-event signals when these signals are represented in computer. To solve this problem, a two-dimension domain called "super-dense time" is used as the domain for defining signals such that a signal may have multiple values at the same time point while keeping the values ordered.

In Chapter 2, the Tagged Signal model is used as the mathematical framework to formally define different kinds of signals, including CT and DE signals. Then discrete representations of these signals are given, which are exactly the same as how a computer represents the signals.

In Chapter 3, actors, where signals interact with each other, are formally defined as functions mapping a set of signals to another set of signals. This chapter gives a small library typical actors used in hybrid systems and two useful mechanisms of how CT and DE signals interact with each other.

In Chapter 5, the existence, uniqueness, and liveness properties of behaviors for compositions of actors are studied by leveraging a least fixed point theorem. An interface theory is developed to abstractly represent *causality* of actors and allow algebraic compositions of causality interfaces for composite actors.

Chapter 5 gives an operational semantics based on the denotational semantics given in Chapter 4. This operational semantics describes in details the incremental construction of discrete representations of signals and the evaluation of signal values by leveraging the synchronous languages principles and abstract actor semantics. This chapter formalizes the operational semantics with the Abstract State Machine.

Chapter 6 gives a practical technique for simulating Zeno hybrid systems beyond their Zeno time points. This technique gives a systematic way to complete the system dynamics by introducing *post-Zeno* states, and provides a practical simulation strategy for constructing approximations of Zeno signals. This technique allows executions to *jump over* Zeno time points and proceed with the dynamics defined in the post-Zeno states. In this way, simulation of a Zeno hybrid system model can go beyond its Zeno time point and therefore reveal the complete dynamics of the system being modeled.

The operational semantics proposed in this dissertation has been implemented in HyVisual, which is a software tool for modeling and simulating hybrid systems. HyVisual is part of the Ptolemy II software framework, which is available in open-source form (BSD-style license) at http://ptolemy.org.

7.2 Future Work

The operational semantics presented in this dissertation supports heterogeneous and hierarchical composition of different models of computation, such as CT, DE, finite state machines, and synchronous languages, and modular execution of the composition as a whole. This ability makes it easy to jointly model and design software controlled systems.

In hybrid systems, a typical way to generate discrete events is through event detection. Therefore, being able to accurately resolve continuous dynamics is of great importance. In this dissertation, we focused on the Runge-Kutta methods. However, in most efficient solvers, linear multi-step methods are used because they tend to be faster and more accurate. There are numerous papers about the

numerical analysis techniques. For example, [83, 77] are specifically targeted on circuit simulations. Therefore, another interesting future work will be to investigate how the linear multi-step methods might improve the accuracy of numerical resolutions for continuous dynamics.

Any model is an abstraction of the real systems. Hybrid system models abstract rapidly changing dynamics as discrete changes. One benefit from this abstraction is the avoidance of stiff systems and therefore the simulation performance can be greatly improved. However, which level of abstraction to choose is always an engineering question and requires a lot of experience and expertise from model designers. It will be very useful to be able to compare the execution results of hybrid system models and continuous models, where these models describe the same system, and reason about the effect of abstractions and choose the right level of abstraction.

Bibliography

[1] S. Abramsky, S. J. Gay, and R. Nagarajan. Interaction categories and the foundations of typed concurrent programming, 1995. 27

[2] R. Alur, T. Dang, J. Esposito, Y. Hur, F. Ivancic, V. Kumar, I. Lee, P. Mishra, G. J. Pappas, and O. Sokolsky. Hierarchical modeling and analysis of embedded systems. *Proceedings of the IEEE*, 91(1):11–28, 2003. 1

[3] R. Alur, R. Grosu, I. Lee, and O. Sokolsky. Compositional modeling and refinement for hierarchical hybrid systems. *Process Theory for Hybrid Systems*, 68(1-2):105, 2006. 1

[4] A. D. Ames, A. Abate, and S. Sastry. Sufficient conditions for the existence of zeno behavior. In *44th IEEE Conference on Decision and Control and European Control Conference ECC*, 2005. 148, 153

[5] A. D. Ames and S. Sastry. Blowing up affine hybrid systems. In *Decision and Control, 43rd IEEE Conference*, volume 1, pages 473– 478, 2004. 148

[6] A. D. Ames and S. Sastry. Characterization of zeno behavior in hybrid systems using homological methods. In *24th American Control Conference*, 2005. 34, 37

[7] A. D. Ames, P. Tabuada, and S. Sastry. On the stability of zeno equilibria. In A. Tiwari, editor, *Hybrid Systems: Computation and Control (HSCC)*, volume LNCS 3927, pages 34 – 48, Santa Barbara, CA, USA, 2006. Springer-Verlag. 148

[8] A. D. Ames, H. Zheng, R. Gregg, and S. Sastry. Is there life after zeno? taking executions past the breaking (zeno) point. In *American Control Conference (ACC)*, 2006. 148, 158, 160

[9] F. Arbab. Abstract behavior types: A foundation model for components and their composition. *Science of Computer Programming*, 55:3–52, 2005. 27

[10] J. R. Armstrong and F. G. Gray. *VHDL Design Representation and Synthesis*. Prentice-Hall, 2000. 47

[11] E. Asarin, O. Bournez, T. Dang, and O. Maler. Approximate reachability analysis of piecewise-linear dynamical systems. In *Hybrid Systems: Computation and Control (HSCC)*, volume LNCS 1790. Springer-Verlag, 2000. 1

[12] F. Baccelli, G. Cohen, G. J. Olster, and J. P. Quadrat. *Synchronization and Linearity, An Algebra for Discrete Event Systems*. Wiley, New York, 1992. 100

[13] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in bip. In *International Conference on Software Engineering and Formal Methods (SEFM)*, pages 3–12, Pune, 2006. 3

[14] H. Bekic. Definable operation in general algebras, and the theory of automata and flowcharts. In *Programming Languages and Their Definition - Hans Bekic (1936-1982)*, pages 30–55, London, UK, 1984. Springer-Verlag. 116

[15] A. Benveniste. Compositional and uniform modeling of hybrid systems. *IEEE Transactions on Automatic Control*, 43(4):579–584, 1998. 2

[16] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991. 2, 12, 77, 114

[17] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedsings of the IEEE*, 91(1):64–83, 2003. 114

[18] G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992. 114

[19] E. Borger. High level system design and analysis using abstract state machines. In *FM-Trends 98: Proceedings of the International Workshop on Current Trends in Applied Formal Method*, pages 1–43, London, UK, 1999. Springer-Verlag. 119, 120

[20] R. Boute. Integrating formal methods by unifying abstractions. In E. Boiten, J. Derrick, and G. Smith, editors, *Fourth International Conference on Integrated Formal Methods (IFM)*, volume LNCS 2999, page 441460, Canterbury, Kent, England, 2004. Springer-Verlag. 3

[21] C. Brooks, A. Cataldo, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, and H. Zheng. Hyvisual: A hybrid system visual modeler. Technical report, University of California, Berkeley, July 15 2005. 1, 3, 143

[22] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Heterogeneous concurrent modeling and design in java. Technical report, University of California, July 29 2004. 3, 143

[23] 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"*, 4:155–182, 1994. 2

[24] R. L. Burden and J. D. Faires. *Numerical analysis, 7th ed.* Brroks/Cole, 2001. 167, 168

[25] L. P. Carloni, M. D. DiBenedetto, A. Pinto, and A. Sangiovanni-Vincentelli. Modeling techniques, programming languages, and design toolsets for hybrid systems. Technical report, Columbus Project, June 2004. 2

[26] L. P. Carloni, R. Passerore, A. Pinto, and A. Sangiovanni-Vincentelli. Languages and tools for hybrid systems design. *Foundations and Trends in Design Automation*, 1(1):1–204, January 2006. 1

[27] C. G. Cassandras. *Discrete Event Systems, Modeling and Performance Analysis.* Irwin, 1993. 2, 47

[28] A. Cataldo, C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, and H. Zheng. Hyvisual: A hybrid system visual modeler. Technical report, University of California, Berkeley, July 17 2003. 4, 143

[29] P. G. D. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981. 119

[30] J. B. Dabney and T. L. Harman. *Mastering SIMULINK.* Prentice Hall Professional Technical Reference, 2003. 1, 135

[31] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 2002. 79, 81, 82, 115, 156

[32] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool kronos. In *Hybrid Systems III: Verification and Control.*, volume LNCS 1066. Springer-Verlag, 1996. 1

[33] A. Deshpande, A. Gollu, and L. Semenzato. The shift programming language for dynamic networks of hybrid automata. *IEEE Trans. on Automatic Control*, 43(4), 1998. 1, 12

[34] A. Deshpande and P. Varaiya. Information structures for control and verification of hybrid systems. In *American Control Conference (ACC)*, 1995. 1

[35] R. Djenidi, C. Lavarenne, R. Nikoukhah, Y. Sorel, and S. Steer. From hybrid simulation to real-time implementation. In *11th European Simulation Symposium and Exhibition*, Erlangen-Nuremberg, 1999. 1

[36] S. A. Edwards and E. A. Lee. The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming*, 48(1):21–42, 2003. 114, 116

[37] J. M. Esposito, V. Kumar, and G. J. Pappas. Accurate event detection for simulating hybrid systems. In *Hybrid Systems: Computation and Control (HSCC)*, volume LNCS 2034, pages 204–217, London, UK, 2001. Springer-Verlag. 169, 171

[38] R. Esser, J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions On Programming Languages And Systems*, 9(3):319–349, 1987. 106

[39] G. S. Fishman. *Discrete-Event Simulation: Modeling, Programming, and Analysis*. Springer-Verlag, 2001. 2

[40] C. Fong. *Discrete-Time Dataflow Models for Visual Simulation in Ptolemy II*. Master's report, University of California, Berkeley, 2001. vii, 5, 6

[41] P. Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley, 2003. 2

[42] A. Girault, B. Lee, and E. A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions On Computer-aided Design Of Integrated Circuits And Systems*, 18(6), 1999. 103

[43] G. Goessler and A. Sangiovanni-Vincentelli. Compositional modeling in metropolis. In *Second International Workshop on Embedded Software (EMSOFT)*, Grenoble, France, 2002. Springer-Verlag. 2

[44] G. Goessler and J. Sifakis. Composition for component-based modeling. *Science of Computer Programming*, 55, 2005. 3

[45] S. Graf, P. L. Guernic, T. Gauthier, M. L. Borgne, and C. L. Maire. Programming real-time applications with signal. *Proceedings of the IEEE*, 79(9), 1991. 114

[46] Y. Gurevich. Evolving algebras: An attempt to discover semantics. In G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science*, pages 266–292. World Scientific", River Edge, NJ, 1993. 119, 120

[47] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1319, 1991. 114

[48] T. A. Henzinger. The theory of hybrid automata. In M. Inan and R. Kurshan, editors, *Verification of Digital and Hybrid Systems*, volume 170 of *NATO ASI Series F: Computer and Systems Sciences*, pages 265–292. Springer-Verlag, 2000. 1

[49] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. (HYTECH): A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1(1-2):110–122, 1997. 1

[50] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. In *EMSOFT 2001*, volume LNCS 2211, Tahoe City, CA, 2001. Springer-Verlag. 77

[51] F. Herrera and E. Villar. A framework for embedded system specification under different models of computation in systemc. In *Design Automation Conference (DAC)*, San Francisco, 2006. ACM. 2

[52] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, volume SIGPLAN Notices 23(7), pages 35–46, Atlanta, Georgia, 1988. 106

[53] J. K. Huggins and C. Wallace. An abstract state machine primer. Technical report, Technical Report CS-TR-02-04, Computer Science Department, Michigan Technological University, December. 119

[54] A. Jantsch. *Modeling Embedded Systems and SoCs - Concurrency and Time in Models of Computation*. Morgan Kaufmann, 2003. 2

[55] K. H. Johansson, J. Lygeros, S. Sastry, and M. Egerstedt. Simulation of zeno hybrid automata. In *Proceedings of the 38th IEEE Conference on Decision and Control*, Phoenix, AZ, 1999. 148, 149, 151, 153

[56] P. Kopke, T. Henzinger, A. Puri, and P. Varaiya. What's decidable about hybrid automata? In *27th Annual ACM Symposioum on Theory of Computing (STOCS)*, pages 372–382, 1995. 1

[57] K. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2), 1997. 1

[58] E. A. Lee. Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering*, 7:25–45, 1999. 2, 13, 36, 93

[59] E. A. Lee. Overview of the ptolemy project. Technical report, University of California, Berkeley, 2003. 3, 5, 143

[60] E. A. Lee, S. Neuendorffer, and M. J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, 12(3):231–260, 2003. 3

[61] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on CAD*, 17(12), 1998. 2, 4, 27, 57

[62] E. A. Lee and H. Zheng. Operational semantics of hybrid systems. In L. Thiele, editor, *Hybrid Systems: Computation and Control (HSCC)*, volume LNCS 3414, pages 25–53, Zurich, Switzerland, 2005. Springer-Verlag. 2, 29, 135, 136

[63] E. A. Lee and H. Zheng. Hyvisual: A hybrid system modeling framework based on ptolemy ii. In *to appear in IFAC Conference on Analysis and Design of Hybrid Systems (ADHS'06))*, Alghero, Sardinia, 2006. 1, 3

[64] E. A. Lee, H. Zheng, and Y. Zhou. Causality interfaces and compositional causality analysis. In *Invited paper in Foundations f Interface Technologies (FIT), Satellite to CONCUR 2005*, San Francisco, CA, 2005. 80, 98

[65] M. A. Lemkin. *Micro Accelerometer Design with Digital Feedback Control*. Ph.D. dissertation, University of California, Berkeley, 1997. 5

[66] J. Liu. *Continuous Time and Mixed-Signal Simulation in Ptolemy II*. M.S. thesis, University of California, Berkeley, 1998. 5, 135

[67] J. Liu and E. A. Lee. On the causality of mixed-signal and hybrid models. In *6th International Workshop on Hybrid Systems: Computation and Control (HSCC '03)*, Prague, Czech Republic, 2003. 50

[68] X. Liu. Semantic foundation of the tagged signal model. Ph.D. Thesis Technical Memorandum UCB/EECS-2005-31, EECS Department, University of California, Berkeley, December 20 2005. 28, 29, 91, 93, 94

[69] X. Liu and E. A. Lee. CPO semantics of timed interactive actor networks. Technical Report UCB/EECS-2006-67, EECS Department, University of California, Berkeley, May 18 2006. 28, 80, 90

[70] J. Lygeros. *Lecture Notes on Hybrid Systems*. ENSIETA 2-6/2/2004, 2004. 153

[71] J. Lygeros, C. Tomlin, and S. Sastry. Controllers for reachability specifications for hybrid systems. *Automatica*, (Special Issue on Hybrid Systems), 1999. 1

[72] N. Lynch, R. Segala, F. Vaandrager, and H. Weinberg. Hybrid I/Oautomata. In *Proceedings of the DIMACS/SYCON workshop on Hybrid systems III: verification and control*, volume LNCS 1066, pages 496–510. Springer-Verlag, 1996. 1

[73] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, volume 600, pages 447–484, London, UK, 1992. Springer-Verlag. 4, 29

[74] I. Mitchell and C. Tomlin. Level set methods for computation in hybrid systems. In *Hybrid Systems: Computation and Control (HSCC)*, volume LNCS 1790. Springer-Verlag, 2000. 1

[75] P. J. Mosterman. An overview of hybrid simulation phenomena and their support by simulation packages. In J. H. v. Schuppen, editor, *Hybrid Systems: Computation and Control (HSCC)*, volume LNCS 1569. Springer-Verlag, 1999. 2, 15, 148

[76] P. J. Mosterman. Hybrsim - a modelling and simulation environment for hybrid bond graphs. *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering*, V216(1):35, 2002. 10.1243/0959651021541417. 1

[77] A. R. Newton and A. L. Sangiovanni-Vincentelli. Relaxation-based electrical simulation. *Ieee Transactions On Electron Devices*, 30(9):1184–1207, 1983. 175

[78] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. *SIGPLAN Notices*, 19(5):177–184, 1984. 106

[79] T. Park and P. I. Barton. State event location in differential-algebraic models. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 6(2):137–165, 1996. 169

[80] A. Pinto, L. P. Carloni, R. Passerone, and A. Sangiovanni-Vincentelli. Interchange formats for hybrid systems: Abstract semantics. In J. Hespanha and A. Tiwari, editors, *Hybrid Systems: Computation and Control*, pages 491–506, March 2006. 2

[81] A. Puri and P. Varaiya. Verification of hybrid systems using abstractions. In *Hybrid Systems Workshop*, volume Hybrid Systems II, LNCS 999, pages 359–369. Springer-Verlag, 1994. 1

[82] H. Royden. *Real Analysis*. Prentice Hall, Englewood Cliffs, NJ, 3rd edition, 1988. 59

[83] A. Sangiovanni-Vincentelli. Circuit simulation [invited paper]. In P. Antognetti, D. O. Pederson, and H. De Man, editors, *Computer Design Aids for VLSI Circuits, Sjthoff and Noordhoff, Alphen aan den Rijin, pp. 19-113, 1981. and Proceedings of the NATO Advanced Study Institute, Urbano, Italy, July 1980.* 175

[84] L. F. Shampine, I. Gladwell, and R. W. Brankin. Reliable solution of special event location problems for odes. *ACM Trans. Math. Softw.*, 17(1):11–25, 1991. 169, 171

[85] B. I. Silva, K. Richeson, B. Krogh, and A. Chutinan. Modeling and verifying hybrid dynamic systems using checkmate. In *Automation of Mixed Processes: Dynamic Hybrid Systems (ADPM)*, Dortmund Germany, 2000. Shaker Verlag, Aachen. 1

[86] S. Swan. An introduction to system level modeling in systemc 2.0. Technical report, Open SystemC Initiative, May 2001 2001. 2

[87] J. Sztipanovits and G. Karsai. Model-integrated computing. *IEEE Computer*, 1997. 2

[88] C. L. Talcott. Interaction semantics for components of distributed systems. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, 1996. 27

[89] P. Taylor. *Practical Foundations of Mathematics*. Cambridge University Press, 1999. 28

[90] U. o. P. M. team. Hsif semantics (version 3, synchronous edition). Technical report, University of Pennsylvania, August 22 2002. 2

[91] M. Tiller. *Introduction to Physical Modeling with Modelica*. Kluwer Academic Publishers, 2001. 517110. 1, 2

[92] F. D. Torrisi, A. Bemporad, G. Bertini, P. Hertach, D. Jost, and D. Mignone. Hysdel 2.0.5 - user manual. Technical report, ETH, 2002. 1, 12

[93] A. van der Schaft and H. Schumacher. *An Introduction to Hybrid Dynamical Systems*. Springer-Verlag, 2000. 29, 149

[94] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation*. Academic Press, 2000. 2, 47

[95] J. Zhang, K. H. Johansson, J. Lygeros, and S. Sastry. Zeno hybrid systems. *Int. J. Robust and Nonlinear Control*, 11(2):435–451, 2001. 148, 153, 157

[96] H. Zheng, E. A. Lee, and A. D. Ames. Beyond zeno: Get on with it! In A. Tiwari, editor, *Hybrid Systems: Computation and Control (HSCC)*, volume LNCS 3927, pages 568 – 582, Santa Barbara, CA, USA, 2006. Springer-Verlag. 149

[97] Y. Zhou and E. A. Lee. Causality interfaces for actor networks. Technical Report UCB/EECS-2006-148, EECS Department, University of California, Berkeley, November 16 2006. 80, 98