Chapter 2 from: C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng "Heterogeneous Concurrent Modeling and Design in Java (Volume 3: Ptolemy II Domains)," Technical Memorandum UCB/ERL M04/17, University of California, Berkeley, CA USA 94720, June 24, 2004.

2

CT Domain

Author: *Jie Liu*

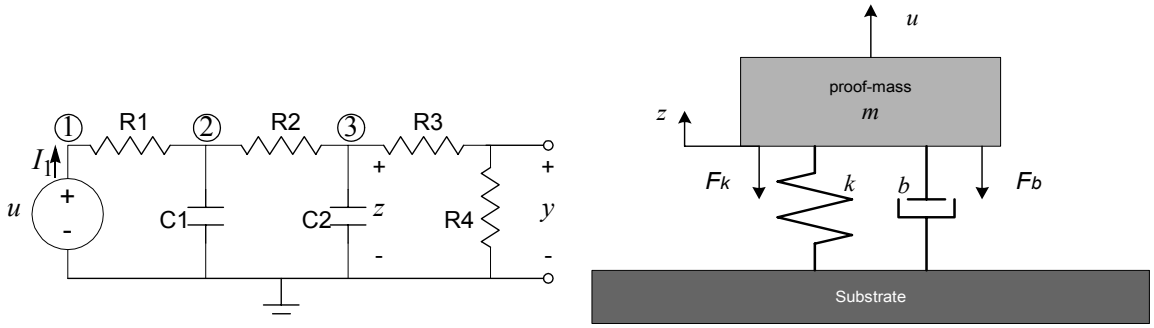
2.1 Introduction

The continuous-time (CT) domain in Ptolemy II aims to help the design and simulation of systems that can be modeled using ordinary differential equations (ODEs). ODEs are often used to model analog circuits, plant dynamics in control systems, lumped-parameter mechanical systems, lumped-parameter heat flows and many other physical systems.

Let's start with an example. Consider a second order differential system,

$$\begin{aligned} &m\ddot{z}(t) + b\dot{z}(t) + kz(t) = u(t) \quad . \\ &y(t) = c \cdot z(t) \\ &z(0) = 10, \dot{z}(0) = 0. \end{aligned} \tag{1}$$

The equations could be a model for an analog circuit as shown in figure 2.1(a), where z is the voltage



(a) A circuit implementation.

(b) A mechanical implementation.

FIGURE 2.1. Possible implementations of the system equations.

of node 3, and

$$m = R1 \cdot R2 \cdot C1 \cdot C2 \tag{2}$$

$$k = R1 \cdot C1 + R2 \cdot C2$$

$$b = 1$$

$$c = \frac{R4}{R3 + R4}.$$

Or it could be a lumped-parameter spring-mass mechanical model for the system shown in figure 2.1(b), where z is the position of the mass, m is the mass, k is the spring constant, b is the damping parameter, and $c = 1$.

In general, an ODE-based continuous-time system has the following form:

$$\dot{x} = f(x, u, t) \tag{3}$$

$$y = g(x, u, t) \tag{4}$$

$$x(t_0) = x_0, \tag{5}$$

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

$$\dot{x} = \frac{dx}{dt}. \tag{6}$$

Equations (3), (4), and (5) are called the *system dynamics*, the *output map*, and the *initial condition* of the system, respectively.

For example, for the mechanical system above, if we define a vector

$$x(t) = \begin{bmatrix} z(t) \\ \dot{z}(t) \end{bmatrix}, \tag{7}$$

then system (1) can be written in form of (3)-(5), like

$$\dot{x}(t) = \frac{1}{m}\begin{bmatrix} 0 & 1 \\ -k & -b \end{bmatrix} x(t) + \begin{bmatrix} 0 \\ 1/m \end{bmatrix} u(t) \tag{8}$$

$$y(t) = \begin{bmatrix} c & 0 \end{bmatrix} x(t)$$

$$x(0) = \begin{bmatrix} 10 \\ 0 \end{bmatrix}.$$

The solution, $x(t)$, of the set of ODE (3)-(5), is a continuous function of time, also called a *waveform*, which satisfies the equation (3) and initial condition (5). The output of the system is then defined as a function of $x(t)$ and $u(t)$, which satisfies (4). The precise solution of a set of ODEs is usually impossible to be found using digital computers. Numerical solutions are approximations of the precise solution. A numerical solution of ODEs are usually done by integrating the right-hand side of (3) on a

discrete set of time points. Using digital computers to simulate continuous-time systems has been studied for more than three decades. One of the most well-known tools is Spice [109]. The CT domain differs from Spice-like continuous-time simulators in two ways — the system specification is somewhat different, and it is designed to interact with other models of computation.

2.1.1 System Specification

There are usually two ways to specify a continuous-time system, the conservation-law model and the signal-flow model [61]. The conservation-law models, like the nodal analysis in circuit simulation [58] and bond graphs [125] in mechanical models, define systems by their physical components, which specify relations of *cross* and *through* variables, and *conservation laws* are used to compile the component relations into global system equations. For example, in circuit simulation, the cross variables are voltages, the through variables are currents, and the conservation laws are Kirchhoff's laws. This model directly reflects the physical components of a system, thus is easy to construct from a potential implementation. The actual mathematical representation of the system is hidden. In signal-flow models, entities in a system are maps that define the mathematical relation between their input and output signals. Entities communicate by passing signals. This kind of models directly reflects the mathematical relations among signals, and is more convenient for specifying systems that do not have an explicit physical implementation yet.

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

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

2. *The signal flow model is more flexible and extensible*, in the sense that it is easy to embed components that are designed using other models. For example, a discrete controller can be modeled as a component that internally follows a discrete event model of computation but exposes a continuous-time interface.

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

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

The signal flow block diagram of the system (3) - (5) is shown in figure 2.2. The system dynamics (3) is built using integrators with feedback. In this figure, u, $\dot{x}$, x, and y, are continuous signals flowing from one block to the next. Notice that this diagram is only conceptual, most models may involve multiple integrators[1]. Time is shared by all components, so it is not considered as an input. At any fixed time t, if the "snapshot" values $x(t)$ and $u(t)$ are given, then $\dot{x}(t)$ and $y(t)$ can be found by evaluating f

1. Ptolemy II does not support vectorization in the CT domain yet.

and g, which can be achieved by firing the respective blocks. The "snapshot" of all the signals at t is called the *behavior* of the system at time t.

The signal-flow model for the example system (1) is shown in figure 2.3. For comparison purpose, the conservation-law model (modified nodal analysis) of the system shown in figure 2.1(a) is shown in (9).

$$\begin{bmatrix} \frac{1}{R1} & -\frac{1}{R1} & 0 & 0 & -1 \\ -\frac{1}{R1} & \frac{1}{R1}+\frac{1}{R2}+C1\frac{d}{dt} & -\frac{1}{R2} & 0 & 0 \\ 0 & -\frac{1}{R2} & \frac{1}{R2}+\frac{1}{R3}+C2\frac{d}{dt} & -\frac{1}{R3} & 0 \\ 0 & 0 & -\frac{1}{R3} & \frac{1}{R3}+\frac{1}{R4} & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ y \\ I_1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ u \end{bmatrix} \tag{9}$$

By doing some math, we can see that (9) and (8) are in fact equivalent. Equation (9) can be easily assembled from the circuit, but it is more complicated than (8). Notice that in (9) d/dt is the derivative operator, which is replaced by an integration algorithm at each time step, and the system equations reduce to a set of algebraic equations. Spice software is known to have a very good simulation engine for models in form of (9).

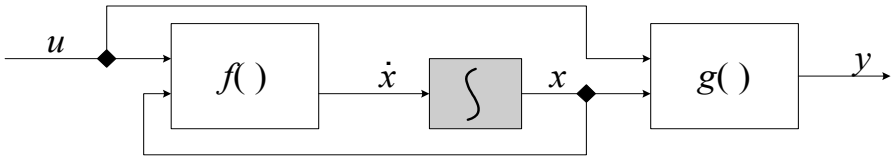


FIGURE 2.2. A conceptual block diagram for continuous time systems.

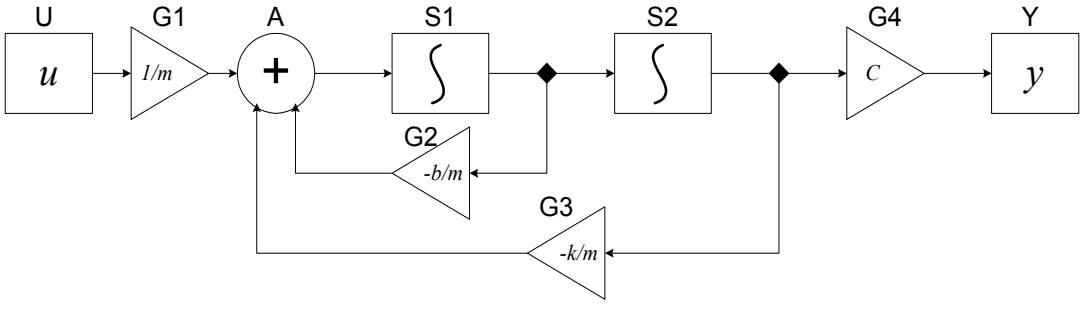


FIGURE 2.3. The block diagram for the example system.

2.1.2 Time

One distinct characterization of the CT model is the continuity of time. This implies that a continuous-time system have a behavior at any time instance. The simulation engine of the CT model should be able to compute the behavior of the system at any time point, although it may march discretely in time. In order to achieve an accurate simulation, time should be carefully discretized. The discretization of time, which appears as integration step sizes, may be determined by time points of interest (e.g. discontinuities), by the numerical error of integration, and by the convergence in solving algebraic equations.

Time is also global, which means that all components in the system share the same notion of time.

2.2 Solving ODEs numerically

We outline some basic terminologies on numerical ODE solving techniques that are used in this chapter. This is not a summary of numerical ODE solving theory. For a detailed treatment for ODEs and their numerical solutions, please refer to books on numerical solutions for ODEs, e.g. [43].

Not all ODEs have a solution, and some ODEs have more than one solution. In such situations, we say that the solution is not well defined. This is usually a result of errors in the system modeling. We restrict our discussion to systems that have unique solutions. Theorem 1 in Appendix A states the conditions for the existence and uniqueness of solutions of ODEs. Roughly speaking, we denote by D a set in $\Re$ which contains at most a finite number of points per unit interval, and let u be piecewise-continuous on $\Re - D$. Then, for any fixed $u(t)$, if f is also piecewise-continuous on $\Re - D$, and f satisfies the Lipschitz condition (see e.g. [43]), then the ODE (3) with the initial condition (5) has a unique solution. The solution is called the *state trajectory* of the system. The key of simulating a continuous-time system numerically is to find an accurate numerical approximation of the state trajectory.

2.2.1 Basic Notations

Usually, only the solution on a finite time interval $[t_0,t_f]$ is needed. A simulation of the system is performed on discrete time points in this interval. We denote by

$$Tc = \{t_0, t_1, t_2, \ldots t_n, \ldots t_f\}, Tc \subset [t_0, t_f], \tag{10}$$

where

$$t_0 < t_1 < t_2 < \ldots < t_n < \ldots < t_f, \tag{11}$$

the set of the discrete time points of interest. To explicitly illustrate the discretization of time and the difference between the precise solution and the numerical solution, we use the following notation in the rest of the chapter:

- t_n : the n-th time point, to explicitly show the discretization of time. However, we write t, if the index n is not important.
- $x[t_i, t_j]$: the *precise* (continuous) state trajectory from time t_i to t_j;
- $x(t_n)$: the *precise* solution of (3) at time t_n;
- x_{t_n} : the *numerical* solution of (3) at time t_n;
- $h_n = t_n - t_{n-1}$: step size of numerical integration. We also write h if the index n in the sequence

is not important. For accuracy reason, h may not be uniform.

- $\|x(t_n) - x_{t_n}\|$: the 2-normed difference between the precise solution and the numerical solution at step *n* is called the (*global*) *error* at step *n*; the difference, when we assume $x_{t_0} \ldots x_{t_{n-1}}$ are precise, is called the *local error* at step *n*. Local errors are usually easy to estimate and the estimation can be used for controlling the accuracy of numerical solutions.

A general way of numerically simulating a continuous-time system is to compute the state and the output of the system in an increasing order of t_n. Such algorithms are called the *time-marching* algorithms, and, in this chapter, we only consider these algorithms. There are variety of time marching algorithms that differ on how x_{t_n} is computed given $x_{t_0} \ldots x_{t_{n-1}}$. The choice of algorithms is application dependent, and usually reflects the speed, accuracy, and numerical stability trade-offs.

2.2.2 Fixed-Point Behavior

Numerical ODE solving algorithms approximate the derivative operator in (3) using the history and the current knowledge on the state trajectory. That is, at time t_n, the derivative of x is approximated by a function of $x_{t_0}, \ldots, x_{t_{n-1}}, x_{t_n}$, i.e.

$$\dot{x}_{t_n} = p(x_{t_0}, \ldots, x_{t_{n-1}}, x_{t_n}) . \tag{12}$$

Plugging (3) in this, we get

$$p(x_{t_0} \ldots x_{t_{n-1}}, x_{t_n}) = f(x_{t_n}, u(t_n), t_n) \tag{13}$$

Depending on whether x_{t_n} explicitly appears in (13), the algorithms are called *explicit integration algorithms* or *implicit integration algorithms*. That is, we end up solving a set of algebraic equations in one of the two forms:

$$x_{t_n} = F_E(x_{t_0}, \ldots, x_{t_{n-1}}) \tag{14}$$

or

$$F_I(x_{t_0}, \ldots, x_{t_n}) = 0 , \tag{15}$$

where F_E and F_I are derived from the time t_n, the input $u(t_n)$, the function *f*, and the history of x and $\dot{x}$. Solving (14) or (15) at a particular time t_n is called an *iteration* of the CT simulation at t_n.

Equation (14) can be solved simply by a function evaluation and an assignment. But the solution of (15) is the *fixed point* of F_I, which may not exist, may not be unique, or may not be able to be found. The *contraction mapping theorem* [21] shows the existence and uniqueness of the fixed-point solution, and provides one way to find it. Given the map F_I that is a local contraction map (generally true for small enough step sizes) and let an initial guess σ_0 be in the contraction radius, then a unique fixed point exists and can be found by iteratively computing:

$$\sigma_1 = F_E(\sigma_0), \sigma_2 = F_E(\sigma_1), \sigma_3 = F_E(\sigma_2), \ldots \tag{16}$$

Solving both (14) and (15) should be thought of as finding the fixed-point behavior of the system at a particular time. This means both functions F_E and F_I should be smooth w.r.t. time, during one iteration of the simulation. This further implies that the topology of the system, all the parameters, and all the internal states that the firing functions depend on should be kept unchanged. We require that

domain polymorphic actors to update internal states only in the `postfire()` method exactly for this reason.

2.2.3 ODE Solvers Implemented

The following solvers has been implemented in the CT domain.

1. Forward Euler solver:

$$\begin{aligned} x_{t_{n+1}} &= x_{t_n} + h_{n+1} \cdot \dot{x}_{t_n} \\ &= x_{t_n} + h_{n+1} \cdot f(x_{t_n}, u_{t_n}, t_n) \end{aligned} \tag{17}$$

2. Backward Euler solver:

$$\begin{aligned} x_{t_{n+1}} &= x_{t_n} + h_{n+1} \cdot \dot{x}_{t_{n+1}} \\ &= x_{t_n} + h_{n+1} \cdot f(x_{t_{n+1}}, u_{t_{n+1}}, t_{n+1}) \end{aligned} \tag{18}$$

3. 2(3)-order Explicit Runge-Kutta solver

$$\begin{aligned} K_0 &= h_{n+1} \cdot f(x_{t_n}, u_{t_n}, t_n) \\ K_1 &= h_{n+1} \cdot f(x_{t_n} + K_0/2, u_{t_n + h_{n+1}/2}, t_n + h_{n+1}/2) \\ K_2 &= h_{n+1} \cdot f(x_{t_n} + 3K_1/4, u_{t_n + 3h_{n+1}/4}, t_n + 3h_{n+1}/4) \\ \tilde{x}_{t_{n+1}} &= x_{t_n} + \frac{2}{9}K_0 + \frac{1}{3}K_1 + \frac{4}{9}K_2 \end{aligned} \tag{19}$$

with error control:

$$\begin{aligned} K_3 &= h_{n+1} \cdot f(\tilde{x}_{t_{n+1}}, u_{t_{n+1}}, t_{n+1}) \\ LTE &= -\frac{5}{72}K_0 + \frac{1}{12}K_1 + \frac{1}{9}K_2 - \frac{1}{8}K_3 \end{aligned} \tag{20}$$

if $|LTE| < ErrorTolerance$, $x_{t_{n+1}} = \tilde{x}_{t_{n+1}}$, otherwise, fail. If this step is successful, the next integration step size is predicted by:

$$h_{n+2} = h_{n+1} \cdot max(0.5, 0.8 \cdot \sqrt[3]{(ErrorTolerance)/|LTE|}) \tag{21}$$

4. Trapezoidal Rule solver:

$$\begin{aligned} x_{t_{n+1}} &= x_{t_n} + \frac{h_{n+1}}{2}(\dot{x}_{t_n} + \dot{x}_{t_{n+1}}) \\ &= x_{t_n} + \frac{h_{n+1}}{2}(\dot{x}_{t_n} + f(x_{t_{n+1}}, u_{t_{n+1}}, t_{n+1})) \end{aligned} \quad . \tag{22}$$

Among these solvers, 1) and 3) are explicit; 2) and 4) are implicit. Also, 1) and 2) do not perform step size control, so are called fixed-step-size solvers; 3) and 4) change step sizes according to error estimation, so are called variable-step-size solvers. Variable-step-size solvers adapt the step sizes according to changes of the system flow, thus are “smarter” than fixed-step-size solvers.

2.2.4 Discontinuity

The existence and uniqueness of the solution of an ODE (Theorem 1 in Appendix A) allows the right-hand side of (3) to be discontinuous at a countable number of discrete points *D*, which are called the *breakpoints* (also called the *discontinuous points* in some literature). These breakpoints may be caused by the discontinuity of input signal *u*, or by the intrinsic flow of *f*. At these points, the solutions are defined based on a discrete-event semantics and achieved by performing discrete-phase executions. The solutions defined on the left and right limits are solved with the normal continuous-time equations based on the ODE.

One impact of breakpoints on ODE solvers is that history solutions are useless when approximating the derivative of *x* after the breakpoints. The solver should resolve the new initial conditions and start the solving process as if it is at a starting point. So, the discretization of time should step exactly on breakpoints for the left limit, and start at the breakpoint again after finding the right limit.

A breakpoint may be known beforehand, in which case it is called a *predictable breakpoint*. For example, a square wave source actor knows its next flip time. This information can be used to control the discretization of time. A breakpoint can also be *unpredictable*, which means it is unknown until the time it occurs. For example, an actor that varies its functionality when the input signal crosses a threshold can only report a "missed" breakpoint after an integration step is finished. How to handle breakpoints correctly is a big challenge for integrating continuous-time models with discrete models like DE and FSM.

2.2.5 Breakpoint ODE Solvers

Breakpoints in the CT domain are handled by adjusting integration steps. We use a table to handle predictable breakpoints, and use the step size control mechanism to handle unpredictable breakpoints. The breakpoint handling are transparent to users, and the implementation details (provided in section 2.8.4) are only needed when developing new directors, solvers, or event generators.

Since the history information is useless at breakpoints, special ODE solvers are designed to restart the numerical integration process. In particular, we have implemented the following breakpoint ODE solvers.

1. DerivativeResolver:

   It calculates the derivative of the current state, i.e. $\frac{dx}{dt}$. This is simply done by evaluation the right-hand side of (3). At breakpoints, this solver is used for the first step to generate history information for explicit methods or one step methods.

2. ImpulseBESolver:

$$\tilde{x}_{t_{n+1}} = x_{t_n} + h_{n+1} \cdot \hat{x}_{t_{n+1}} \qquad (23)$$
$$x_{t_n^+} = \tilde{x}_{t_{n+1}} - h_{n+1} \cdot \dot{x}_{t_n^+}$$

The two time points t_n and t_n^+ have the same time value. This solver is used for breakpoints at which a Dirac impulse signal appears.

Notice that none of these solvers advance time. They can only be used at breakpoints.

2.3 Signal Types

The CT domain of Ptolemy II supports continuous time mixed-signal modeling. As a consequence, there could be two types of signals in a CT model: continuous signals and discrete events. Note that for both types of signals, time is continuous. These two types of signals directly affect the behavior of a receiver that contains them. A continuous CTReceiver contains a sample of a continuous signal at the current time. Reading a token from that receiver will not consume the token. A discrete CTReceiver may or may not contain a discrete event. Reading from a discrete CTReceiver with an event will consume the event, so that events are processed exactly once[1]. Reading from an empty discrete CTReceiver is not allowed.

Note that some actors can be used to compute on both continuous and discrete signals. For example, an adder can add two continuous signals, as well as two sets of discrete events. Whether a particular link among actors is continuous or discrete is resolved by a *signal type system*. The signal type system understands signal types on specific actors (indicated by the interfaces they implement or the parameters specified on their ports), and try to resolve signal types on the ports of domain polymorphic actors.

The signal type system in the CT domain works on a simple lattice of signal types, shown in Figure 2.4. A type lower in the lattice is more specific than a type higher in the lattice. A CT model is *well-defined* and executable, if and only if all ports are resolved to either CONTINUOUS or DISCRETE. Some actors have their signal types fixed. For example, an `Integrator` has a CONTINUOUS input and a CONTINUOUS output; a `PeriodicSampler` has a CONTINUOUS input and a DISCRETE output; a `TriggeredSampler` has one CONTINUOUS input (the input), one DISCRETE input (the trigger), and a DISCRETE output; and a `ZeroOrderHold` has a DISCRETE input and a CONTINUOUS output. For domain polymorphic actors that implement the `SequenceActor` interface, i.e. they operate solely on sequences of tokens, their inputs and outputs are treated as DISCRETE. For other domain polymorphic actors that can operate on both continuous and discrete signals, the signal type on their ports are initially UNRESOLVED. The signal type system will resolve and check signal types of ports according to the following two rules:

- If a port *p* is connected to another port *q* with a more specific type, then the type of *p* is resolved to that of the port *q*. If *p* is CONTINUOUS but *q* is DISCRETE, then both of them are resolved to

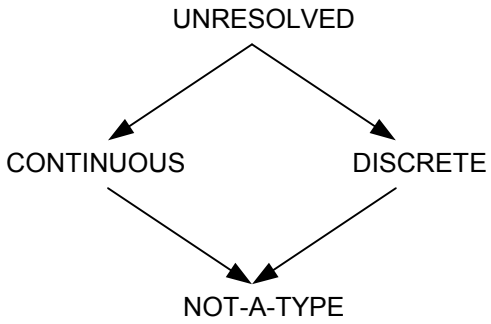


FIGURE 2.4. A signal type lattice.

1. This distinction of receivers is also called state and event semantics in some literatures [68].

NOT-A-TYPE.

- Unless otherwise specified, the types of the input ports and output ports of an actor are the same.

At the end of the signal-type resolution, if any port is of type UNRESOLVED or NOT-A-TYPE, then the topology of the system is illegal, and the execution is denied.

The signal type of a port can also be forced by adding an parameter "signalType" to the port. The signal type system will recognize this parameter and resolve other types accordingly. To add this parameter, right click on the port, select Configure, then add a parameter with the name signalType and the value of a string of either "CONTINUOUS" or "DISCRETE", noting the quotation marks.

Signal types may be more trickier at the boundaries of opaque composite actors than within a CT model. Because of the information hiding, it may not be obvious which port of another level of hierarchy is continuous and which port is discrete. In the CT domain, we follow these rules to resolve signal types for composite ports:

- A `TypedCompositeActor` within a CT model is always treated as entirely discrete. Within a CT model, for any opaque composite actor that may contain continuous dynamics at a deeper level, use the `CTCompositeActor` (listed in the actor library as "continuous time composite actor" in domain specific actors) or the modal model composite actor.
- For a `CTCompositeActor` or a modal model within a CT model, all its ports are treated as continuous by default. To allow a discrete event going through the composite actor boundary, manually set the signal type of that port by adding the signalType parameter.
- For a `TypedCompositeActor` *containing* a CT model, all the ports of the `TypedCompositeActor` are treated as discrete, and the CT director to use is the CTMixedSignalDirector (listed as CTDirector in the vergil director library).
- For a `CTCompositeActor` or a modal model *containing* a CT model, all the signal types of the ports of the container are treated as continuous, and can be set by adding the signalType parameter. The CTDirector to use in this situation is the CTEmbeddedDirector.

2.4 CT Actors

A CT system can be built up using actors in the ptolemy.domains.ct.lib package and domain polymorphic actors that have continuous behaviors (i.e. all actors that do not implement the SequenceActor interface). The key actor in CT is the integrator. It serves the unique role of wiring up ODEs. Other actors in a CT system are usually stateless. A general understanding is that, in a pure continuous-time model, all the information — the state of the system— is stored in the integrators.

2.4.1 CT Actor Interfaces

In order to schedule the execution of actors in a CT model and to support the interaction between CT and other domains (which are usually discrete), we provide the following interfaces.

- **CTDynamicActor**. Dynamic actors are actors that contains continuous dynamics in their I/O path. An integrator is a dynamic actor, and so are all actors that have integration relations from their inputs to their outputs.
- **CTEventGenerator**. *Event generators* are actors that convert continuous time input signals to discrete output signals.
- **CTStatefulActor**. Stateful actors are actors that have internal states. The reason to classify this kind of actor is to support rollback, which may happen when a CT model is embedded in a discrete

event model.

- **CTStepSizeControlActor**. Step size control actors influence the integration step size by telling the director whether the current step is accurate. The accuracy is in the sense of both tolerable numerical errors and absence of unpredictable breakpoints. It may also provide information about refining a step size for an inaccurate step and suggesting the next step size for an accurate step.
- **CTWaveformGenerator**. *Waveform generators* are actors that convert discrete input signals to continuous-time output signals.

Strictly speaking, event generators and waveform generators do not belong to any domain, but the CT domain is design to handle them intrinsically. When building systems, CT parts can always provide discrete interface to other domains.

Neither a loop of dynamic actors nor a loop of non-dynamic actors are allowed in a CT model. They introduce problems about the order that actors be executed. A loop of dynamic actors can be easily broken by a Scale actor with scale 1. A loop of non-dynamic actors builds an algebraic equation. The CT domain does not support modeling algebraic equations, yet.

2.4.2 Actor Library

CTPeriodicalSampler. This event generator periodically samples the input signal and generates events with the value of the input signal at these time points. The sampling rate is given by the *samplePeriod* parameter, which has default value 0.1. The sampling time points, which are known beforehand, are examples of predictable breakpoints.

CTTriggeredSampler. This actor samples the continuous input signal when there is a discrete event present at the "trigger" input.

ContinuousTransferFunction. A transfer function in the continuous time domain. This actor implements a transfer function where the single input (u) and single output (y) can be expressed in (Laplace) transfer function form as the following equation:

$$\frac{Y(s)}{U(s)} = \frac{b_1 s^{m-1} + b_2 s^{m-2} + \ldots + b_m}{a_1 s^{n-1} + a_2 s^{n-2} + \ldots + a_n} \tag{24}$$

where m and n are the number of numerator and denominator coefficients, respectively. This actors has two parameters – *numerator* and *denominator* – containing the coefficients of the numerator and denominator in descending powers of s. The parameters are double arrays. The order of the denominator (n) must be greater than or equal to the order of the numerator (m).

DifferentialSystem. The differential system model implements a system whose behavior is defined by:

$$\begin{aligned} \dot{x} &= f(x, u, t) \\ y &= g(x, u, t) \\ x(t_0) &= x_0 \end{aligned} \tag{25}$$

where x is the state vector, u is the input vector, and y is the output vector, t is the time. Users must give the name of the variables by filling in the parameter and add ports with proper names. The actor, upon creation, has no inputs and no outputs. After creating proper ports, their names can be used in the

expressions of state equations and output equations. The name of the state variables are manually added by filling in the *stateVariableNames* parameter.

The state equations and output maps must be manually created by users as parameters. If there are *n* state variables $x_1 \ldots x_n$ then users need to create *n* additional parameters, one for each state equation. And the parameters must be named as x_1_dot, ..., x_n_dot, respectively. Similarly, if the output ports have names $y_1 \ldots y_r$, then users must create additional *r* parameters for output maps. These parameters should be named y_1, ..., y_r, respectively.

Integrator: The integrator for continuous-time simulation. An integrator has one input port and one output port. Conceptually, the input is the derivative of the output, and an ordinary differential equation is modeled as an integrator with feedback.

An integrator is a dynamic, step-size-control, and stateful actor. To help resolve new states from previous states, a set of variables are used:

- *state and its derivative*: These are the new state and its derivative at a time point, which have been confirmed by all the step size control actors.
- *tentative state and tentative derivative*: These are the state and derivative which have not been confirmed. It is a starting point for other actors to estimate the accuracy of this integration step.
- *history*: The previous states and derivatives. An integrator remembers the history states and their derivatives for the past several steps. The history is used by multistep methods.

An integrator has one parameter: *initialState*. At the initialization stage of the simulation, the state of the integrator is set to the initial state. Changes of *initialState* will be ignored after the simulation starts, unless the `initialize()` method of the integrator is called again. The default value of this parameter is 0.0. An integrator can possibly have several auxiliary variables. These auxiliary variables are used by ODE solvers to store intermediate states for individual integrators.

The CTBaseIntegrator class, which is the super class of Integrator overrides the pruneDependencies() method of the AtomicActor class to issue the following statement:

```
removeDependency(input, output);
```

This statement declares that the output from an integrator from a firing does not depend on its input. The CT scheduler does not use this information to construct an execution schedule. However, when a CT model works as a subsystem in a DE model, this dependency information is used to help the DE director to construct a DE schedule and assign priorities (execution orders).

LinearStateSpace. The State-Space model implements a system whose behavior is defined by:

$$\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx + Du \\ x(t_0) &= x_0 \end{aligned} \tag{26}$$

where *x* is the state vector, *u* is the input vector, and *y* is the output vector. The matrix coefficients must have the following characteristics:

- A must be an *n*-by-*n* matrix, where n is the number of states.
- B must be an *n*-by-*m* matrix, where m is the number of inputs.

- C must be an *r*-by-*n* matrix, where r is the number of outputs.
- D must be an *r*-by-*m* matrix.

The actor accepts *m* inputs and generates *r* outputs through a multiple input port and a multiple output port. The widths of the ports must match the number of rows and columns in corresponding matrices, otherwise, an exception will be thrown.

ZeroCrossingDetector. This is an event generator that monitors the signal coming in from an input port – trigger. If the trigger is zero, then output the token from the input port. Otherwise, there is no output. This actor controls the integration step size to accurately resolve the time that the zero crossing happens. It has a parameter, *errorTolerance*, which controls how accurately the zero crossing is determined.

ZeroOrderHold. This is a waveform generator that converts discrete events into continuous signals. This actor acts as a zero-order hold. It consumes the token when the `consumeCurrentEvent()` is called. This value will be held and emitted every time it is fired, until the next time `consumeCurrentEvent()` is called. This actor has one single input port, one single output port, and no parameters.

ThresholdMonitor. This actor controls the integration steps so that the given threshold (on the input) is not crossed in one step. This actor has one input port and one output port. It has two parameters *thresholdWidth* and *thresholdCenter*, which have default value 1e-2 and 0, respectively. If the input is within the range defined by the threshold center and threshold width, then a true token is emitted from the output.

2.4.3 Domain Polymorphic Actors

Not all domain polymorphic actors can be used in the CT domain. Whether an actor can be used depends on how the internal states of the actor evolve when executing.

- **Stateless actors**: All stateless actors can be used in CT. In fact, most CT systems are built by integrators and stateless actors.
- **Timed actors**: Timed actors change their states according to the notion of time in the model. All actors that implement the TimedActor interface can be used in CT, as long as they do not also implement SequenceActor. Timed actors that can be used in CT include plotters that are designed to plot timed signals.
- **Sequence actors**: Sequence actors change their states according to the number of input tokens received by the actor and the number of times that the actor is postfired. Since CT is a time driven model, rather than a data driven model, the number of received tokens and the number of postfires do not have a significant semantic meaning. So, none of the sequence actors can be used in the CT domain. For example, the Ramp actor in Ptolemy II changes its state –– the next token to emit –– corresponding to the number of times that the actor is postfired. In CT, the number of times that the actor is postfired depends on the discretization of time, which further depend on the choice of ODE solvers and setting of parameters. As a result, the slope of the ramp may not be a constant, and this may lead to very counterintuitive models. The same functionality is replaced by a CurrentTime actor and a Scale actor. If sequence behaviors are indeed required, event generators and waveform generators may be helpful to convert continuous and discrete signals.

2.5 CT Directors

There are three CT directors –– CTMultiSolverDirector, CTMixedSignalDirector, and CTEmbeddedDirector. The first one can only serve as a top-level director, a CTMixedSignalDirector can be used both at the top-level or inside a composite actor, and a CTEmbeddedDirector can only be contained in a CTCompositeActor. In terms of mixing models of computation, all the directors can execute composite actors that implement other models of computation, as long as the composite actors are properly connected (see section 2.6). Only CTMixedSignalDirector and CTEmbeddedDirector can be contained by other domains. The outside domain of a composite actor with CTMixedSignalDirector can be any discrete domain, such as DE, DT, etc. The outside domain of a composite actor with CTEmbeddedDirector must also be CT or FSM, if the outside domain of the FSM model is CT. (See also the HSDirector in the FSM domain.)

2.5.1 ODE Solvers

There are six ODE solvers implemented in the ptolemy.domains.ct.kernel.solver package. Some of them are specific for handling breakpoints. These solvers are ForwardEulerSolver, BackwardEulerSolver, ExplicitRK23Solver, TrapezoidalRuleSolver, DerivativeResolver, and ImpulseBESolver. They implement the ODE solving algorithms in section 2.2.3 and section 2.2.5, respectively.

2.5.2 CT Director Parameters

The CTDirector base class maintains a set of parameters which controls the execution. These parameters, shared by all CT directors, are listed in Table 10 on page 34. Individual directors may have their own (additional) parameters, which will be discussed in the appropriate sections.

Table 10: CTDirector Parameters

Name	Description	Type	Default Value
errorTolerance	The upper bound of local errors. Actors that perform integration error control (usually integrators in variable step size ODE solving methods) will compare the estimated local error to this value. If the local error estimation is greater than this value, then the integration step is considered inaccurate, and should be restarted with a smaller step sizes.	double	1e-4
initStepSize	This is the step size that users specify as the desired step size. For fixed step size solvers, this step size will be used in all non-breakpoint steps. For variable step size solvers, this is only a suggestion.	double	0.1
maxIterations	This is used to avoid the infinite loops in (implicit) fixed-point iterations. If the number of fixed-point iterations exceeds this value, but the fixed point is still not found, then the fixed-point procedure is considered failed. The step size will be reduced by half and the integration step will be restarted.	int	20
maxStepSize	The maximum step size used in a simulation. This is the upper bound for adjusting step sizes in variable step-size methods. This value can be used to avoid sparse time points when the system dynamic is simple.	double	1.0
minStepSize	The minimum step size used in a simulation. This is the lower bound for adjusting step sizes. If this step size is used and the errors are still not tolerable, the simulation aborts. This step size is also used for the first step after breakpoints.	double	1e-5

Table 10: CTDirector Parameters

Name	Description	Type	Default Value
startTime	The start time of the simulation. This is only applicable when CT is the top level domain. Otherwise, the CT director follows the time of its executive director.	double	0.0
stopTime	The stop time of the simulation. This is only applicable when CT is the top level domain. Otherwise, the CT director follows the time of its executive director.	double	Double. MAX_ VALUE
synchronizeTo-RealTime	Indicate whether the execution of the model is synchronized to real time at best effort.	boolean	false
timeResolution	This controls the comparison of time. Since time in the CT domain is a double precision real number, it is sometimes impossible to reach or step at a specific time point. If two time points are within this resolution, then they are considered identical.	double	1e-10
valueResolution	This is used in (implicit) fixed-point iterations. If in two successive iterations the difference of the states is within this resolution, then the integration step is called converged, and the fixed point is considered reached.	double	1e-6

2.5.3 CTMultiSolverDirector

A CTMultiSolverDirector has two ODE solvers — one for ordinary use and one specifically for breakpoints. Thus, besides the parameters in the CTDirector base class, this class adds two more parameters as shown in Table 11 on page 35.

Table 11: Additional Parameter for CTMultiSolverDirector

Name	Description	Type	Default Value
ODESolver	The fully qualified class name for the ODE solver class.	string	“ptolemy.domains.ct.kernel.solver.ForwardEulerSolver”
breakpointODESolver	The fully qualified class name for the breakpoint ODE solver class.	string	“ptolemy.domains.ct.kernel.solver.DerivativeResolver”

A CTMultiSolverDirector can direct a model that has composite actors implementing other models of computation. One simulation iteration is done in two phases: the continuous phase and the discrete phase. Let the current iteration be *n.* In the continuous phase, the differential equations are integrated from time t_{n-1} to t_n. After that, in the discrete phase, all (discrete) events which happen at t_n are processed. The step size control mechanism will assure that no events will happen between t_{n-1} and t_n.

2.5.4 CTMixedSignalDirector

This director is designed to be the director when a CT subsystem is contained in an event-based system, like DE or DT. As proved in [94], when a CT subsystem is contained in the DE domain, the CT subsystem should run ahead of the global time, and be ready for rollback. This director implements this optimistic execution.

Since the outside domain is event-based, each time the embedded CT subsystem is fired, the input data are events. In order to convert the events to continuous signals, breakpoints have to be introduced. So this director extends CTMultiSolverDirector, which always has two ODE solvers. There is one

more parameter used by this director — the *runAheadLength*, as shown in Table 12 on page 36.

Table 12: Additional Parameter for CTMixedSignalDirector

Name	Description	Type	Default Value
runAheadLength	The maximum length of time for the CT subsystem to run ahead of the global time.	double	1.0

When the CT subsystem is fired, the CTMixedSignalDirector will get the current time τ and the next iteration time τ' from the outer domain, and take the $\min(\tau - \tau', l)$ as the fire end time, where l is the value of the parameter *maxRunAheadLength*. The execution lasts as long as the fire end time is not reached or an output event is not detected.

This director supports rollback; that is when the state of the continuous subsystem is confirmed (by knowing that no events with a time earlier than the CT current time will be present), the state of the system is marked. If an optimistic execution is known to be wrong, the state of the CT subsystem will roll back to the latest marked state.

This director can also be used at the top level of a model, which works just as a CTMultipleSolverDirector.

2.5.5 CTEmbeddedDirector

This director is used when a CT subsystem is embedded in another continuous time system, either directly or through a hierarchy of finite state machines, like in the hybrid system scenario [96]. This director can pass step size control information up to its executive director. To achieve this, the director must be contained in a CTCompositeActor, which implements the CTStepSizeControlActor interface and can pass the step size control information from the inner domain to the outer domain.

This director extends CTMultiSolverDirector, with no additional parameters. A major difference between this director and the CTMixedSignalDirector is that this director does not support rollback. In fact, when a CT subsystem is embedded in a continuous-time environment, rollback is not necessary.

2.6 Interacting with Other Domains

The CT domain can interact with other domains in Ptolemy II. In particular, we consider interaction among the CT domain, the discrete event (DE) domain and the finite state machine (FSM) domain. Following circuit design communities, we call a composition of CT and DE a *mixed-signal model*; following control and computation communities, we call a composition of CT and FSM a *hybrid system model*.

There are two ways to put CT and DE models together, depending on the containment relation. In either case, event generators and waveform generators are used to convert the two types of signals. Figure 2.5 shows a DE component wrapped by an event generator and a waveform generator. From the input/output point of view, it is a continuous time component. Figure 2.6 shows a CT subsystem wrapped by a waveform generator and an event generator. From the input/output point of view, it is a discrete event component. Notice that event generators and waveform generators always stay in the CT domain.

A hierarchical composition of FSM and CT is shown in figure 2.7. A CT component, by adopting the event generation technique, can have both continuous and discrete signals as its output. The FSM

can use predicates on these signals, as well as its own input signals, to build trigger conditions. The actions associated with transitions are usually setting parameters in the destination state, including the initial conditions of integrators.

2.7 CT Domain Demos

Here are some demos in the CT domain showing how this domain works and the interaction with other domains.

2.7.1 Lorenz System

The Lorenz System (see, for example, pp. 213-214 in [38]) is a famous nonlinear dynamic system that shows chaotic attractors. The system is given by:

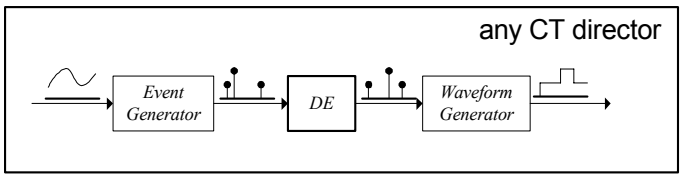


FIGURE 2.5. Embedding a DE component in a CT system.

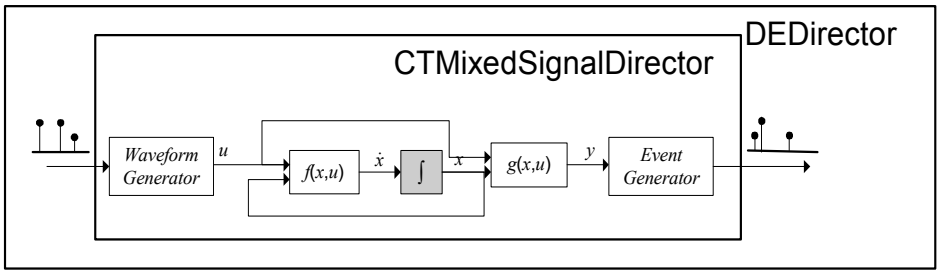


FIGURE 2.6. Embedding a CT component in a DE system.

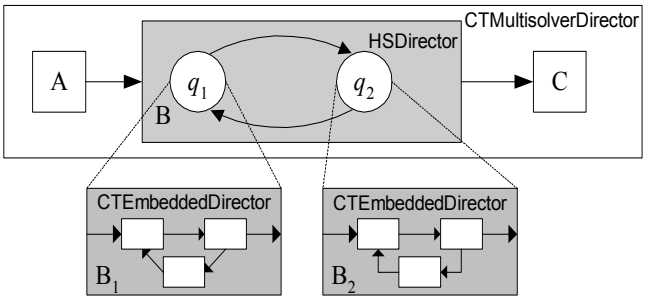


FIGURE 2.7. Hybrid system modeling.

$$\begin{aligned} \dot{x}_1 &= \sigma(x_2 - x_1) \\ \dot{x}_2 &= (\lambda - x_3)x_1 - x_2 \\ \dot{x}_3 &= x_1 \cdot x_2 - b \cdot x_3 \end{aligned} \tag{27}$$

The system is built by integrators and stateless domain polymorphic actors, as shown in figure 2.8.

The result of the state trajectory projecting onto the (x_1, x_2) plane is shown in figure 2.9. The initial conditions of the state variables are all 1.0. The default value of the parameters are: $\sigma = 1, \lambda = 25, b = 2.0$.

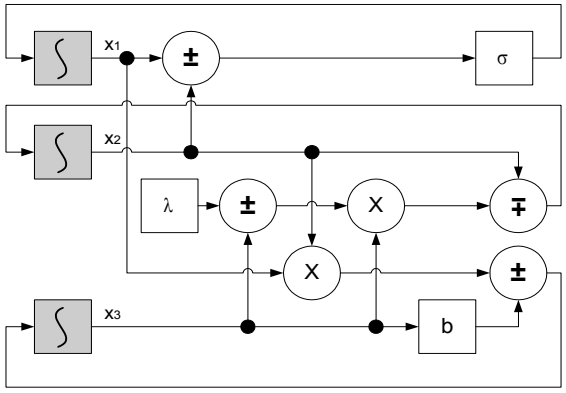


FIGURE 2.8. Block diagram for the Lorenz system.

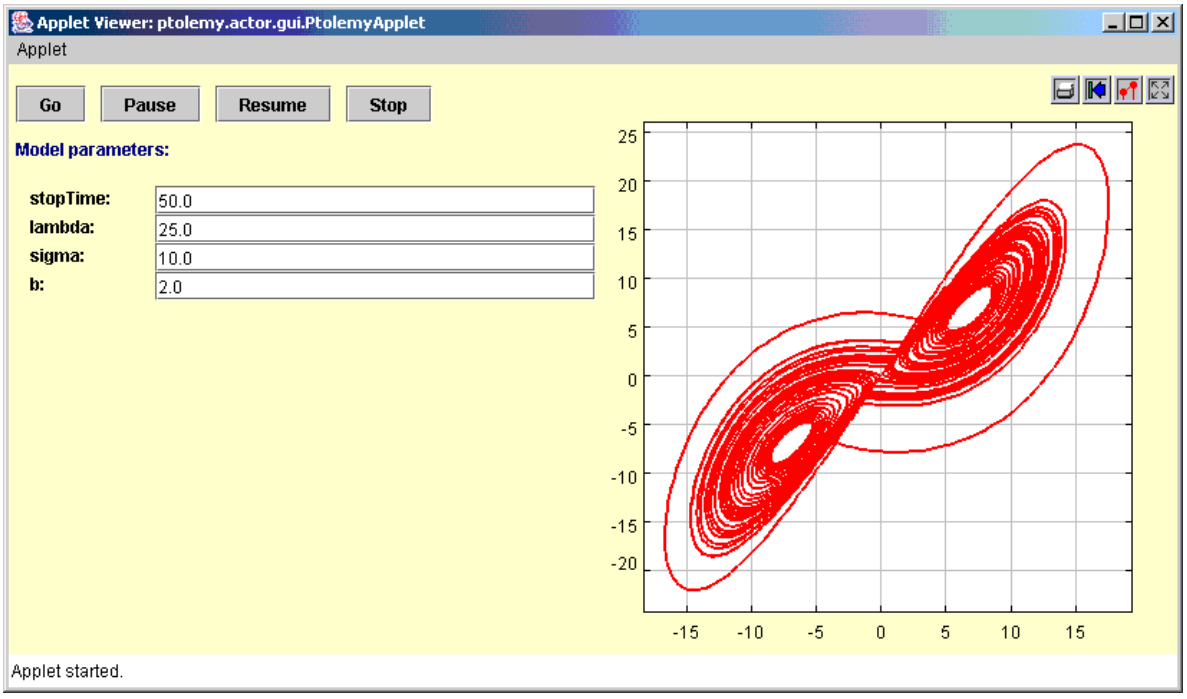


FIGURE 2.9. The simulation result of the Lorenz system.

2.7.2 Microaccelerometer with Digital Feedback.

Microaccelerometers are MEMS devices that use beams, gaps, and electrostatics to measure acceleration. Beams and anchors, separated by gaps, form parallel plate capacitors. When the device is accelerated in the sensing direction, the displacement of the beams causes a change of the gap size, which further causes a change of the capacitance. By measuring the change of capacitance (using a capacitor bridge), the acceleration can be obtained accurately. Feedback can be applied to the beams by charging the capacitors. This feedback can reduce the sensitivity to process variations, eliminate mechanical resonances, and increase sensor bandwidth, selectivity, and dynamic range.

Sigma-delta modulation [24], also called pulse density modulation or a bang-bang control, is a digital feedback technique, which also provides the A/D conversion functionality. Figure 2.10 shows the conceptual diagram of system. The central part of the digital feedback is a one-bit quantizer.

We implemented the system as Mark Alan Lemkin designed [85]. As shown in the figure 2.11, the second order CT subsystem is used to model the beam. The voltage on the beam-gap capacitor is sampled every T seconds (much faster than the required output of the digital signal), then filtered by a lead compensator (FIR filter), and fed to an one-bit quantizer. The outputs of the quantizer are converted to force and fed back to the beams. The outputs are also counted and averaged every NT seconds to produce the digital output. In our example, the external acceleration is a sine wave.

The execution result of the microaccelerometer system is shown in figure 2.12. The upper plot in the figure plots the continuous signals, where the low frequency (blue) sine wave is the acceleration

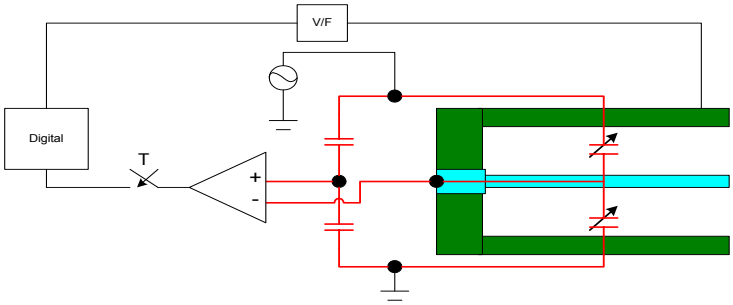


FIGURE 2.10. Micro-accelerator with digital feedback

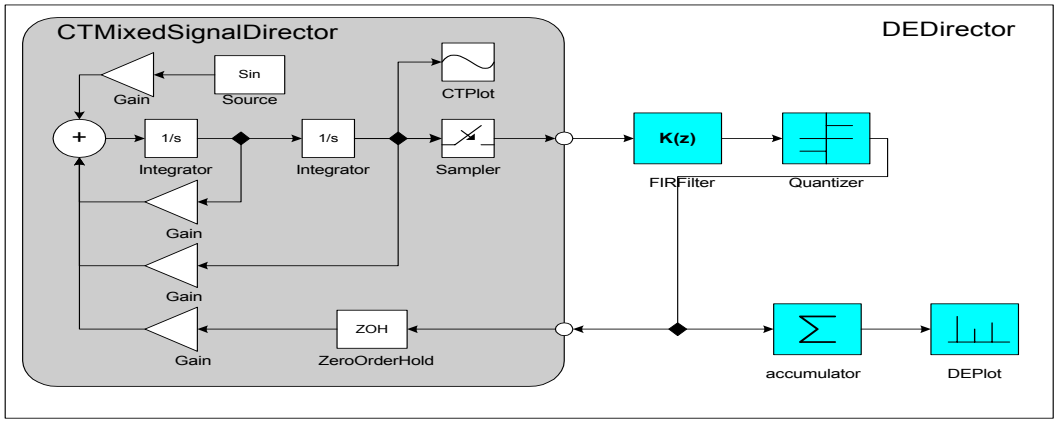


FIGURE 2.11. Block diagram for the micro-accelerator system.

input, the high frequency waveform (red) is the capacitance measurement, and the squarewave (green) is the zero-order hold of the feedback from the digital part. In the lower plot, the dense events (blue) are the quantized samples of the capacitance measurements, which has value +1 or -1, and the sparse events (red) are the accumulation and average of the previous 64 quantized samples. The sparse events are the digital output, and as expected, they have a sinsoidal shape.

2.7.3 Sticky Point Masses System

This sticky point mass demo shows a simple hybrid system. As shown in figure 2.13, there are two point masses on a frictionless table with two springs attaching them to fixed walls. Given initial positions other than the equilibrium points, the point masses oscillate. The distance between the two walls are close enough that the two point masses may collide. The point masses are sticky, in the way so that when they collide, they will sticky together and become one point mass with two springs attached to it. We also assume that the stickiness decays exponentially after the collision, such that eventually the

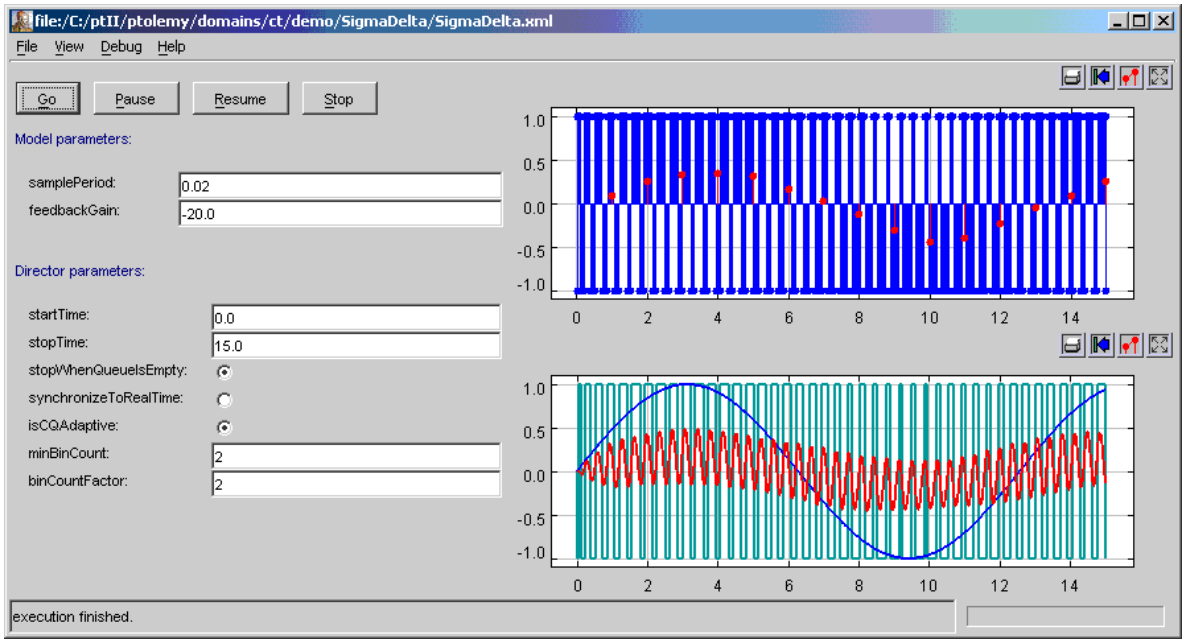


FIGURE 2.12. Execution result of the microaccelerometer system.

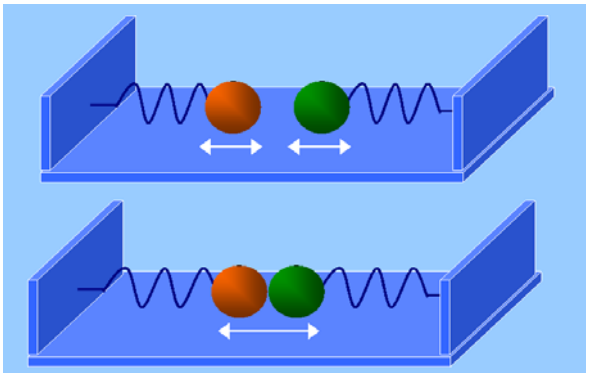


FIGURE 2.13. Sticky point masses system

pulling force between the two springs is big enough to pull the point masses apart. This separation gives the two point masses a new set of initial positions, and they oscillate freely until they collide again.

The system model, as shown in figure 2.14, has three levels of hierarchy –– CT, FSM, and CT. The top level is a continuous time model with two actors, a CTCompositeActor that outputs the position of the two point masses, and a plotter that simply plots the trajectories. The composite actor is a finite state machine with two modes, *separated* and *together*.

In the separated state, there are two differential equations modeling two independently oscillating point masses. There is also an event detection mechanism, implemented by subtracting one position from another and comparing the result to zero. If the positions are equal, within a certain accuracy, then the two point masses collide, and a collision event is generated. This event will trigger a transition from the separated state to the together state. And the actions on the transition set the velocity of the stuck point mass based on Law of Conservation of Momentum.

In the together state, there is one differential equation modeling the stuck point masses, and another first order differential equation modeling the exponentially decaying stickiness. There is another expression computing the pulling force between the two springs. The guard condition from the together state to the separated state compares the pulling force to the stickiness. If the pulling force is bigger than the stickiness, then the transition is taken. The velocities of the two separated point masses equal to their velocities before the separation. The simulation result is shown in figure 2.15, where the position of the two point masses are plotted.

2.8 Implementation

The CT domain consists of the following packages, ct.kernel, ct.kernel.util, ct.kernel.solver, and ct.lib, as shown in figure 2.16.

2.8.1 ct.kernel.util package

The ct.kernel.util package provides a basic data structure — `TotallyOrderedSet`, which is used to store breakpoints. The UML for this package is shown in figure 2.17. A totally ordered set is a set (i.e. no duplicated elements) in which the elements are totally comparable. This data structure is used to store breakpoints since breakpoints are processed in their chronological order.

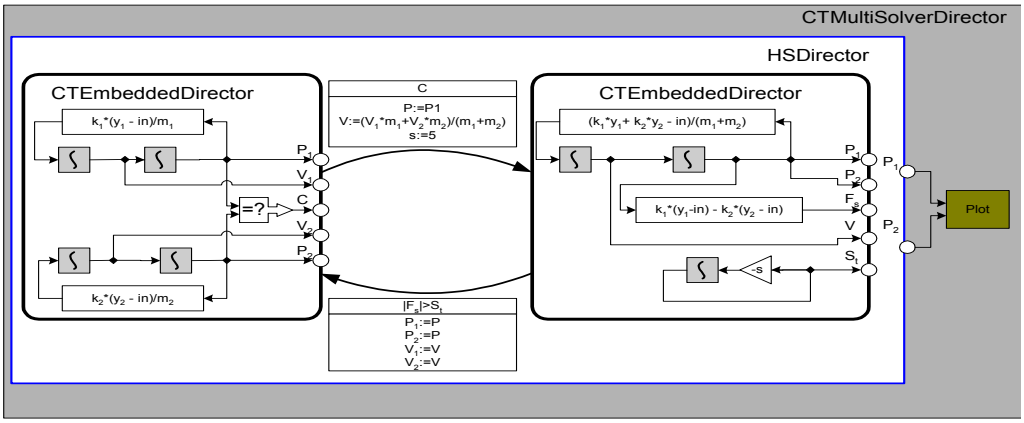


FIGURE 2.14. Modeling sticky point masses.

2.8.2 ct.kernel package

The ct.kernel package is the key package of the CT domain. It provides interfaces to classify actors, scheduler, director, and a base class for ODE solvers. The interfaces are used by the scheduler to generate schedules. The classes, including the `CTBaseIntegrator` class and the `ODESolver` class, are shown in figure 2.18. Here, we use the delegation and the strategy design patterns [48][42] in the `CTBaseIntegrator` and the `ODESolver` classes to support seamlessly changing ODE solvers without reconstructing integrators. The execution methods of the `CTBaseIntegrator` class are delegated to the `ODESolver` class, and subclasses of `ODESolver` provide the concrete implementations of these methods, depending on the ODE solving algorithms.

CT directors implement the semantics of the continuous time execution. As shown in figure 2.19, directors that are used in different scenarios derive from the `CTDirector` base class. The `CTScheduler` class provides schedules for the directors.

The ct.kernel.solver package provides a set of ODE solvers. The classes are shown in figure 2.20.

FIGURE 2.15. The simulation result of the sticky point masses system.

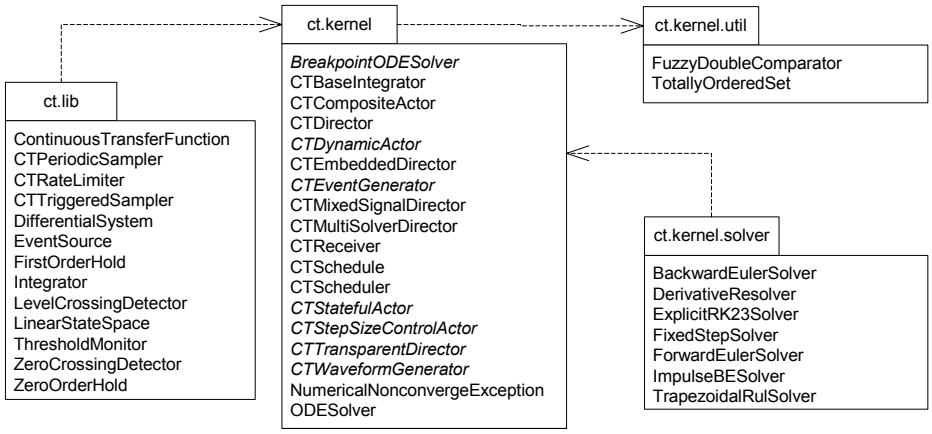


FIGURE 2.16. The packages in the CT domain.

In order for the directors to choose among ODE solvers freely during the execution, the strategy design pattern is used again. A director class talks to the abstract `ODESolver` base class and individual ODE solver classes extend the `ODESolver` to provide concrete strategies.

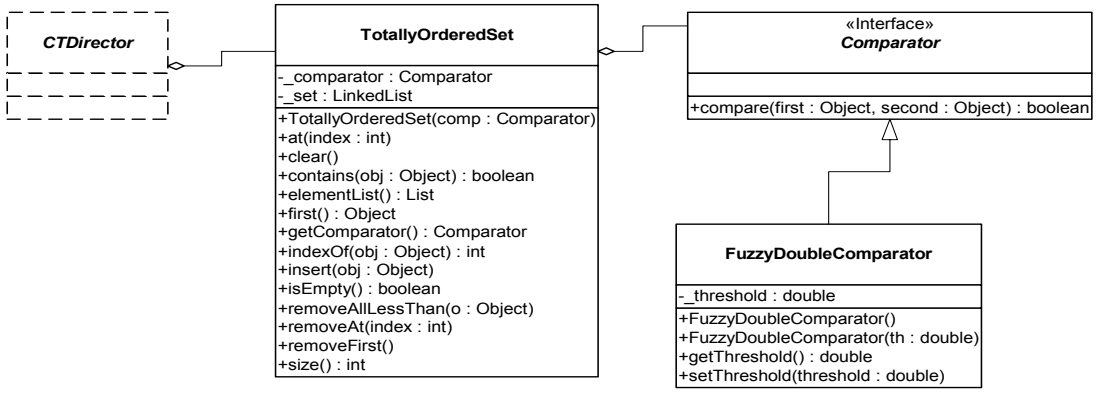


FIGURE 2.17. UML for ct.kernel.util package.

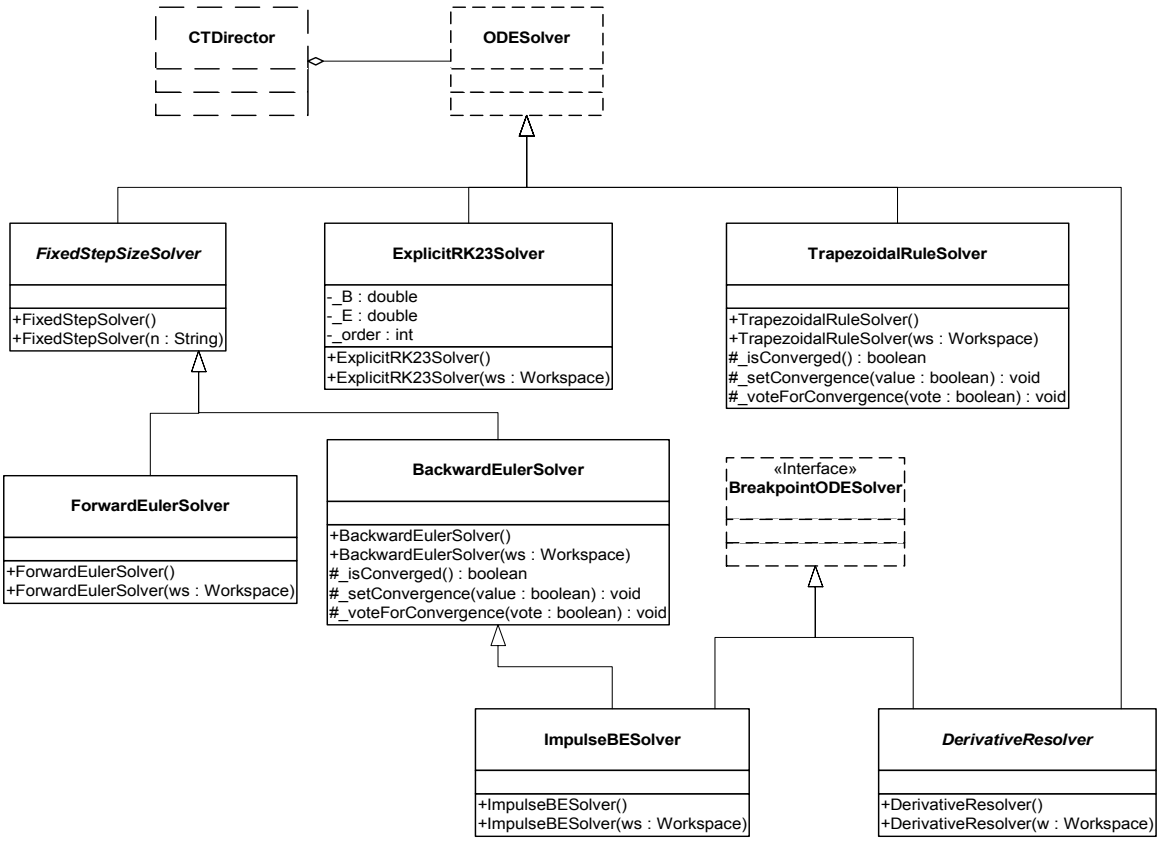


FIGURE 2.20. UML for ct.kernel.solver package.

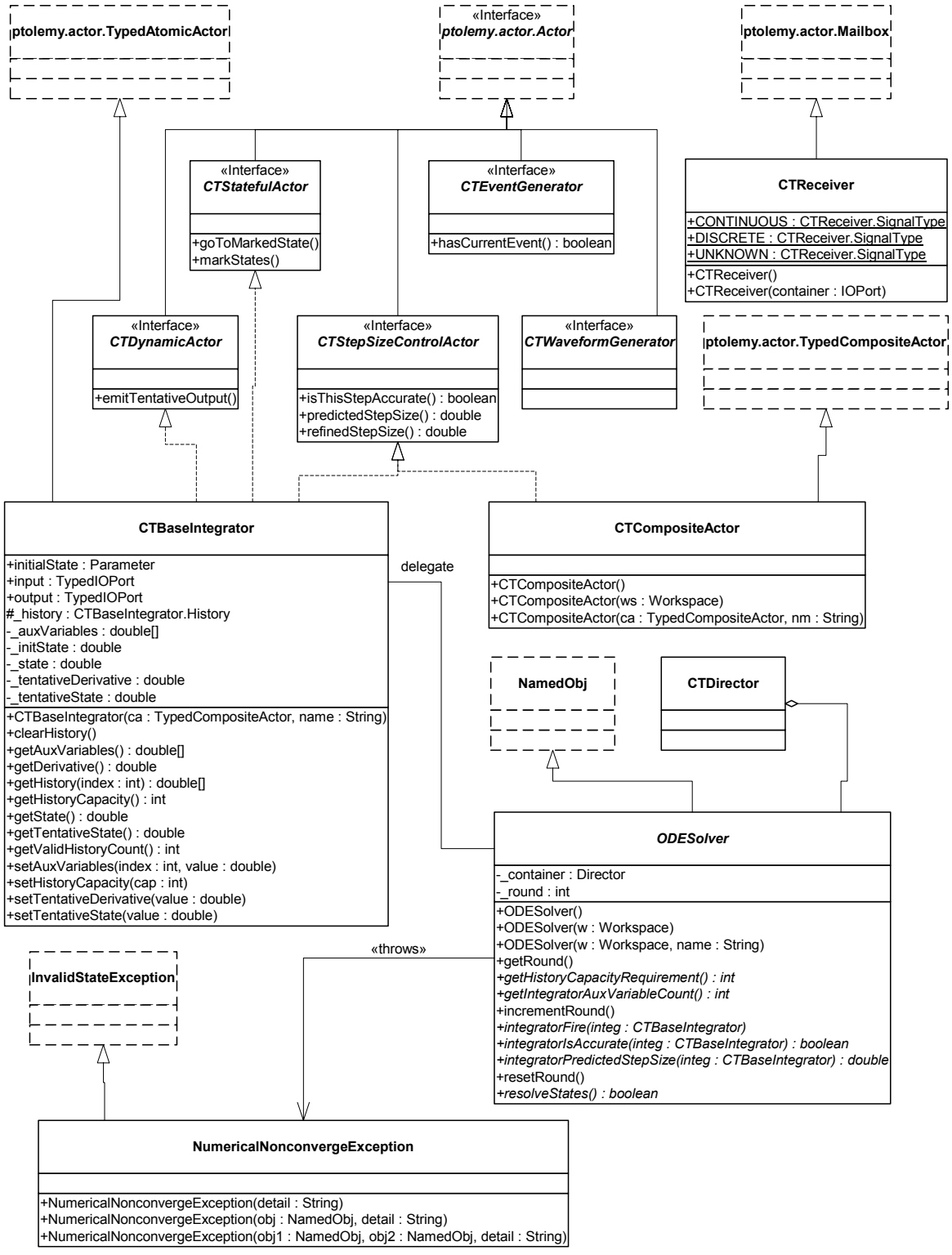


FIGURE 2.18. UML for ct.kernel package, actor related classes.

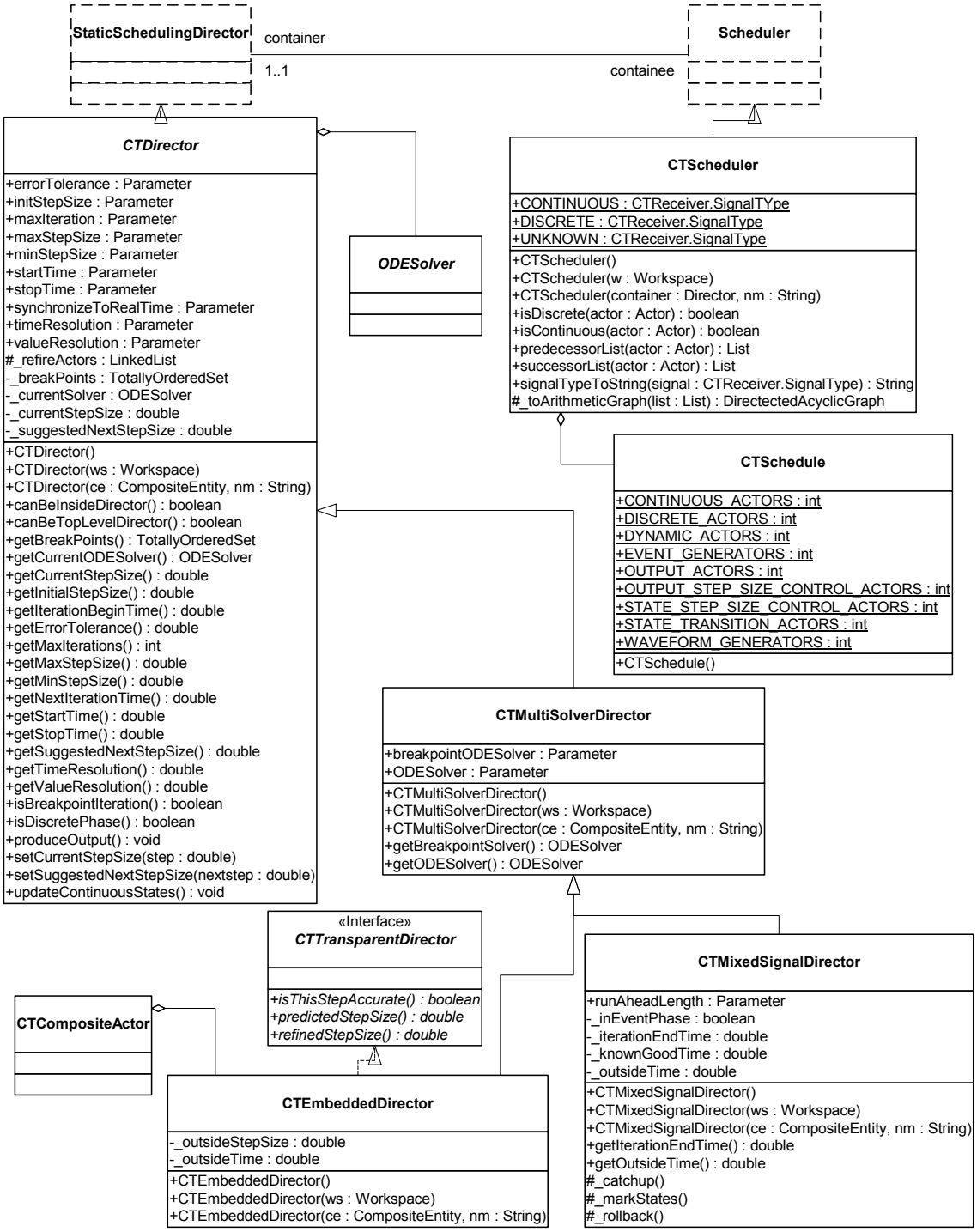


FIGURE 2.19. UML for ct.kernel package, director related classes.

2.8.3 Scheduling

This section and the following three sections provide technical details and design decisions made in the implementation of the CT domain. These details are only necessary if the readers want to implement new directors or ODE solvers.

In general, simulating a continuous-time system (3)-(5) by a time-marching ODE solver involves the following execution steps:

1. Given the state of the system $x_{t_0} \ldots x_{t_{n-1}}$ at time points $t_0 \ldots t_{n-1}$, if the current integration step size is h, i.e. $t_n = t_{n-1} + h$, compute the new state x_{t_n} using the numerical integration algorithms. During the application of an integration algorithm, each evaluation of the $f(a, b, t)$ function is achieved by the following sequence:

- Integrators emit tokens corresponding to a;
- Source actors emit tokens corresponding to b;
- The current time is set to t;
- The tokens are passed through the topology (in a data-driven way) until they reach the integrators again. The returned tokens are $\dot{x}|_{x=a} = f(a, b, t)$.

2. After the new state x_{t_n} is computed, test whether this step is successful. Local truncation error and unpredictable breakpoints are the issues to be concerned with, since those could lead to an unsuccessful step.

3. If the step is successful, predict the next step size. Otherwise, reduce the step size and try again.

Due to the signal-flow representation of the system, the numerical ODE solving algorithms are implemented as actor firings and token passings under proper scheduling.

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

The state transition cluster includes all the actors that are in the signal flow path for evaluating the f function in (3). It starts from the source actors and the outputs of the integrators, and ends at the inputs of the integrators. In other words, integrators, and in general dynamic actors, are used to break causality loops in the model. A topological sort of the cluster provides an enumeration of actors in the order of their firings. This enumeration is called the *state transition schedule*. After the integrators produce tokens representing x_t, one iteration of the state transition schedule gives the tokens representing $\dot{x}_t = f(x_t, u(t), t)$ back to the integrators.

The output cluster consists of actors that are involved in the evaluation of the output map g in (4). It is also similarly sorted in topological order. The *output schedule* starts from the source actors and the integrators, and ends at the sink actors.

For example, for the system shown in figure 2.3, the state transition schedule is

```
U-G1-G2-G3-A
```

where the order of `G1`, `G2`, and `G3` are interchangeable. The output schedule is

```
G4-Y
```

The event generating schedule is empty.

A special situation that must be taken care of is the firing order of a chain of integrators, as shown in figure 2.21. For the implicit integration algorithms, the order of firings determines two distinct kinds of fixed point iterations. If the integrators are fired in the topological order, namely $x_1 \rightarrow x_2$ in our example, the iteration is called the *Gauss-Seidel iteration*. That is, x_2 always uses the new guess from

x_1 in this iteration for its new guess. On the other hand, if they are fired in the reverse topological order, the iteration is called the *Gauss-Jacobi iteration*, where x_2 uses the tentative output from x_1 in the last iteration for its new estimation. The two iterations both have their pros and cons, which are thoroughly discussed in [112]. Gauss-Seidel iteration is considered faster in the speed of convergence than Gauss-Jacobi. For explicit integration algorithms, where the new states x_{t_n} are calculated solely from the history inputs up to $\dot{x}_{t_{n-1}}$, the integrators must be fired in their reverse topological order. For simplicity, the scheduler of the CT domain, at this time, always returns the reversed topological order of a chain of integrators. This order is considered safe for all integration algorithms.

2.8.4 Controlling Step Sizes

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

- *Error control*. For all integration algorithms, the *local error* at time t_n is defined as a vector norm (say, the 2-norm) of the difference between the actual solution $x(t_n)$ and the approximation x_{t_n} calculated by the integration method, given that the last step is accurate. That is, assuming $x_{t_{n-1}} = x(t_{n-1})$ then

$$E_{t_n} = \left\| x_{t_n} - x(t_n) \right\| . \tag{28}$$

  It can be shown that by carefully choosing the parameters in the integration algorithms, the local error is approximately of the *p*-th order of the step size, where *p*, an integer closely related to the number of *f* function evaluations in one integration step, is called the *order* of the integration algorithm, i.e. $E_{t_n} \sim O((t_n - t_{n-1})^p)$. Therefore, in order to achieve an accurate solution, the step size should be chosen to be small. But on the other hand, small step sizes means long simulation time. In general, the choice of step size reflects the trade-off between speed and accuracy of a simulation.
- *Convergence*. The local contraction mapping theorem (Theorem 2 in Appendix A) shows that for implicit ODE solvers, in order to find the fixed point at t_n, the map $F_I(\ )$ in (15) must be a (local) contraction map, and the initial guess must be within an ε ball (the contraction radius) of the solution. It can be shown that $F_I(\ )$ can be made contractive if the step size is small enough. (The choice of the step size is closely related to the Lipschitz constant). So the general approach for resolving the fixed point is that if the iterating function $F_I(\ )$ does not converge at one step size, then reduce the step size by half and try again.
- *Discontinuity.* At discontinuous points, the derivatives of the signals are not continuous, so the integration formula is not applicable. That means the discontinuous points can not be crossed by one integration step. In particular, suppose the current time is *t* and the intended next time point is *t+h*. If there is a discontinuous point at $t + \delta$, where $\delta < h$, then the next step size should be reduced to $t + \delta$. For a predictable breakpoint, the director can adjust the step size accordingly before starting an integration step. However for an unpredictable breakpoint, which is reported "missed" after an integration step, the director should be able to discard its last step and restart

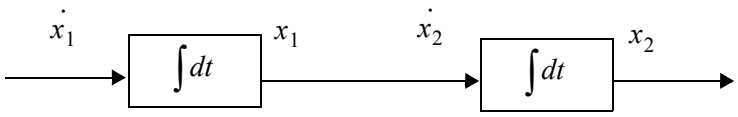


FIGURE 2.21. A chain of integrators.

with a smaller step size to locate the actual discontinuous point.

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

2.8.5 Mixed-Signal Execution

DE inside CT.

Since time advances monotonically in CT and events are generated chronologically, the DE component receives input events monotonically in time. In addition, a composition of causal DE components is causal [79], so the time stamps of the output events from a DE component are always greater than or equal to the global time. From the view point of the CT system, the events produced by a DE component are predictable breakpoints.

Note that in the CT model, finding the numerical solution of the ODE at a particular time is semantically an instantaneous behavior. During this process, the behavior of all components, including those implemented in a DE model, should keep unchanged. This implies that the DE components should not be executed during one integration step of CT, but only between two successive CT integration steps.

CT inside DE.

When a CT component is contained in a DE system, the CT component is required to be causal, like all other components in the DE system. Let the CT component have local time t, when it receives an input event with time stamp τ. Since time is continuous in the CT model, it will execute from its local time t, and may generate events at any time greater or equal to t. Thus we need

$$t \geq \tau \tag{29}$$

to ensure causality. This means that the local time of the CT component should always be greater than or equal to the global time whenever it is executed.

This ahead-of-time execution implies that the CT component should be able to remember its past states and be ready to rollback if the input event time is smaller than its current local time. The state it needs to remember is the state of the component after it has processed an input event. Consequently, the CT component should not emit detected events to the outside DE system before the global time reaches the event time. Instead, it should send a pure event to the DE system at the event time, and wait until it is safe to emit it.

2.8.6 Hybrid System Execution

Although FSM is an untimed model, its composition with a timed model requires it to transfer the notion of time from its external model to its internal model. During continuous evolution, the system is simulated as a CT system where the FSM is replaced by the continuous component refining the current FSM state. After each time point of CT simulation, the triggers on the transitions starting from the current FSM state are evaluated. If a trigger is enabled, the FSM makes the corresponding transition. The continuous dynamics of the destination state is initialized by the actions on the transition. The simulation continues with the transition time treated as a breakpoint.

Appendix A: Brief Mathematical Background

Theorem 1. [Existence and uniqueness of the solution of an ODE] Consider the initial value ODE problem

$$\begin{aligned} \dot{x} &= f(x, t) \quad . \\ x(t_0) &= x_0 \end{aligned} \tag{30}$$

If f satisfies the conditions:

1. [*Continuity Condition*] Let D be the set of possible discontinuity points; it may be empty. For each fixed $x \in \Re^n$ and $u \in \Re^m$, the function $f: \Re \setminus \mathrm{D} \to \Re^n$ in (30) is continuous. And $\forall \tau \in D$, the left-hand and right-hand limit $f(x, u, \tau^-)$ and $f(x, u, \tau^+)$ are finite.
2. [*Lipschitz Condition*] There is a piecewise continuous bounded function $k: \Re \to \Re^+$, where $\Re^+$ is the set of non-negative real numbers, such that $\forall t \in \Re$, $\forall \zeta, \xi \in \Re^n$, $\forall u \in \Re^m$

$$\| f(\xi, u, t) - f(\zeta, u, t) \| \le k(t) \| \xi - \zeta \| . \tag{31}$$

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

$$\psi(t_0) = x_0 \tag{32}$$

and

$$\dot{\psi}(t) = f(\psi(t), u(t), t) \qquad \forall t \in \Re \setminus \mathrm{D} . \tag{33}$$

This function $\psi(t)$ is called the *solution* through (t_0, x_0) of the ODE (30).

◆

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

$$\sigma_1 = F(\sigma_0), \sigma_2 = F(\sigma_1), \sigma_3 = F(\sigma_2), \ldots \tag{34}$$

converges to σ.