9

Hybrid Systems

In chapter 6, we studied continuous-time models of physical plants and controllers. In this chapter, we turn our attention to systems whose dynamics consists of both continuous evolution as time elapses and discrete instantaneous updates to state. In fact, the timed processes of chapter 7 already exhibit this mix of discrete and continuous updates in their dynamics: the values of the clock variables of a timed process increase as time elapses while the process waits in a mode, and state variables are updated in a discrete manner during a mode switch. Hybrid systems admit more general forms of continuous-time evolution for state variables described using differential and algebraic equations. Such models provide a unified framework for designing and analyzing systems that integrate computation, communication, and control of the physical world.

9.1 Hybrid Dynamical Models

The model of computation for hybrid processes is a generalization of the model for timed processes studied in chapter 7.

9.1.1 Hybrid Processes

We describe a hybrid process using an extended-state machine with modes and mode-switches. Each process has input, output, and state variables, and some of these variables are of type `cont`. A variable of type `cont` takes values from the set of real numbers (or an interval of real numbers) and is updated *continuously* as time progresses while a process waits in a mode. A mode-switch is executed discretely and takes zero time. As usual, such a switch is guarded with a condition over state and input variables, can update state and output variables, and describes either an input, an output, or an internal action. A mode is annotated with differential and algebraic equations that specify how state and output variables of type `cont` evolve. In addition, each mode also specifies a constraint on how long the process can wait in that mode using a

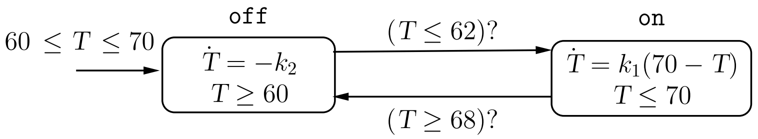


Figure 9.1: A Thermostat Model Switching between Two Modes

Boolean expression over the state variables.

Switching Thermostat

As an example of a dynamical system switching between two modes, consider a simple model of a self-regulating thermostat shown in figure 9.1. The process `Thermostat` can be in two modes, `off` and `on`. The temperature T is a continuous-time variable. When the mode is `on`, the dynamics of the system is given by the differential equation $\dot{T} = k_1(70 - T)$, where k_1 is a constant. The temperature changes continuously according to this differential equation. Note that this dynamics is linear, and for a given initial value of the temperature, there is a unique response signal that captures how the temperature evolves with time. The constraint $(T \leq 70)$ associated with the mode specifies that the process can stay in this mode only as long as this condition holds: the switch to the mode `off` must occur before this constraint is violated. The switch to the mode `off` is guarded by the condition $(T \geq 68)$. This implies that the switch may happen any time after the temperature exceeds 68.

In the mode `off`, the dynamics of how the temperature changes is described by the differential equation $\dot{T} = -k_2$, where k_2 is a constant. Thus, the temperature falls linearly with time when the thermostat is off. The constraint $(T \geq 60)$ associated with the mode `off` specifies that the process must switch to the mode `on` before the temperature falls below 60, and the guard $(T \leq 62)$ associated with the switch from the mode `off` to the mode `on` implies that this switch may occur at any time after the temperature drops below 62.

Initially the mode is `off` and the temperature is T_0. The set of choices for the initial temperature is described by the initialization constraint $60 \leq T \leq 70$. Figure 9.2 shows a possible execution of the thermostat process for the initial temperature $T_0 = 66$ with the constants $k_1 = 0.6$ and $k_2 = 2$. The execution proceeds in phases: during each phase, the mode stays unchanged, and the temperature changes as a continuous function of time according to the differential equation of the current mode. When a mode-switch occurs, state changes discontinuously. In this model, there is an uncertainty in the times at which the mode-switches occur, and thus the model is non-deterministic, and even when we fix the initial temperature, the process has many possible executions.

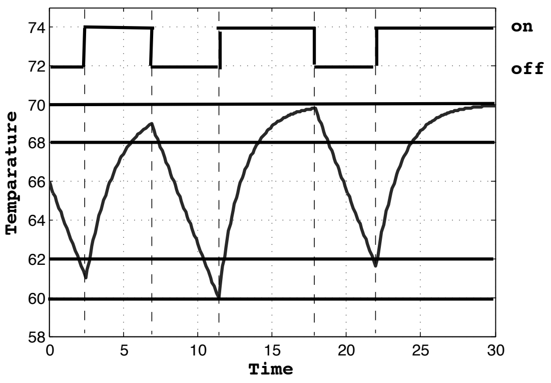


Figure 9.2: A Possible Execution of the Process `Thermostat`

The behavior of the process `Thermostat` while it stays in a given mode can be analyzed using techniques discussed in chapter 6. If the process switches to the mode `off` at time t^* with the temperature equal to T^*, then until the next mode-switch, the value of the temperature at time t is given by the expression $T^* - k_2\,(t - t^*)$. Assuming that the temperature T^* upon entry is at least 62, the process spends at least $(T^* - 62)/k_2$ seconds and at most $(T^* - 60)/k_2$ seconds in the mode `off`.

If the process switches to the mode `on` at time t^* with the temperature equal to T^*, then until the next mode-switch, the value of the temperature at time t is given by the expression $70 - (70 - T^*)\,e^{-k_1(t-t^*)}$. Assuming that the entry temperature T^* does not exceed 68, the process spends at least `ln` $(2/(70 - T^*))/k_1$ seconds in the mode `on`. It can stay there for an indefinite period as the value of the temperature is guaranteed not to exceed 70 according to this equation.

Bouncing Ball

Consider a ball dropped from an initial height $h = h_0$ with initial velocity $\dot{h} = v = v_0$. The ball drops freely with its dynamics given by the differential equation $\dot{v} = -g$, where g is the gravitational acceleration. When it hits the ground, that is, when the value of the variable h becomes 0, there is a discontinuous update in its velocity. This discrete change can be modeled as a mode switch with the update given by $v := -a\,v$. This assumes that the collision is inelastic, and the velocity decreases by a factor a, for some appropriate constant $0 < a < 1$. This behavior is captured by the hybrid process `BouncingBall` of figure 9.3. It has a single mode and two state variables of type `cont`. Whenever the condition $(h = 0)$ holds, a mode-switch is triggered. The output event *bump* is issued by the process, and the assignment is executed reflecting the change in the direction

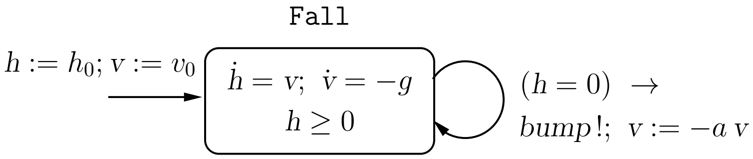


Figure 9.3: A Bouncing Ball as a Single-mode Hybrid System

of the velocity. The invariant constraint ($h \geq 0$) ensures that the switch is executed whenever the height h becomes 0. Figure 9.4 shows an execution of the process `BouncingBall` with the initial velocity $v_0 = 0$, initial height $h_0 = 5m$, gravitational acceleration $g = 9.8\,m/s^2$, and damping coefficient $a = 0.8$.

Note that the model capturing the continuous-time evolution of the state is a two-dimensional linear system: for each state variable, its rate of change with time is given as a linear function of the state variables. However, due to the discontinuous update of the velocity during the discrete mode-switch, the value of a state variable at time t cannot be expressed as a nice closed-form function of the initial state.

Formal Model

We define the formal model for hybrid processes along the lines of the formal model for timed processes. A hybrid process consists of an asynchronous process that is defined by listing its input channels, output channels, state variables, initialization, input tasks, output tasks, and internal tasks. Input, output, and internal actions are defined as in the case of asynchronous processes and can discretely update any of the variables. A variable can be of type `cont`, indicating that the variable evolves continuously during a timed action. A variable of a type other than `cont` is updated only discretely, and let us call such variables *discrete* variables.

To execute a timed action of duration δ, for each continuously updated input variable u, we need a continuous signal $\overline{u}$ that specifies the value of the input u over the interval $[0, \delta]$. As in the case of continuous-time components, the continuous-time evolution is specified by a real-valued expression h_y for every continuously updated output variable y and a real-valued expression f_x for every continuously updated state variable x. Each of these expressions is an expression over the continuously updated input variables and state variables. The value of the output variable y of type `cont` at time t is obtained by evaluating the expression h_y using the values of the state and input variables at time t. The signal for a continuously updated state variable x should be a differentiable function such that its rate of change at time t equals the value of the expression f_x evaluated using the values of the state and input variables at time t. Note that the discrete input and discrete output variables are not relevant during a

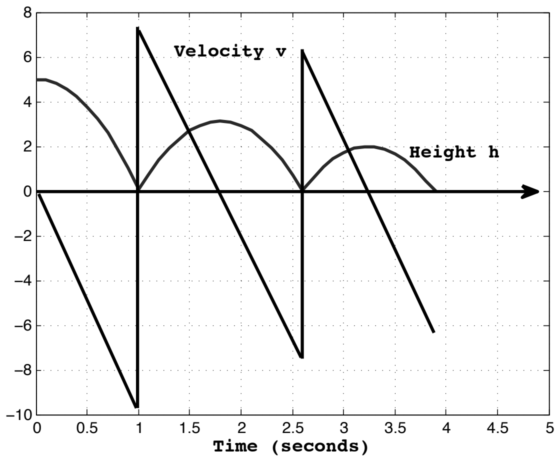


Figure 9.4: The Execution of the Hybrid Process `BouncingBall`

timed action, and the value of a discrete state variable stays unchanged during a timed action. These rules define the state signal and the output signal for the continuously updated output variables for the duration $[0, \delta]$. If the expressions h_y and f_x used to define the dynamics are Lipschitz-continuous, then the state and the output signals corresponding to a given continuous input signal during a timed action are uniquely defined. The *continuous-time invariant* is specified as a Boolean expression over the state variables, and it is required that at every time instance during the timed action, the state signal satisfies this invariant. During a timed action of duration δ, the process and its environment synchronize on the evolution of the continuously updated variables as in the case of dynamical models and also agree on not executing a discrete action for this duration.

Let us revisit the process `Thermostat` shown in figure 9.1. Let us assume that the output of the process is the temperature. Then the hybrid process corresponding to this state machine consists of the following components:

- it has no input variables;
- it has a single output variable T of type `cont`;
- it has a discrete state variable *mode* of enumerated type $\{$`off`, `on`$\}$, and a state variable T of type `cont`;
- the variable *mode* is initialized to `off`, and the initialization of the variable T is given by the nondeterministic choice $60 \leq T \leq 70$;
- it has no output tasks, which means that the value of the temperature is not transmitted during the discrete actions;

- it has two internal tasks corresponding to the two mode-switches: one task has the guard ($\mathit{mode} = \texttt{off} \ \wedge \ T \leq 62$) and the update $\mathit{mode} := \texttt{on}$, and the second task has the guard ($\mathit{mode} = \texttt{on} \ \wedge \ T \geq 68$) and the update $\mathit{mode} := \texttt{off}$;

- the expression defining the value of the output variable T equals the state variable T;

- the expression defining the derivative of the state variable T is given by the *conditional* expression

$$\texttt{if}\ (\mathit{mode} = \texttt{off})\ \texttt{then}\ -k_2\ \texttt{else}\ k_1\,(70 - T);$$

- the continuous-time invariant CI is given by the expression:

$$(\mathit{mode} = \texttt{off}) \ \rightarrow \ (T \geq 60)\,] \ \wedge \ [\,(\mathit{mode} = \texttt{on}) \ \rightarrow \ (T \leq 70)\,].$$

The formal definition is summarized below:

Hybrid Process

A *hybrid process HP* consists of (1) an asynchronous process P where some of the input, output, and state variables are of type `cont`; (2) a *continuous-time invariant CI*, which is a Boolean expression over the state variables S; (3) for every output variable y of type `cont`, a real-valued expression h_y over the state and input variables of type `cont`; and (4) for every state variable x of type `cont`, a real-valued expression f_x over the state and input variables of type `cont`. Inputs, outputs, states, initial states, internal actions, input actions, and output actions of the hybrid process HP are the same as those of the asynchronous process P. Given a state s, a real-valued time $\delta > 0$, and an input signal $\overline{u}$ for every input variable u of type `cont` over the interval $[0, \delta]$, the corresponding timed action of the process HP is the differentiable state signal $\overline{S}$ over the state variables, and the signal $\overline{y}$ for every output variable y of type `cont` over the interval $[0, \delta]$ such that (1) for every state variable x, $\overline{x}(0) = s(x)$; (2) for every discrete state variable x and time $0 \leq t \leq \delta$, $\overline{x}(t) = s(x)$; (3) for every output variable y of type `cont` and time $0 \leq t \leq \delta$, $\overline{y}(t)$ equals the value of h_y evaluated using the values $\overline{u}(t)$ and $\overline{S}(t)$; (4) for every state variable x of type `cont` and time $0 \leq t \leq \delta$, the time derivative $(d/dt)\,\overline{x}(t)$ equals the value of f_x evaluated using the values $\overline{u}(t)$ and $\overline{S}(t)$; and (5) for all $0 \leq t \leq \delta$, the continuous-time invariant CI is satisfied by the values $\overline{S}(t)$ of the state variables at time t.

Executions

The execution of a hybrid process starts in an initial state. At each step, either an internal, an input, an output, or a timed action is executed. For example,

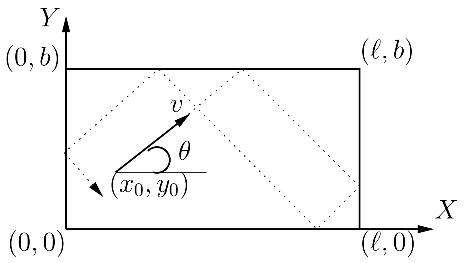


Figure 9.5: Motion of a Billiard Ball

the execution of the process `Thermostat` shown in figure 9.2 corresponds to the following sequence of alternating timed and internal actions:

$$(\texttt{off},66) \xrightarrow{2.5} (\texttt{off},61) \xrightarrow{\varepsilon} (\texttt{on},61) \xrightarrow{3.7} (\texttt{on},69.02) \xrightarrow{\varepsilon}$$
$$(\texttt{off},69.02) \xrightarrow{4.4} (\texttt{off},60.22) \xrightarrow{\varepsilon} (\texttt{on},60.22) \xrightarrow{7.6} (\texttt{on},69.9) \xrightarrow{\varepsilon}$$
$$(\texttt{off},69.9) \xrightarrow{4.1} (\texttt{off},61.7) \xrightarrow{\varepsilon} (\texttt{on},61.7) \xrightarrow{7.7} (\texttt{on},69.92).$$

During each timed action, the process continuously outputs the value of the temperature. For example, during the first timed action of duration 2.5, the temperature signal is given by $\overline{T}(t) = 66 - 2\,t$, while during the second timed action of duration 3.7, the temperature signal is given by $70 - 9\,e^{-0.6t}$.

Note that discrete and timed actions need not strictly alternate during an execution: two timed actions can appear consecutively and so can two discrete actions. In particular, the first timed action of duration 2.5 in the execution above can be split into multiple timed actions:

$$(\texttt{off},66) \xrightarrow{1.5} (\texttt{off},63) \xrightarrow{0.8} (\texttt{off},61.4) \xrightarrow{0.2} (\texttt{off},61).$$

A state s of a hybrid process is reachable if there is an execution that ends in the state s. Given a hybrid process HP and a property φ over its state variables, the property φ is said to be an invariant of the process HP if every reachable state of the process HP satisfies the property φ. For example, the property $60 \leq T \leq 70$ is an invariant of the process `Thermostat`.

Exercise 9.1: Specify the model of the bouncing ball shown in figure 9.3 as a formal hybrid process by listing all its components. Assume that the outputs of the process are the discrete bump events and the continuously updated height. ■

Exercise 9.2: In this problem, we want to construct a hybrid systems model of the motion of a ball on a billiards table with perfect collisions (see figure 9.5). The table has length ℓ units and breadth b units. The ball is initially hit from the position (x_0, y_0) with an initial speed v in the direction θ. Whenever the

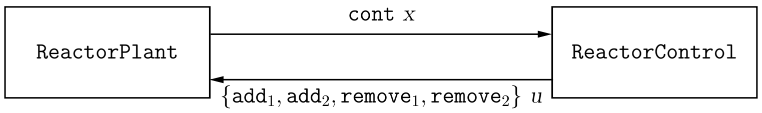

ReactorPlant
cont x
ReactorControl
$\{\texttt{add}_1, \texttt{add}_2, \texttt{remove}_1, \texttt{remove}_2\}$ u

Figure 9.6: The Block Diagram for the Reactor Example

ball hits a side parallel to the X-axis, its velocity in the Y-direction flips sign, and the velocity in the X-direction stays unchanged. Symmetrically, whenever the ball hits a side parallel to the Y-axis, its velocity in the X-direction flips sign, and the velocity in the Y-direction stays unchanged. Thus, we are ignoring friction and collision impact. When the ball reaches one of the corner points, it drops and stops. Describe a precise hybrid state machine corresponding to this description. ■

Exercise 9.3 *: Consider a mobile robot that moves in a two-dimensional world corresponding to the positive quadrant of the X-Y plane. The robot is initially at the origin and is stationary. The input command to the robot consists of a target location to go to. Assume that there are no obstacles. The robot can move in the horizontal direction at speed 6 m/s, in the vertical direction at speed 8 m/s, or along any other arbitrary direction at speed 5 m/s. The robot plans its trajectory to the target to minimize the time taken. Once it reaches the target, it waits there to receive another input command to move to a new target and repeats the same behavior. Construct a hybrid process (using the extended-state machine notation) to model the behavior of the robot. Clearly specify input and state variables along with their types. For the purpose of this question, you can assume that the time needed to change the velocity is negligible (that is, the robot can change its speed from, say, 0 to 5, instantaneously). ■

9.1.2 Process Composition

Hybrid processes can be put together using block diagrams. Operations such as instantiation, variable renaming, and output hiding are defined in the usual way. To compose two hybrid processes, we compose the corresponding asynchronous processes using the composition operation for asynchronous processes and compose the dynamics during timed actions as in the case of continuous-time components. Thus, two hybrid processes are compatible and can be composed, provided the state variables of the two are disjoint, the output variables of the two are disjoint, and there are no cyclic await dependencies among the continuously updated common input/output variables of the two. The continuous-time invariant for the composed process is simply the conjunction of the continuous-time invariants of the component processes. Thus, the discrete actions of the composite process are obtained by the asynchronous composition of the discrete actions of the component processes, and the timed actions of the composite

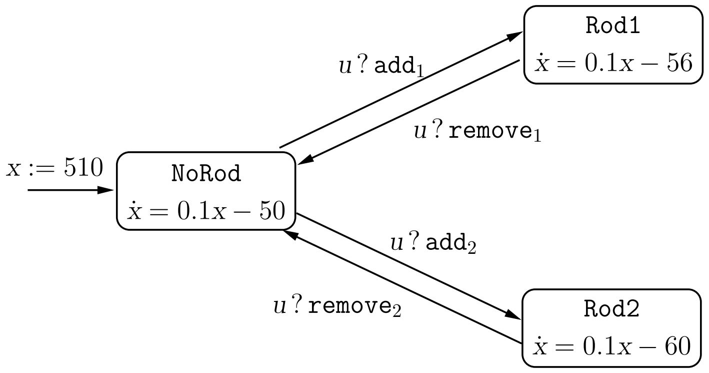


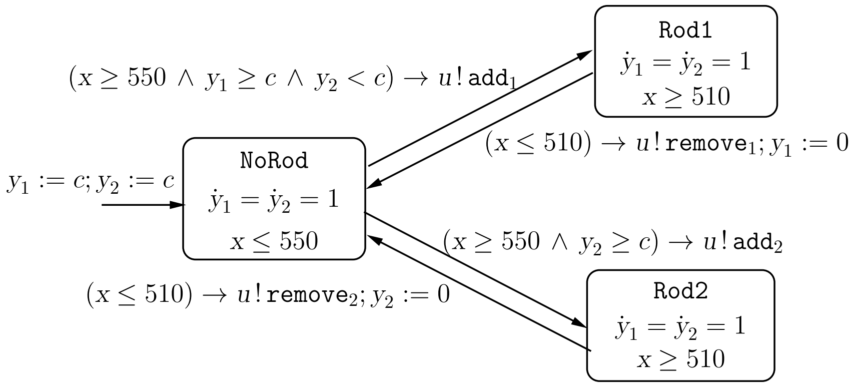
Figure 9.7: The Hybrid Model of the Reactor Plant

process are obtained by the synchronous composition of the timed actions of the component processes. In particular, a timed action of a duration δ is possible in the composite process only when both components are willing to evolve continuously for the duration of δ without an interrupting discrete action.

To illustrate process composition, we consider a toy model of a nuclear reactor with two control rods. The reactor and the controller are modeled by the hybrid processes `ReactorPlant` and `ReactorControl`, respectively. The interaction pattern is shown in the block diagram of figure 9.6. The output of the process `ReactorPlant` is the continuously updated variable x that captures the reactor temperature, and this variable is monitored by the controller. The output of the process `ReactorControl` is the event variable u that can take values $\texttt{add}_1$ (a control command to insert the first rod), $\texttt{remove}_1$ (a control command to remove the first rod), $\texttt{add}_2$ (a control command to insert the second rod), and $\texttt{remove}_2$ (a control command to remove the second rod).

The hybrid process corresponding to the plant model is shown in figure 9.7. The process has three modes `NoRod`, `Rod1`, and `Rod2`, corresponding, respectively, to whether there is no rod in the reactor, the first rod is in the reactor, or the second rod is in the reactor. Initially, the temperature is 510 degrees, and no rods are in the reactor. The dynamics for the change in the temperature is described by the differential equation $\dot{x} = 0.1x - 50$. When the controller issues the event $\texttt{add}_1$, the plant switches to the mode `Rod1`. The rod has a dampening effect, which slows down the rate of increase in temperature, and the dynamics is given by the differential equation $\dot{x} = 0.1x - 56$. On receiving the control command $\texttt{remove}_1$, the plant switches back to the mode `NoRod`. The mode `Rod2` is similar except that the second rod causes a stronger dampening with the dynamics given by the differential equation $\dot{x} = 0.1x - 60$.

The controller for the plant is shown in figure 9.8. Once a rod is removed, it cannot be reinserted for a time period of c time units. To capture this restriction,



Rod1
$\dot{y}_1 = \dot{y}_2 = 1$
$x \geq 510$
$(x \geq 550 \wedge y_1 \geq c \wedge y_2 < c) \rightarrow u!\mathtt{add}_1$
$(x \leq 510) \rightarrow u!\mathtt{remove}_1; y_1 := 0$
NoRod
$y_1 := c; y_2 := c$
$\dot{y}_1 = \dot{y}_2 = 1$
$x \leq 550$
$(x \geq 550 \wedge y_2 \geq c) \rightarrow u!\mathtt{add}_2$
$(x \leq 510) \rightarrow u!\mathtt{remove}_2; y_2 := 0$
Rod2
$\dot{y}_1 = \dot{y}_2 = 1$
$x \geq 510$

Figure 9.8: The Hybrid Model of the Reactor Controller

we introduce two variables: y_1 and y_2. The rate of change of these continuously updated variables is 1 in all the modes, and thus they are the same as the clock variables of timed models. Initially, the clock y_1 equals c and is reset to 0 every time the first rod is removed. The controller can issue the output $\mathtt{add}_1$ only when the clock y_1 is at least c. This ensures that the delay between an event $\mathtt{remove}_1$ and the subsequent $\mathtt{add}_1$ is at least c time units. The variable y_2 is updated similarly and ensures that the delay between an event $\mathtt{remove}_2$ and the subsequent $\mathtt{add}_2$ is at least c time units.

Initially, the controller is in the mode `NoRod`. The continuous-time invariant $(x \leq 550)$ ensures that when the plant temperature rises to 550 degrees, a mode-switch is triggered. Note that in this example, the variable x is updated by the process `ReactorPlant`, which specifies the differential equations regarding how the variable evolves. It is monitored by the process `ReactorControl`, which constrains the durations of timed actions via continuous-time invariants that refer to x. When the temperature reaches 550, depending on the values of the clock variables y_1 and y_2, the controller process can decide to insert the first rod by issuing the output $\mathtt{add}_1$ or the second rod by issuing the output $\mathtt{add}_2$. If both choices are available, then the controller prefers the second rod with the stronger dampening effect. The controller process stays in the mode `Rod1` or `Rod2` as long as the temperature is above 510 degrees, and if it falls below that, it switches back to the mode `NoRod` by issuing a command to remove the corresponding rod.

When the mode equals `NoRod`, if the temperature rises to 550 and both the clock variables y_1 and y_2 are smaller than c, thereby disabling both the control actions $\mathtt{add}_1$ and $\mathtt{add}_2$, the temperature of the core can rise to an unacceptable level, causing an alarm. Formal analysis can reveal the range of values for the parameter c for which such an alarm condition is not feasible.

Exercise 9.4: Consider the composition of the hybrid processes `ReactorPlant` and `ReactorControl`. Show executions of the system using a simulation tool (such as MATLAB) for the following values of the constant c: 10, 20, 30, 40, 50, and 60. ■

9.1.3 Zeno Behaviors

Execution of the Bouncing Ball

Let us reconsider the hybrid process `BouncingBall` corresponding to the bouncing ball of figure 9.3. Let us assume that the initial velocity v_0 equals 0. Then before the first bump, the change in height is described by the equation $\overline{h}(t) = h_0 - g\,t^2/2$. The guard condition $(h = 0)$ for the mode-switch becomes true at time $\delta_1 = \sqrt{2\,h_0/g}$. The first timed action is of this duration δ_1, and during this action, the height decreases from h_0 to 0, and the velocity changes from 0 to $-v_1$, where $v_1 = g\,\delta_1 = \sqrt{2\,g\,h_0}$. At this instance, a discrete output action is executed, the event *bump* is issued, and the velocity changes its direction while its magnitude decreases by a factor of a. Thus, the new velocity is $v_2 = av_1$. After the first bump until the next bump, the height signal during the timed action is given by $\overline{h}(t) = v_2\,t - g\,t^2/2$ and the velocity signal is given by $\overline{v}(t) = v_2 - g\,t$. This timed action is of duration $\delta_2 = 2\,v_2/g$, and captures one bounce corresponding to the parabolic motion of the ball. At the end of this action, the height becomes 0 again, and the velocity is $-v_2$. As a result of the bump, the velocity is updated to $v_3 = a\,v_2 = a^2\,v_1$, and the entire cycle repeats.

If we concatenate a sequence of successive timed actions into one single timed action, then the model has a single infinite execution that can be described as

$$(h_0, 0) \xrightarrow{\delta_1} (0, -v_1) \xrightarrow{bump!} (0, v_2) \xrightarrow{\delta_2} (0, -v_2) \xrightarrow{bump!} (0, v_3) \xrightarrow{\delta_3} \cdots$$

where for each i, $v_{i+1} = a\,v_i = a^i\,v_1$ and $\delta_{i+1} = 2\,v_{i+1}/g = 2\,a^i\,v_1/g$. After k bumps, the velocity of the ball is $a^k\,v_1$. Since $a < 1$, this sequence converges to 0. Similarly, the sequence of durations $\delta_1, \delta_2, \ldots$ of timed actions corresponding to the successive bounces of the ball is decreasing, converging to 0. However, at no point during this infinite execution of the ball, it is stationary, and there is always one more bounce possible. An inductive reasoning about such a behavior of the ball can lead to the flawed conclusion that at no point in time the ball is at rest.

Zeno's Paradox

The phenomenon exhibited by the bouncing ball was noted by the Greek philosophers many centuries ago and is known as the *Zeno's Paradox*. This was originally posed in the context of a race between Achilles and the tortoise, in which the tortoise has a head start. Suppose the tortoise is ahead of Achilles by d_1

meters at the beginning of a round. By the time Achilles has covered this distance d_1, the tortoise has moved a little bit ahead, say d_2 meters, with $d_2 < d_1$, and for the next round, Achilles has now to cover another d_2 meters, during which the tortoise has moved farther by d_3 meters with $d_3 < d_2$. By inductive reasoning, for every natural number n, after n rounds, the tortoise is ahead of Achilles by a non-zero distance, and thus Achilles will never be able to catch up with the tortoise! The source of paradoxical behavior lies in the fact that executions in which the total elapsed time does not grow in an unbounded manner do not adequately describe the system state with progression of time.

The total time elapsed along an execution is the sum of the durations of all the timed actions in the execution. In our bouncing ball example, this sum is $\Sigma_{i\geq 1}\,\delta_i$. Since the sequence $\delta_1, \delta_2, \delta_3 \ldots$ converges to 0, the sum is bounded by a constant K. Plugging in the expressions for the durations δ_i and velocities v_i discussed above, and using the fact that the geometric series $\Sigma_{i\geq 1}\,a^i$ equals $a/(1-a)$, we can compute the expression for the constant K:

$$\sum_{i\geq 1} \delta_i \;=\; \sqrt{2\,g\,h_0}\;(1+a)/(1-a).$$

Thus, the execution, even though it contains infinitely many output and timed actions, does not describe what happens at time K (and beyond). In reality, the ball would be stationary on the ground at time K, but the execution never gets to time K.

Zeno Executions and Zeno States

An infinite execution of a hybrid process *HP* is said to be a *Zeno execution* if the sum of the durations of all the timed actions in the execution is bounded by a constant. Thus, a non-Zeno execution is an execution in which time diverges. Zeno executions are an artifact of mathematical modeling, and proving properties using Zeno executions will lead to misleading conclusions.

In the bouncing-ball model, from the initial state, every possible infinite execution is a Zeno execution. Such a state is called a *Zeno state*. It is the analog of the deadlock states discussed in chapter 4. From a deadlock state there is no enabled action, and thus there is no way to continue the execution even for one step. From a Zeno state, there is no way to produce an infinite execution on which time keeps increasing unboundedly.

Note that the existence of a Zeno execution starting from a state does not imply that the state is Zeno. For example, consider the initial state $(\texttt{off}, 66)$ of the process `Thermostat` of figure 9.1. Consider the execution in which we first execute a timed action of duration $\delta_1 = 0.5$, then a timed action of duration $\delta_2 = 0.25$, and repeat this pattern: the ith action in the execution is a timed action of duration $1/2^i$. This is an infinite execution, where the total sum of durations of all the timed actions is bounded by 1. Thus, the execution is a Zeno execution. However, the initial state is not a Zeno state: we could have chosen durations of timed actions so that the resulting execution is non-Zeno (in particular, see the execution corresponding to the one illustrated in figure 9.2).

A hybrid process is called Zeno if some reachable state of the system is a Zeno state. For a Zeno process, the execution can end up in a state from which every possible way of continuing the execution causes convergence in the sum of the durations of timed actions. The process `BouncingBall` of figure 9.3 is a Zeno process. However, the process `Thermostat` of figure 9.1 is a non-Zeno process, and so is the process obtained by composing the processes `ReactorPlant` and `ReactorControl` (see section 9.1.2).

These definitions are summarized below.

Zeno Executions, States, and Processes

An infinite execution of a hybrid process *HP* is said to be a *Zeno execution* if the sum of the durations of all the timed actions in the execution is bounded by a constant. A state s of a hybrid process *HP* is said to be a *Zeno state* if every infinite execution that contains the state s is a Zeno execution. A hybrid process *HP* is said to be a *Zeno process* if there exists a state s that is reachable and is a Zeno state.

Revising Zeno Models

The Zenoness of the hybrid process `BouncingBall` is perhaps not a serious problem if we were analyzing it in isolation. However, consider the composite process

$$HP \;=\; \texttt{BouncingBall} \parallel \texttt{Thermostat}$$

obtained by the parallel composition with the thermostat model. Even though the two processes `BouncingBall` and `Thermostat` do not communicate, they synchronize on the passage of time. In particular, the composite process *HP* is a Zeno process, and its initial state is a Zeno state. This can lead to absurd conclusions. For instance, suppose that the parameters of the two processes are chosen so that the expression $\sqrt{2\,g\,h_0}\;(1+a)/(1-a)$ that gives the bound on the sum of the durations of all the timed actions of the bouncing ball is less than the earliest time, given by the expression $(T_0 - 62)/k_2$, by which the thermostat can switch to the mode `on`. Then on every execution of the composite process *HP*, the mode of the process `Thermostat` stays `off`, and its temperature variable keeps decreasing from the initial value T_0 without ever crossing 62. By definition, a state is considered reachable only when it appears on some execution of the system. Hence, the following property is an invariant of the composite process *HP*:

$$(mode \;=\; \texttt{off}) \;\wedge\; (62 \;\leq\; T \;\leq\; T_0).$$

This property of course is not an invariant of the process `Thermostat` and will be violated in the physical world regardless of whether a ball is bouncing next to the thermostat.

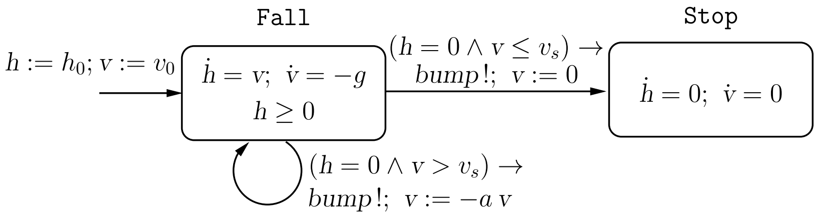

Fall
Stop
$h := h_0;\ v := v_0$
$\dot{h} = v;\ \dot{v} = -g$
$h \geq 0$
$(h = 0 \wedge v \leq v_s) \rightarrow$
$bump!;\ v := 0$
$\dot{h} = 0;\ \dot{v} = 0$
$(h = 0 \wedge v > v_s) \rightarrow$
$bump!;\ v := -a\,v$

Figure 9.9: A Non-Zeno Model of the Bouncing Ball

Thus, in a system consisting of multiple components, the presence of a single Zeno process can affect the analysis of the entire system in unexpected ways. This suggests that Zeno components should be avoided during formal modeling.

A Zeno process can be converted into a non-Zeno process by modifying the model so that the model does not force mode-switches after shorter and shorter durations. For example, figure 9.9 shows the non-Zeno process `NonZenoBall` obtained from the process `BouncingBall` by adding a new mode called `Stop` in which the ball is stationary and adding a mode-switch from the initial mode `Fall` to the mode `Stop` when the velocity during a bump is smaller than some threshold value v_s. Starting from the initial state, if we execute a timed action of maximum possible duration, followed by the output action corresponding to the discrete bump, then it is guaranteed that after finitely many actions, the magnitude of the velocity becomes smaller than this threshold value, and the process switches to the mode `Stop`. Once in the mode `Stop`, timed actions of arbitrary durations are possible, and in particular one can generate a non-Zeno execution.

If we compose the process `Thermostat` and the modified bouncing ball process `NonZenoBall`, then the behavior of the process `Thermostat` is not influenced in any meaningful way. In particular, verify that for any property φ that refers to the mode and the temperature of the thermostat, the property φ is an invariant of the composite process `NonZenoBall` $\|$ `Thermostat` if and only if it is an invariant of the process `Thermostat`.

Exercise 9.5: Consider the following scenario. Two trains are heading toward each other on a single track at constant speeds: the train E is traveling east at a fixed speed v_e, and the train W is traveling west at a fixed speed v_w. A bee B is initially traveling west at a fixed speed v_b along the line joining the two trains. When the bee reaches the train E, it reverses its direction, heads east at the same speed v_b, and reverses its direction again when it reaches the train W. This cycle repeats. Model this scenario as a hybrid process. The state machine can have two modes, one each corresponding to the direction in which the bee is traveling, and three state variables that capture the positions of the train E, the train W, and the bee B. Show that the process is Zeno. Compute the formula that expresses the distance between the two trains as a function of

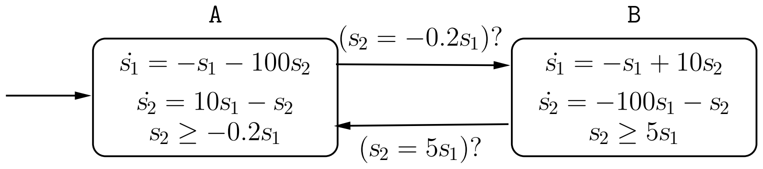

A
B
$\dot{s_1} = -s_1 - 100s_2$
$\dot{s_2} = 10s_1 - s_2$
$s_2 \geq -0.2s_1$
$(s_2 = -0.2s_1)?$
$\dot{s_1} = -s_1 + 10s_2$
$\dot{s_2} = -100s_1 - s_2$
$s_2 \geq 5s_1$
$(s_2 = 5s_1)?$

Figure 9.10: Instability Due to Mode Switching

the speeds v_e, v_w, and v_b and the initial values of the three position variables. ■

Exercise 9.6 *: In this exercise, we establish that the property of being non-Zeno is not preserved by parallel composition. Consider the following specification of the hybrid process *HP*$_1$. It has a single input event x and a single output event y. Whenever it receives an input, it waits for a duration of $1/2^i$ time units, if this is the ith input event it has received so far, and then issues an output event (it does not accept any inputs while it is waiting to issue its output). Design the process *HP*$_1$ as an extended-state machine. It suffices for the process *HP*$_1$ to use a single clock variable, and thus the process *HP*$_1$ is a timed process. Argue that the process *HP*$_1$ is non-Zeno.

Now consider the following specification of another process *HP*$_2$. It has a single input event y and a single output event x. It first issues an output event after a delay of 1 second; subsequently, whenever it receives an input, it waits for a duration of $1/2^i$ time units, if this is the ith input event it has received so far, and then issues an output event. Design the timed process *HP*$_2$ as an extended-state machine. Argue that the process *HP*$_2$ is non-Zeno.

Now consider the parallel composition *HP*$_1$‖*HP*$_2$. Show that the composite process is Zeno. ■

9.1.4 Stability

As discussed in chapter 6, stability is a desirable feature for dynamical systems. Recall that a state s_e of a dynamical system is an equilibrium state if the system, starting in the state s_e, continues to stay in that state in the absence of external inputs. Such an equilibrium state is stable if we perturb the system state slightly, that is, choose the initial state s such that the distance $\|s - s_e\|$ is small, then at all times the state of the system stays within a bounded distance from the equilibrium, and is asymptotically stable if the state converges to the equilibrium with the passage of time.

We can use these same definitions to understand stability of hybrid processes. However, due to the presence of mode-switches, the mathematical analysis used to characterize stability for linear systems, and the associated techniques for

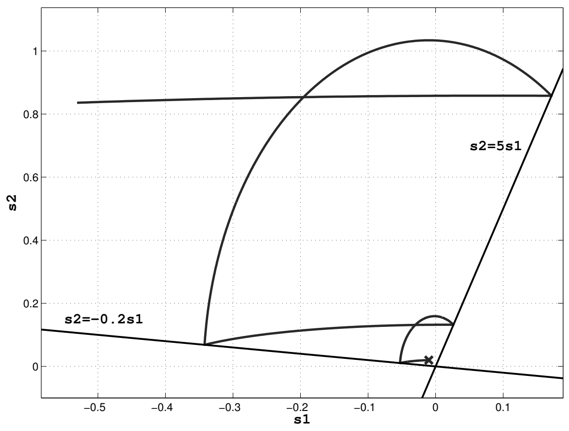


Figure 9.11: Unstable Response due to Mode-switching

designing stabilizing controllers, are not applicable to hybrid systems. We will illustrate the difficulties introduced due to mode-switching using an example.

Consider the hybrid process shown in figure 9.10. The dynamics in the mode A is specified by the linear differential equations $\dot{s}_1 = -s_1 - 100s_2$ and $\dot{s}_2 = 10s_1 - s_2$. Observe that the origin, that is, the state 0, is an equilibrium state. The eigenvalues of the dynamics matrix are $-1 + \sqrt{1000}j$ and $-1 - \sqrt{1000}j$, and from theorem 6.3, we can conclude that the continuous-time system with this dynamics is asymptotically stable.

However, the hybrid process of figure 9.10 stays in the mode A only as long as the invariant $(s_2 \geq -0.2s_1)$ holds, and when the state satisfies the switching condition $(s_2 = -0.2s_1)$, it switches to mode B. The dynamics associated with the mode B is specified by the linear differential equations $\dot{s}_1 = -s_1 + 10s_2$ and $\dot{s}_2 = -100s_1 - s_2$. Observe that this dynamics matrix is the transpose of the dynamics matrix in mode A with identical eigenvalues, and thus a system that evolves according to this dynamics is also asymptotically stable. The process can stay in the mode B only as long as the invariant $(s_2 \geq 5s_1)$ holds, and when the condition $(s_2 = 5s_1)$ is satisfied, it switches back to mode A.

Although the dynamics in the individual modes A and B are asymptotically stable, the switching causes instability. Figure 9.11 shows the execution of the system from the initial state $(-0.01, 0.02)$ in mode A. Indeed the origin is unstable: if the initial state s in mode A is not the origin, then no matter how close it is to the origin, the system state diverges from the origin as time passes.

Analyzing stability of hybrid systems turns out to be a difficult problem. Generalizing analysis techniques from the theory of continuous-time systems to hybrid systems remains an active area of research and is beyond the scope of this textbook.

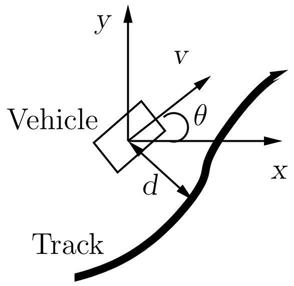


Figure 9.12: The Design Problem for the Automated Guided Vehicle

9.2 Designing Hybrid Systems

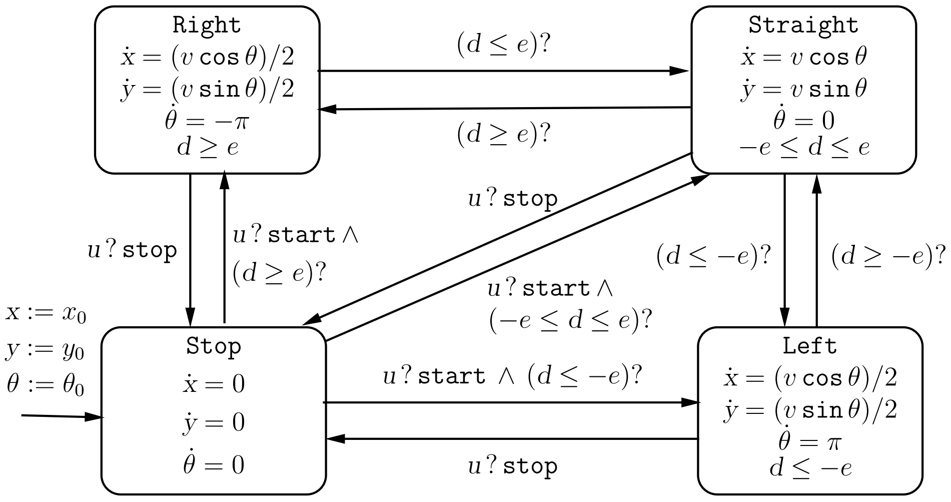
We illustrate the modeling and design of controllers for hybrid systems using three examples. The first example illustrates design of a controller that switches between different modes of operation, the second example illustrates multi-agent collaboration for improved planning, and the third example of multi-hop control networks shows how to model a system that integrates control, computation, and communication.

9.2.1 Automated Guided Vehicle

Consider an autonomous vehicle that needs to be programmed to move along a track as closely following the track as possible. The track is not known to the vehicle in advance but is equipped with sensors. In particular, assuming that the vehicle has not strayed too far from the track, the sensors can measure the distance d of the vehicle from the center of the track. Such information can be provided, for instance, by placing photodiodes along the track.

The vehicle dynamics is modeled as a planar rigid-body motion with two degrees of freedom. It can move forward along its body axis with a maximum possible speed v, and it can rotate about its center of gravity with an angular speed ω, which can range over the interval from $-\pi$ to $+\pi$ radians/second. The variables (x, y) model the position of the vehicle, and θ gives the relative angle with respect to some fixed planar global frame in which the vehicle is headed.

Figure 9.12 shows the design problem for the automated guided vehicle. Based on the current measurement of the distance d, the controller must adjust the control inputs v and ω so as to keep the value of the distance d as close to 0 as possible. The control design problem is additionally constrained by the requirement that the vehicle hardware provides only three discrete settings for the angular speed ω: it can be 0, $+\pi$, or $-\pi$. When $\omega = 0$, the direction θ stays unchanged, and the vehicle is going straight. When $\omega = -\pi$, the direction θ is decreasing as fast as possible, and thus the vehicle is attempting to turn right.



Right
$\dot{x} = (v\,\mathtt{cos}\,\theta)/2$
$\dot{y} = (v\,\mathtt{sin}\,\theta)/2$
$\dot{\theta} = -\pi$
$d \geq e$
Straight
$\dot{x} = v\,\mathtt{cos}\,\theta$
$\dot{y} = v\,\mathtt{sin}\,\theta$
$\dot{\theta} = 0$
$-e \leq d \leq e$
Stop
$\dot{x} = 0$
$\dot{y} = 0$
$\dot{\theta} = 0$
Left
$\dot{x} = (v\,\mathtt{cos}\,\theta)/2$
$\dot{y} = (v\,\mathtt{sin}\,\theta)/2$
$\dot{\theta} = \pi$
$d \leq -e$
$(d \leq e)?$
$(d \geq e)?$
$u\,?\,\mathtt{stop}$
$u\,?\,\mathtt{start} \wedge (d \geq e)?$
$u\,?\,\mathtt{stop}$
$u\,?\,\mathtt{start} \wedge (-e \leq d \leq e)?$
$(d \leq -e)?$
$(d \geq -e)?$
$x := x_0$
$y := y_0$
$\theta := \theta_0$
$u\,?\,\mathtt{start} \wedge (d \leq -e)?$
$u\,?\,\mathtt{stop}$

Figure 9.13: The Hybrid Controller for the Automated Guided Vehicle

Conversely, when $\omega = \pi$, the direction θ is increasing as fast as possible, and thus the vehicle is attempting to turn left.

The control designer makes a further design decision that when the vehicle is headed straight, it moves as fast as possible, with speed v. When the vehicle is turning left or turning right, it attempts to do so at half the maximum possible speed. This leads to four modes of operation as shown in the hybrid state machine of figure 9.13. In the mode `Stop`, the vehicle is stationary; in the mode `Straight`, the vehicle is moving with speed v and $\omega = 0$; in the mode `Left`, the vehicle is turning left with speed $v/2$ and $\omega = \pi$; and in the mode `Right`, the vehicle is turning right with speed $v/2$ and $\omega = -\pi$. In this model, we have assumed that the vehicle can change its speed and direction instantaneously during a mode-switch. This assumption is justifiable if the time needed to switch from one mode to another is insignificant compared with time spent in each mode.

The input variables to the process are the discrete input channel *in* that carries the commands `start` and `stop` to start and stop the vehicle and the continuously updated signal d that captures the error of the trajectory from the desired track. By design, a positive value of d means that the vehicle is off to the left of the track, and a negative value of d means that the vehicle is off to the right of the track.

The switching laws for the controller are designed using the parameter e. Whenever the current distance is in the interval $[-e, +e]$, the vehicle is assumed to be close enough to the track, and the controller decides to move straight. Whenever the current distance exceeds the threshold value e, the controller decides

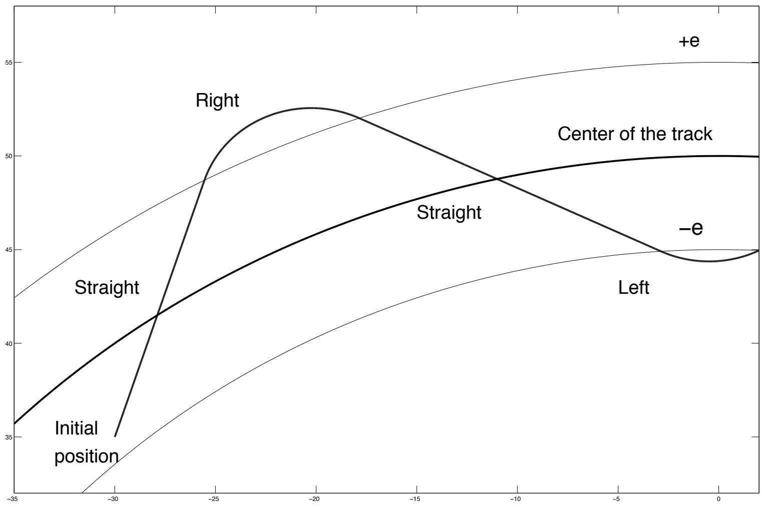


Figure 9.14: A Sample Vehicle Trajectory along a Curved Track

that the vehicle has strayed too far to the left and must be steered to the right by switching to the mode `Right`. Symmetrically, whenever the current distance is smaller than the threshold value $-e$, the controller concludes that the vehicle has strayed too far to the right and must be steered to the left by switching to the mode `Left`. The continuous-time invariants and the guards on the switches of the hybrid process of figure 9.13 capture this logic.

The behavior of the hybrid controller can be understood by considering tracks of different shapes. Figure 9.14 shows a sample trajectory followed by the vehicle along a curve in the track. In the illustrated scenario, the track is a circle centered at the origin with the radius 50. The following values of the parameters are used: $v = 35$, $e = 5$, $x_0 = -30$, $y_0 = 35$, and $\theta_0 = 0.4\pi$. The vehicle is initially going straight. As the distance from the center of the track exceeds the threshold value e, it starts moving right, and when the distance from the center of the track decreases below e, it decides to move in a straight line. In this example, this maneuver causes an overshoot, eventually causing the distance to be less than the threshold $-e$, and this triggers the vehicle to switch to the mode `Left`.

Exercise 9.7: For the automated guided vehicle, consider the following additional constraint: once the vehicle starts moving straight, left, or right, it cannot change its direction before Δ minutes for a given constant Δ. How will you modify the hybrid process of figure 9.13 to capture this additional constraint? How will the trajectory of the vehicle be affected by this change? ■

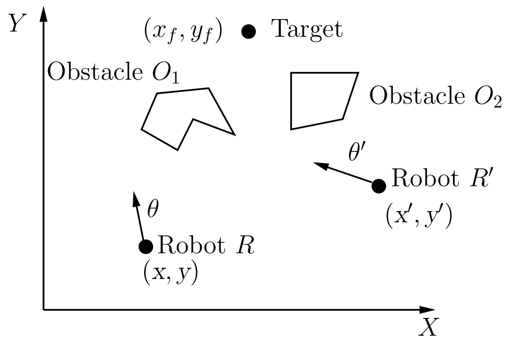


Figure 9.15: Path Planning in Presence of Obstacles for Two Robots

9.2.2 Obstacle Avoidance with Multi-robot Coordination

A challenging application domain for modeling and analysis of hybrid systems is the design of multi-robot coordination for a system of autonomous mobile robots. A typical surveillance task involves identifying a target and exploring a room with unknown geometry, possibly with obstacles, to reach the target. The sensory capabilities of each robot yield only imperfect information about the surroundings, and in particular, each robot has only estimates about the obstacle positions. The robots can send information to one another over wireless links and use this information to improve the accuracy of their estimates for better motion planning. The robots also need to coordinate with one another to achieve collaborative goals. For instance, it may be required that a team of robots should arrive at the same target, or the robots may be required to partition a set of targets among themselves. The solution to the design problem should ideally also be optimal among the space of solutions. For example, the objective can be to minimize either the total distance traveled by the robots or the time by which all the targets are reached. Thus, the design problem involves coordination, planning, and control in an optimal manner while satisfying the safety requirements and is representative of design problems in intelligent vehicle systems and flight management systems.

Illustrative Scenario

To illustrate modeling in a concrete scenario, suppose there are two autonomous mobile robots R and R'. We assume a two-dimensional world in which each robot is modeled as a point in the two-dimensional X-Y plane (see figure 9.15). The initial position of the robot R is (x_0, y_0), and the initial position of the robot R' is (x'_0, y'_0). The goal of each robot is to reach the target located at the position (x_f, y_f). Both the robots want to reach the target while minimizing the total distance traveled. Assume that both robots travel at a fixed speed, say v. Then the sole control input for each robot is the direction in which it is moving. If the state variables (x, y) specify the coordinates of the robot R, the variables (x', y') specify the coordinates of the robot R', the variable θ specifies the direction in which the robot R is headed, and the variable θ' specifies the direction in which the robot R' is headed, then the dynamics of the system is captured by the differential equations:

$$\dot{x} = v\,\texttt{cos}\,\theta; \quad \dot{y} = v\,\texttt{sin}\,\theta; \quad \dot{x}' = v\,\texttt{cos}\,\theta'; \quad \dot{y}' = v\,\texttt{sin}\,\theta'.$$

The room has obstacles, and this can prevent each robot from traveling in a straight line from its initial position to the target. More specifically, suppose the room has two obstacles and the areas occupied by the two obstacles are O_1 and O_2, respectively (see figure 9.15). Each robot has a camera that can detect the approximate position of each obstacle. The robots can also communicate their current estimates so that their joint knowledge can be used for more accurate information.

The safety requirement for the problem is that no robot should ever collide with an obstacle. That is, the following property should be an invariant of the system:

$$[\,(x, y) \notin O_1 \;\wedge\; (x, y) \notin O_2 \;\wedge\; (x', y') \notin O_1 \;\wedge\; (x', y') \notin O_2\,].$$

The liveness requirement is that each robot should eventually reach its target:

$$\Diamond\,[\,(x, y) = (x_f, y_f)\,] \;\wedge\; \Diamond\,[\,(x', y') = (x_f, y_f)\,].$$

Estimating Obstacles

Mapping obstacles accurately using images from a camera is a computationally expensive task. Furthermore, optimal path planning to a target location given complex descriptions of obstacles is also computationally expensive. To address these difficulties, let us estimate each obstacle using a circle. In figure 9.16, the actual obstacle is a concave polygon occupying the area O. The circle of radius r contains this area entirely and is the best possible circular approximation of the obstacle. The image-processing algorithm on board a robot then simply needs to return the parameters of the circle, and this does not require detecting edges of the obstacle accurately. The planning algorithm has to compute a path to the target that avoids the circular shapes, and such a path is guaranteed to avoid collisions with the actual obstacles, thereby satisfying the safety requirement, although it may not be the shortest such path to the target.

In vision applications, the accuracy of obstacle estimation is limited by several factors, and the estimates improve as the distance to the target decreases. This is particularly true when sonars are used for obstacle detection. In figure 9.16, the obstacle is a circle with the center (x_o, y_o) and radius r. The estimate by a robot with the current position (x_1, y_1) is a circle that is concentric with the obstacle circle but with radius e_1 that exceeds r. This estimate depends on the distance d_1 between the obstacle center (x_o, y_o) and the robot position (x_1, y_1).

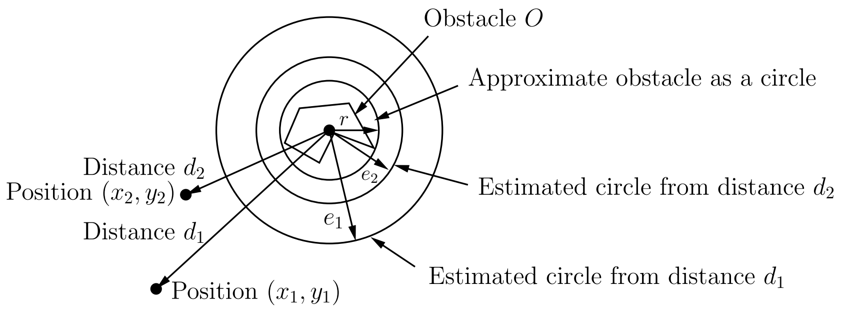

Obstacle O
Approximate obstacle as a circle
r
e_2
e_1
Distance d_2
Position (x_2, y_2)
Distance d_1
Position (x_1, y_1)
Estimated circle from distance d_2
Estimated circle from distance d_1

Figure 9.16: Approximate Estimation of an Obstacle

As the robot moves to the position (x_2, y_2), which is distance d_2 away from the obstacle center, the estimate improves to another concentric circle of radius e_2. As $(d_2 < d_1)$, we have $(e_2 < e_1)$. As the robot approaches the obstacle boundary, the distance between the robot position and the center of the obstacle decreases, and the estimate becomes more and more accurate, converging to the correct value of the obstacle radius. We assume the dependence of the estimate on the distance to be linear. If d is the current distance between the robot position and the obstacle center and r is the obstacle radius, then the estimated radius e is given by the equation

$$e \;=\; r + a\,(d - r)$$

where $0 < a < 1$ is a constant.

In our example scenario of figure 9.15, we have two obstacles. The first obstacle is modeled as a circle with the center (x_o^1, y_o^1) and radius r_1, and the second obstacle is modeled as a circle with the center (x_o^2, y_o^2) and radius r_2. Each robot can compute the estimates of each of the two obstacles based on the distance of its current position from the centers of the two obstacles. Furthermore, obstacle estimation is a computationally expensive process. As a result, the estimates are updated only discretely, every t_e seconds.

Path Planning

Consider the robot R with the current position (x, y). Its goal is to reach the target (x_f, y_f) while avoiding the two obstacles. For the obstacle O_1, the area it occupies according to the robot R is a circle centered at (x_o^1, y_o^1) with the current estimated radius e_1. Similarly, for the obstacle O_2, the area it occupies according to the robot R is a circle centered at (x_o^2, y_o^2) with the current estimated radius e_2. The objective of the planner is to compute a shortest path from the current position to the target so that the trajectory does not intersect the estimated obstacle circles.

The plan is usually updated in a discrete manner. In our design, the planning algorithm is invoked every t_p seconds, and the planning algorithm determines

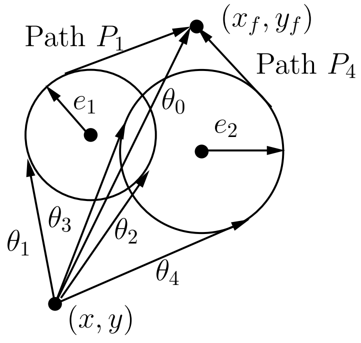


Figure 9.17: Path Planning While Avoiding Circular Obstacles

the control input θ that gives the direction for the robot motion. The direction stays unchanged until the next time the planning algorithm is invoked. We assume that the planning algorithm is captured by the function `plan` that takes as inputs (1) the current position (x, y), (2) the target position (x_f, y_f), (3) the first obstacle circle given by the center (x_o^1, y_o^1) and radius e_1, and (4) the second obstacle circle given by the center (x_o^2, y_o^2) and radius e_2 and returns the direction θ in which the robot should head.

The first step of the planning algorithm is to decide if the straight line from the current position (x, y) to the target (x_f, y_f) intersects any of the two estimated obstacle circles. If not, then the chosen direction is along this straight line. If it does, as is the case in figure 9.17, then the planner considers rays that are tangents from the current position (x, y) to the two obstacle circles. These are shown as directions θ_1 and θ_2 tangential to the first obstacle and directions θ_3 and θ_4 tangential to the second obstacle in figure 9.17. A direction that is tangential to one but intersects the other is discarded. Among the remaining choices, the direction that minimizes the distance to the target is chosen. In figure 9.17, the tangential directions θ_2 and θ_3 are not viable as they intersect with the other obstacle. The direction θ_1 corresponds to the path P_1 to the target, and the direction θ_4 specifies the path P_4 to the target. Since the length of the path P_1 is shorter than the path P_4, the planner returns the direction θ_1.

Note that the robot does not actually follow the path P_1; rather, it starts moving in the direction θ_1. When the planner is invoked again, if the estimates have improved by then, then it can revise its choice. In particular, in our example, as the robot moves along the direction θ_1, it would acquire an improved estimate of the first obstacle as a circle with a shorter radius; as a result, the robot would decrease the value of θ_1 in the clockwise direction and thereby head closer to the actual obstacle.

The function `plan` involves a sequence of floating-point calculations necessary to determine tangents and intersections. Such code is typically written in C or

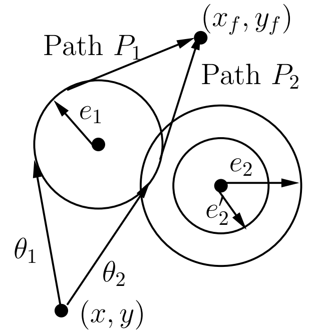


Figure 9.18: Impact of Improved Estimates via Coordination on Path Planning

Matlab and the update description in the model-based design framework calls this function.

Coordination

To understand the impact of coordination, let us revisit the planning example of figure 9.17 as shown in figure 9.18. Suppose the robot R' is closer to the second obstacle and, thus, has a better estimate e'_2 of the radius of this obstacle. If the robot R' communicates this information to the robot R, then the robot R can simply update its value of the estimate e_2 to e'_2. Using this revised estimate, the planner now concludes that the direction θ_2 tangential to the first obstacle is a viable option since it does not intersect the circle with the center (x^2_o, y^2_o) and radius e'_2. The path P_2 corresponding to this choice is shorter than the path P_1 (see figure 9.18); as a result, the planner chooses the direction θ_2, leading to a more optimal solution.

The coordination strategy in this example is simple: every t_c seconds, the robot R sends its estimates e_1 and e_2 of the obstacles' radii to the robot R', and whenever it receives estimates (e'_1, e'_2) of the obstacles' radii from the other robot, it updates the estimate value e_1 to the minimum of its own current estimate e_1 and the received value e'_1, and the estimate value e_2 to the minimum of its own current estimate e_2 and the received value e'_2 (since a smaller radius is an improved estimate).

Hybrid Model

We proceed to describe the model of the robot as a hybrid process. We will describe the model for the robot R, and the model for the robot R' is symmetric and can be obtained by instantiation.

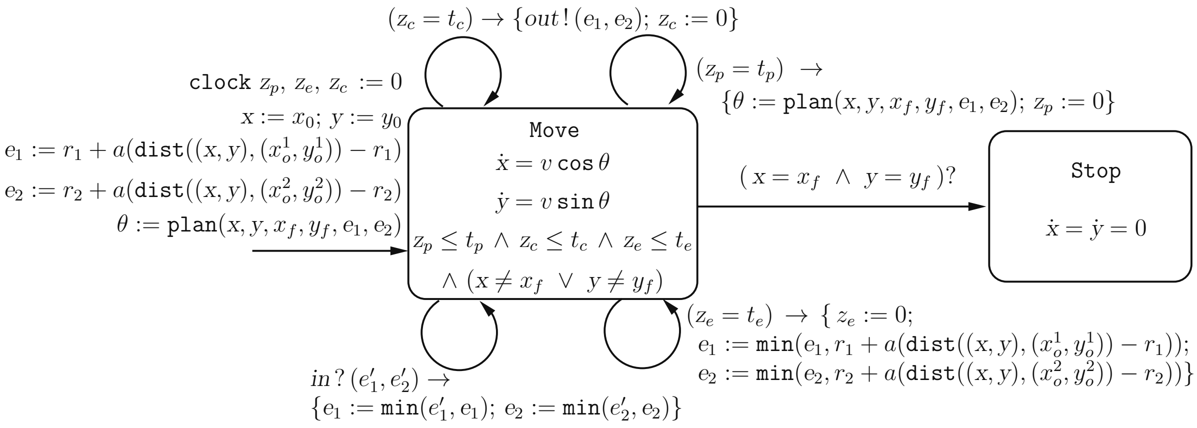
The hybrid process for the robot R is shown in figure 9.19. It uses the following variables:

- It has one input channel *in* of type ($\texttt{real} \times \texttt{real}$) that is used to receive the estimates of the radii of the obstacles from the other robot.
- It has one output channel *out* of type ($\texttt{real} \times \texttt{real}$) that is used to send the estimates of the radii of the obstacles to the other robot.
- The continuously updated state variables x and y, of type `cont`, model the position of the robot. These variables are initialized to x_0 and y_0, respectively.
- The discretely updated state variables e_1 and e_2, both of type `real`, capture the current estimates of the radii of the two obstacles. The initial estimates are obtained by executing the obstacle estimation algorithm and depend on the distance between the initial robot position and the centers of the two obstacles.
- The discretely updated state variable θ ranging over the interval $[-2\pi, 2\pi]$ models the direction in which the robot is currently moving. The initial direction is obtained by executing the function `plan`.
- The continuously updated state variable z_p, initialized to 0, is a clock variable that is used to enforce the timing constraint for invoking the planning algorithm every t_p seconds.
- The continuously updated state variable z_e, initialized to 0, is a clock variable that is used to enforce the timing constraint for updating the estimates using the vision data every t_e seconds.
- The continuously updated state variable z_c, initialized to 0, is a clock variable that is used to enforce the timing constraint for communicating the estimates every t_c seconds.

The process has two modes: `Move` and `Stop`. Initially, the mode is `Move`. During a timed action, the three clock variables z_p, z_e, and z_c increase at the rate 1, and the position variables x and y are updated at rates $v\,\texttt{cos}\,\theta$ and $v\,\texttt{sin}\,\theta$, respectively.

The switches from this mode are the following:

- When the robot reaches its target, captured by the condition ($x = x_f \;\wedge\; y = y_f$), it switches to the mode `Stop`.
- When an input (e'_1, e'_2) is received on the input channel *in*, the obstacle estimates e_1 and e_2 are updated to the minimum of the current and the received values.
- When the clock z_p reaches the value t_p, the direction θ is updated by invoking the planning function `plan` based on the current estimates, and the clock z_p is reset to 0.



$(z_c = t_c) \rightarrow \{out!(e_1, e_2);\ z_c := 0\}$
$(z_p = t_p) \rightarrow \{\theta := \mathtt{plan}(x, y, x_f, y_f, e_1, e_2);\ z_p := 0\}$
$\mathtt{clock}\ z_p,\ z_e,\ z_c := 0$
$x := x_0;\ y := y_0$
$e_1 := r_1 + a(\mathtt{dist}((x,y),(x_o^1, y_o^1)) - r_1)$
$e_2 := r_2 + a(\mathtt{dist}((x,y),(x_o^2, y_o^2)) - r_2)$
$\theta := \mathtt{plan}(x, y, x_f, y_f, e_1, e_2)$
Move
$\dot{x} = v \cos\theta$
$\dot{y} = v \sin\theta$
$z_p \le t_p \wedge z_c \le t_c \wedge z_e \le t_e$
$\wedge\ (x \ne x_f \vee y \ne y_f)$
$(x = x_f \wedge y = y_f)?$
Stop
$\dot{x} = \dot{y} = 0$
$(z_e = t_e) \rightarrow \{ z_e := 0;$
$e_1 := \mathtt{min}(e_1, r_1 + a(\mathtt{dist}((x,y),(x_o^1, y_o^1)) - r_1));$
$e_2 := \mathtt{min}(e_2, r_2 + a(\mathtt{dist}((x,y),(x_o^2, y_o^2)) - r_2))\}$
$in?(e_1', e_2') \rightarrow$
$\{e_1 := \mathtt{min}(e_1', e_1);\ e_2 := \mathtt{min}(e_2', e_2)\}$

Figure 9.19: The Hybrid State Machine for the Robot

- When the clock z_c reaches the value t_c, the current values of the estimates e_1 and e_2 are transmitted on the output channel *out*, and the clock z_c is reset to 0.

- When the clock z_e reaches the value t_e, the current values of e_1 and e_2 are updated by executing the vision-based obstacle estimation algorithm, and the clock z_e is reset to 0. As discussed earlier, the effect of this algorithm is captured by computing the distance between the current robot position and the centers of the two obstacles and updating the estimate values if the revised estimates are better.

The continuous-time invariant of the mode `Move` ensures that when the conditions for updates corresponding to one of the discrete switches is satisfied, the elapse of time is interrupted to execute the corresponding discrete action. When the process is in the mode `Stop`, the robot simply waits there.

The desired system is the parallel composition of the two robots. The system description involves a large number of parameters. The system needs to be simulated many times to choose values for the parameters from possible choices. In particular, we would like to find out the value of t_c that determines how often the robots should communicate so that the communication actually improves the distance traveled.

Illustrative Execution

Figure 9.20 shows sample executions of the model obtained by simulating the model in STATEFLOW/SIMULINK. The initial position of the robot R is $(4.5, 2)$, the initial position of the robot R' is $(10, 2)$, and the target is at the position $(6, 10)$. The obstacle O_1 is centered at $(3.7, 7.5)$ and has a radius 0.9, and the

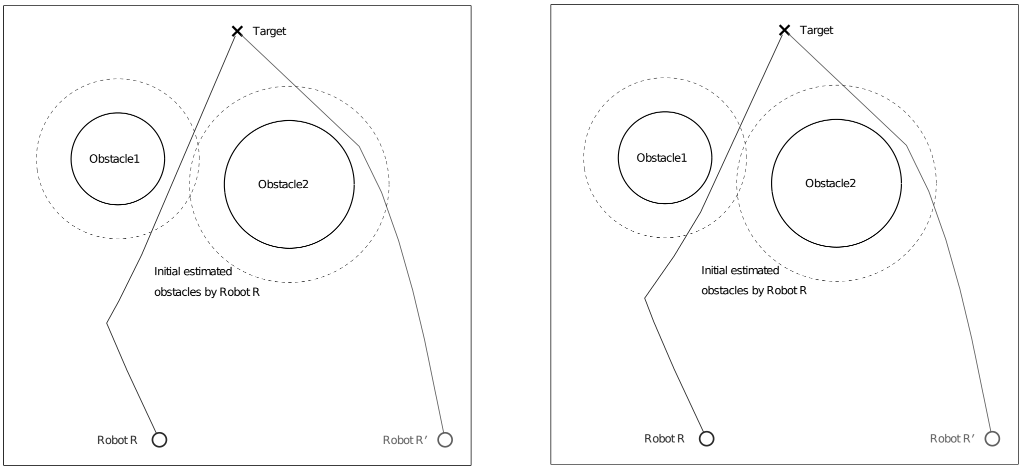


Figure 9.20: Illustrative Execution for Obstacle Avoidance

obstacle O_2 is centered at $(7, 7)$ and has a radius 1.25. The speed v is set to 0.5 units/sec, and the coefficient a used in the obstacle estimation is 0.12. The value of t_p, the time period for executing the planning algorithm, is 2 seconds, and the value of t_e, the time period for updating the obstacle estimates, is 2 seconds. The value of t_c, the time period for communicating the obstacle estimates, is different for the two executions: the left execution is obtained by setting t_c to 4 seconds, and the right execution is obtained by setting t_c to a high value.

Based on the initial estimates of the robot R, the two obstacles seem to overlap. As it gets closer to the obstacles, the estimates improve, suggesting a route to the target that passes between the two obstacles. For the robot R', the planned route does not exhibit a such qualitative change, but note that its estimate of the second obstacle constantly improves as it moves, leading to a curved trajectory. Observe that the distance traveled by the robot R is smaller in the left scenario as a result of communication. This is because it receives a better estimate of the second obstacle from the robot R' and switches its route a bit earlier thanks to this collaboration. In particular, the distance traveled by the robot R is 8.8136 with communication and is 8.6480 in the absence of communication (the distance traveled by the robot R' is 9.1550 in both the scenarios).

Exercise 9.8: For the problem of obstacle avoidance with coordination, consider the following optimization that reduces the needed computation. If a robot determines that the straight-line path from its current position to the target does not intersect any of the obstacles based on its current estimates of the obstacles, such a path cannot further be improved, and the robot can simply decide to move in this direction with no further planning. Modify the model of figure 9.19 so as to include this optimization. ■

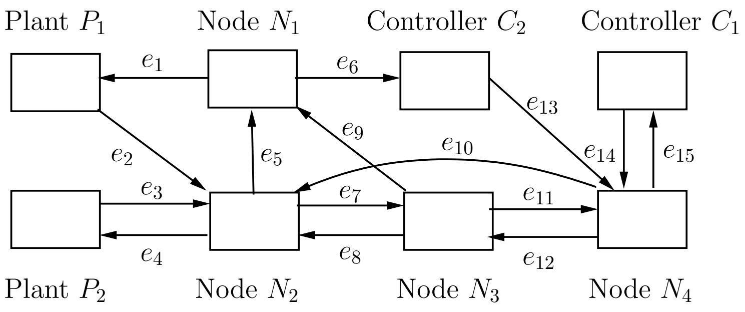


Figure 9.21: Example Multi-hop Control Network

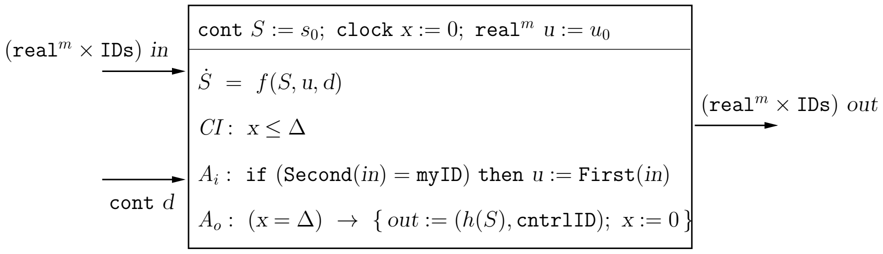
9.2.3 Multi-hop Control Networks *

The classical architecture of a feedback control loop is shown in figure 6.1. In contrast, wireless networked control systems are spatially distributed systems where the communication among sensors, actuators, and computational units is supported by a shared wireless communication network. The deployment of such networked control systems in industrial automation results in flexible architectures and typically reduces the costs for installation, debugging, diagnostics, and maintenance when compared with the classical wired control loops. Design of controllers in the networked architecture faces new challenges. First, communication between a plant and the corresponding controller involves multiple hops and, thus, significant time delays. Second, multiple control loops may share the same network link, leading to mutual dependencies. Thus, the design of control laws for adjusting the plant inputs, the routing policies for transmission of messages through the network, and the scheduling policies for sharing network links must evolve in a synergistic manner. We now describe how to model such multi-hop control networks formally using the modeling concepts discussed in this textbook.

Example Network

Figure 9.21 shows a sample network. It consists of two plants P_1 and P_2 and their corresponding controllers C_1 and C_2. Messages among the plants and controllers are routed over a network consisting of four nodes N_1, N_2, N_3, and N_4. The links $e_1, e_2, \dots e_{15}$ between different components are directed. For example, the output of the plant P_1 can be sent to the node N_2 over the link e_2, and the network can forward such messages to the controller C_1 using the links e_7, e_{11}, and e_{15}.

To be deployed in the context of control applications, the network must provide real-time guarantees regarding delivery of messages. In our set up, let us assume that one message can be delivered over a given link in a duration of time Δ. In other words, time is divided into slots with each slot of duration Δ time units.



(real^m × IDs) *in*
cont *d*
cont $S := s_0$; clock $x := 0$; real^m $u := u_0$
$\dot{S} = f(S, u, d)$
CI: $x \leq \Delta$
A_i: if (Second(in) = myID) then $u :=$ First(in)
A_o: $(x = \Delta) \rightarrow \{ out := (h(S),$ cntrlID$); x := 0\}$
(real^m × IDs) *out*

Figure 9.22: Plant Model in Multi-hop Control Network

At the beginning of each time slot, each node can send one message on each of its outgoing links. The message on each link can be received by the target component at the end of the time slot. Such a network is called a *time-triggered* network. The emerging WirelessHART standard for wireless networks provides such an abstraction and is being increasingly deployed within industrial process control.

Plant Model

The hybrid process modeling a plant in a multi-hop control network is shown in figure 9.22. The plant maintains state variables S that are updated continuously. The state is initialized to the value s_0 and evolves according to the differential equation $\dot{S} = f(S, u, d)$ during a timed action, where the variable u denotes the controlled input and the variable d denotes the uncontrolled input (or the disturbance). The disturbance d is a continuously updated external input signal.

The controlled input u, unlike in the models of continuous-time components in chapter 6, is updated only discretely, when the process receives a new value on the input channel *in*. Let us assume that the input u is an m-dimensional vector. The process model needs to store the value of this variable in its internal state. A message communicated over the network is a pair (v, id), consisting of a value v and the identifier id of the destination of the message. Let `IDs` denote the set of identifiers of all the plants and controllers that are connected by the network. The description of the plant process is then parameterized by its own identifier, denoted `myID`. The processing of an input over the channel *in* is then specified by the input task A_i: whenever it receives a message (v, id) on the input channel, it checks if the identifier id equals its own identifier; if so, it updates the value of the control input stored in the state variable u to the value v. If the channel *in* receives the sequence of values $v_1, v_2, \ldots$ destined for this plant at times $t_1, t_2, \ldots$ respectively, then the evolution of the variable u is a piecewise-constant signal whose value during the interval $[t_i, t_{i+1})$ is v_i, for each $i \geq 1$.

The function h maps the plant state to its output. Let us assume that the plant output is also an m-dimensional vector. Then every message exchanged on the network is of type (`real`^{m} $\times$ `IDs`). The output of the plant is transmitted on the channel *out*. Since each network link can carry only one message every Δ time units, the plant should send a message every Δ time units on the channel *out*. To capture this timing constraint, we use a clock variable x. The clock is initialized to 0. The clock-invariant associated with the process is the condition $(x \leq \Delta)$, and the guard associated with the output task A_o responsible for sending messages on the channel *out* is $(x = \Delta)$. These two together ensure that the message is transmitted exactly every Δ time units. The value of the message to be transmitted is computed by applying the output map h to the plant state. The destination of the message is represented by the parameter `cntrlID`, which is the identifier of the controller responsible for this specific plant.

For the network shown in figure 9.21, we need two instances of the plant process. One is instantiated with `myID` $= P_1$, `cntrlID` $= C_1$, *in* $= e_1$, and *out* $= e_2$, and the other is instantiated with `myID` $= P_2$, `cntrlID` $= C_2$, *in* $= e_4$, and *out* $= e_3$. In each case, the model is completed by filling in the details of the dynamics f and the output map h.

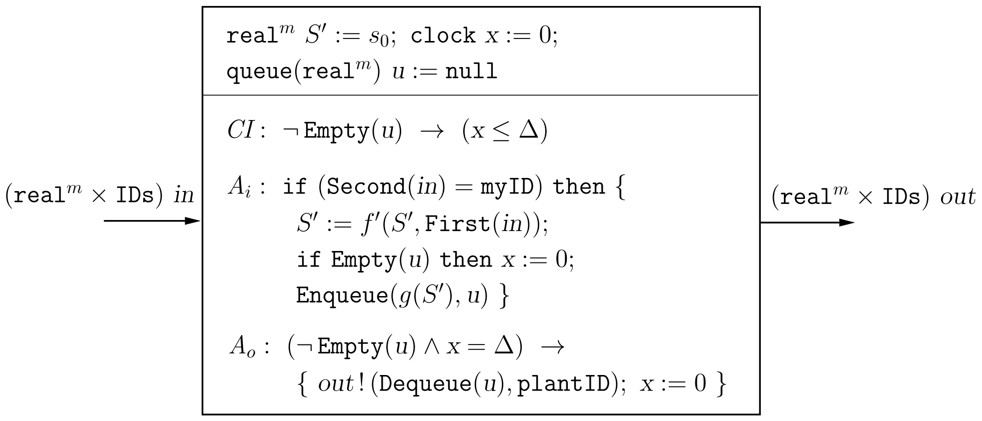
Controller Model

The timed process modeling a controller in a multi-hop control network is shown in figure 9.23. The controller maintains an *estimate* of the state of the plant using the variables S'. The estimate is initialized to the initial plant state s_0.

The input to the controller is the channel *in* on which it receives the observed outputs of the plant communicated over the network. Recall that the messages are tagged with the destination identifier. Whenever the controller receives a message addressed to itself, it updates the state estimate S' based on the current estimate and the newly received plant observation. Based on this estimate, it computes an updated value of the control input $g(S')$ to be communicated back to the plant. This value is enqueued in the queue u that contains messages to be transmitted over the output channel.

A clock variable x is used to ensure that only one message is sent on the output channel every Δ time units. To achieve the desired behavior, we choose the clock invariant to state that "if an output message is waiting, then the clock should not exceed Δ." The output task is enabled when the queue is non-empty and the clock reaches Δ. Whenever a value is sent over the output channel, it is tagged with the identifier of the corresponding plant, denoted by the parameter `plantID`. The output task also resets the clock to 0. When the input task generates a new value to be transmitted, and thus to be enqueued in the output queue, it checks if the output queue is empty and, if so, resets the clock to 0 so that this new value will be transmitted after a delay of Δ time units.

If the controller receives an input only once every Δ time units, then the queue u contains one message most of the time. Consider the state in which the queue contains one message, the clock x equals Δ, and the process that sends messages


($\mathtt{real}^m \times \mathtt{IDs}$) *in*
($\mathtt{real}^m \times \mathtt{IDs}$) *out*

```
real^m $S' := s_0$; clock $x := 0$;
queue(real^m) $u :=$ null

$CI$: $\neg$Empty($u$) $\rightarrow$ ($x \leq \Delta$)

$A_i$: if (Second($in$) = myID) then {
        $S' := f'(S'$, First($in$));
        if Empty($u$) then $x := 0$;
        Enqueue($g(S')$, $u$) }

$A_o$: ($\neg$Empty($u$) $\wedge$ $x = \Delta$) $\rightarrow$
        { $out$!(Dequeue($u$), plantID); $x := 0$ }
```

Figure 9.23: Controller Model in Multi-hop Control Network

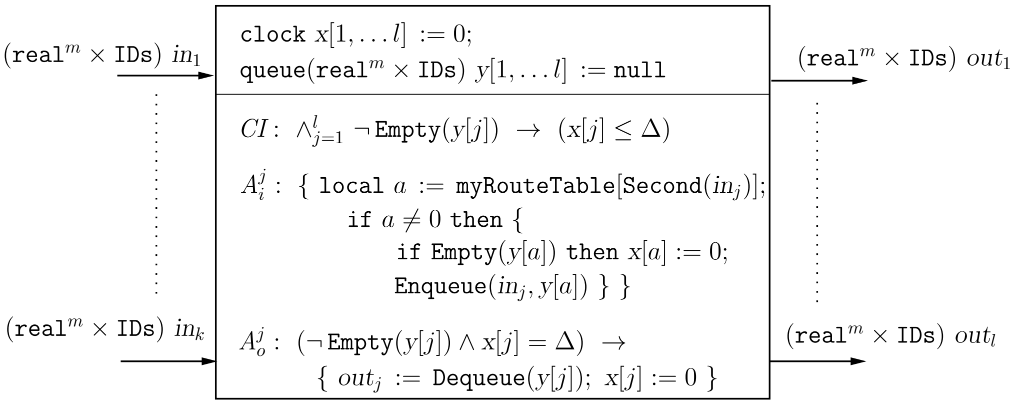
on the channel *in* is ready to transmit a message. In this case, both the input task A_i and the output task A_o are enabled and can execute in either order. No matter in which order they get executed, in the resulting state, the clock x is 0, and the queue contains one message that reflects the update of the controller's output in response to the value just received. If the controller is supplied inputs at a rate higher than once per Δ time units, then the number of messages waiting in the queue u will keep growing, and this scenario should be avoided.

For the network shown in figure 9.21, we need two instances of the controller process. One is instantiated with $\mathtt{myID} = C_1$, $\mathtt{plantID} = P_1$, $in = e_{15}$, and $out = e_{14}$, and the other is instantiated with $\mathtt{myID} = C_2$, $\mathtt{plantID} = P_2$, $in = e_6$, and $out = e_{13}$. In each case, the model is completed by filling in the details of the state estimator function f' and the control map g.

Network Routing

Given the set of plants, controllers, network nodes, and directed links connecting them, we need to determine how to route the messages from each plant to the corresponding controller and back. This problem can be formalized as computing paths between multiple source-destination pairs in a directed graph and is a classical network routing problem. Ideally, we would like all the routes to be mutually *disjoint*. In such a case, transmission of messages along different routes can proceed independently. Additionally, shorter routes mean shorter end-to-end delays in transmission of messages, and hence shorter routes are preferred.

In the example network shown in figure 9.21, we need to determine routes from the plant P_1 to the controller N_1, from the controller C_1 to the plant P_1, from the plant P_2 to the controller N_2, and from the controller C_2 to the plant P_2. A good choice of such routes is:



```
clock $x[1,\dots l]$ := 0;
queue(real$^m$ × IDs) $y[1,\dots l]$ := null

$CI$: $\wedge_{j=1}^{l}$ ¬Empty($y[j]$) → ($x[j] \leq \Delta$)

$A_i^j$ : { local $a$ := myRouteTable[Second($in_j$)];
        if $a \neq 0$ then {
            if Empty($y[a]$) then $x[a]$ := 0;
            Enqueue($in_j$, $y[a]$) } }

$A_o^j$ : (¬Empty($y[j]$) ∧ $x[j] = \Delta$) →
      { $out_j$ := Dequeue($y[j]$); $x[j]$ := 0 }
```

Figure 9.24: Network Node Model in Multi-hop Control Network

- the four-hop path e_2, e_7, e_{11}, e_{15} from the plant P_1 to the controller C_1,
- the three-hop path e_3, e_5, e_6 from the plant P_2 to the controller C_2,
- the four-hop path e_{14}, e_{12}, e_9, e_1 from the controller C_1 to the plant P_1, and
- the three-hop path e_{13}, e_{10}, e_4 from the controller C_2 to the plant P_2.

Note that these four routes are indeed disjoint, and the length of each route equals the length of the shortest path for the corresponding source-destination pair.

The problem of finding a path of shortest length between a single source-destination pair in a directed graph can be solved efficiently in time linear in the size of the graph using classical graph-search algorithms. However, finding disjoint paths among multiple source-destination pairs in a directed graph cannot be solved efficiently, and the problem is known to be NP-complete. In the context of multi-hop control networks, the size of the graph is typically not large (in current industrial process control, a typical graph consists of tens of nodes), and thus it is possible to explore different alternatives in an exhaustive manner to find the desired routes. When multiple sets of disjoint routes are possible, the routing strategy should prefer shorter paths. However, since a solution consists of routes between multiple source-destination pairs, two solutions may be incomparable: the path between a plant-controller pair may be shorter in one solution than in the other, but the path between another plant-controller pair may be shorter in the second solution than in the first. In such a case, the choice among the different solutions can be based on analyzing the overall performance of the entire system in conjunction with the design of control laws.

Modeling Network Node

The timed process modeling a generic network node is shown in figure 9.24. The description is parameterized by (1) the number k of incoming links, (2) the number l of outgoing links, and (3) a routing table `myRouteTable` that maps the set `IDs` of message destinations to one of the outgoing links. For a destination id in the set `IDs`, if `myRouteTable`$[id]$ is a number j between 1 and l, then the message should be transmitted on the jth outgoing link, and if `myRouteTable`$[id]$ is 0, then that means the node is not expecting messages sent to the destination id, and the message should be simply ignored.

The process has an input channel in_j, for $j = 1, \ldots k$, corresponding to each incoming link, and an output channel out_j for $j = 1, \ldots l$, corresponding to each outgoing link. The state of the process has a queue for each output channel: the messages to be transmitted on the output channel out_j are stored in the queue $y[j]$.

The processing of the messages received on the input channel in_j is captured by the input task A_i^j, for $j = 1, \ldots k$. Whenever the process receives a message, it uses the destination of the message, available as the second field of the incoming message, and the routing table to choose the output channel on which the message should be propagated. If the routing table entry is 0, then the node is not expecting such a message, and the message is simply dropped. Otherwise, the message is enqueued in the corresponding queue.

The timing constraint that the process should send only one message every Δ time units on each output channel is enforced in a manner analogous to the controller model. For each output channel out_j, the process has a clock variable x_j. The clock invariant ensures that, for each $j = 1, \ldots l$, if an output message is waiting to be transmitted on the jth output channel, then the corresponding clock should not exceed Δ. The output task A_o^j corresponding to the j-th output channel is enabled when the corresponding queue $y[j]$ is non-empty and the corresponding clock $x[j]$ reaches Δ. The clock corresponding to an output channel is reset to 0 every time a message is transmitted on this channel by the output task and also when an input task enqueues a message in the corresponding queue when it is empty.

For the network of figure 9.21, we need four instances of the network process. For the process corresponding to the network node N_3, the number k of incoming links is 2, and the number l of outgoing links is 3. The input channels in_1 and in_2 are renamed to the link names e_7 and e_{12}, respectively. The output channels out_1, out_2, and out_3 are renamed to the link names e_8, e_9, and e_{11}, respectively. According to the routes that we chose, at this node, messages sent to the controller C_1 should be forwarded on the link e_{11}, and messages sent to the plant P_1 should be forwarded on the link e_9. The node N_3 does not appear on the routes to the controller C_2 and to the plant P_2. Thus, the routing table for the node N_3 should be specified as `myRouteTable`$[P_1] = 2$, `myRouteTable`$[P_2] = 0$, `myRouteTable`$[C_1] = 3$, and `myRouteTable`$[C_2] = 0$.

System Model

The desired system corresponding to the multi-hop control network is the parallel composition of the instances of all the plants, controllers, and network nodes. If each link appears in at most one route, then the traffic flows smoothly through the network. In our example network, the plant P_1 sends an output value every Δ time units on the link e_2. A value v sent at time t is transmitted by the node N_2 at time $(t+\Delta)$ on the link e_7, then by the node N_3 at time $(t+2\Delta)$ on the link e_{11}, and then by the node N_4 at time $(t+3\Delta)$ on the link e_{15}. The controller C_1 updates its internal estimate in response to this value, and the corresponding control value v' is transmitted on the link e_{14} at time $(t+4\Delta)$. This value is propagated by the node N_4 at time $(t+5\Delta)$ on the link e_{12}, then by the node N_3 at time $(t+6\Delta)$ on the link e_9, and then by the node N_1 at time $(t+7\Delta)$ on the link e_1. For the subsequent interval of length Δ, the plant P_1 uses this value as its control input.

When there are no shared links among different routes, each control loop can be analyzed independently. Consider the closed-loop system consisting of a plant and its controller. The state of this system then consists of the plant state variables S and their estimates S' maintained by the controller. Let us assume that the route from the plant to the controller consists of k_1 hops and the route from the controller to the plant consists of k_2 hops. Then a plant observation transmitted at time t is received by the controller at time $[t+(k_1-1)\Delta]$ time; if the controller computes a new control value at time t, then it is received by the plant at time $(t+k_2\Delta)$ time.

Let $t_1=\Delta$, $t_2=2\Delta$, ... be the sequence of times at which messages are processed. We know that in each interval $[t_i, t_{i+1})$, the control input for the plant stays constant. Let $\overline{d}$ be the input signal for the external disturbance. The state response of the system is then defined by the following rules:

- The state signal for the controller estimate $\overline{S'}(t)$ is a piecewise-constant signal. For the first k_1 slots, the controller does not receive any update, and thus, for $i<k_1$, the state s'_i during the interval $[t_i, t_{i+1})$ equals the initial state s_0. After this, at each time t_i, for $i \geq k_1$, the controller receives the plant output (k_1-1) slots earlier, that is, the value $h(s_{i-k_1+1})$. Thus, for $i \geq k_1$, the state s'_i during the interval $[t_i, t_{i+1})$ equals $f'(s'_{i-1}, h(s_{i-k_1+1}))$.

- The state signal for the plant state $\overline{S}(t)$ is a piecewise-continuous signal. For the first (k_1+k_2) slots, the controller does not receive any update for the controlled input. Thus, for $i<(k_1+k_2)$, the state signal $\overline{S}(t)$ during the interval $[t_i, t_{i+1})$ corresponds to the solution of the initial value problem with initial state s_i and dynamics $f(S, u_0, d)$, in response to the disturbance signal $\overline{d}([t_i, t_{i+1}))$. After this, at each time t_i, for $i \geq (k_1+k_2)$, the plant receives the controller's message that reflects its computation k_2 slots earlier. Thus, for $i \geq (k_1+k_2)$, the state signal $\overline{S}(t)$ during the interval $[t_i, t_{i+1})$ corresponds to the solution of the initial value problem with initial state s_i and dynamics $f(S, g(s'_{i-k_2}), d)$. That is, the disturbance is given by the signal $\overline{d}([t_i, t_{i+1}))$, and the controlled input stays constant equal to the value obtained by applying the control map g to its estimated state k_2 slots earlier.

When all the functions f, h, f', and g are linear, the analysis techniques discussed in chapter 6 for linear systems can be adapted to compute the closed-form solution for the state response and to check properties such as stability.

Exercise 9.9: Recall the design of a cruise controller from section 6.3.3. Figure 6.18 shows the response of the car to a PI controller. In exercise 6.21, you considered the response to the same controller of the model of the car on a graded road shown in figure 6.8 for the input signal $\overline{\theta}(t) = [\texttt{sin}\,(t/5)]/3$ (measured in radians). Now let us assume that the sensors measuring the speed communicate with the cruise controller over a (time-triggered) multi-hop network. In terms of the model discussed in section 9.2.3 assume that the output of the car is its velocity, the controller is the same PI controller used in section 6.3.3, the number of hops from the sensors to the controller is three, and the number of hops from the controller back to the plant is two. Using a simulation tool such as MATLAB, plot the velocity of the car using the parameters: the initial velocity v_0 is 0, the mass m is $100\,kg$, the coefficient of friction k is 50, the gravitational acceleration g is $9.8\,m/s^2$, the reference velocity r is $10\,m/s$, the proportional gain K_P is 600, the integral gain K_I is 40, and the time-step Δ of the network is 0.1 seconds. ■

Exercise 9.10*: Suppose the given multi-hop control network with two plants and two controllers is such that the route from the plant P_1 to the controller C_1 shares a link with the route from the controller C_1 back to the plant P_1. In fact, the network is such that this sharing cannot be avoided (the routes involving the plant P_2 and its controller C_2 are disjoint). If we use exactly the same models for the plants, the controllers, and the network nodes as described in section 9.2.3, describe the behavior of the entire system, and how the closed-loop behavior of the two control loops be affected. Discuss a possible modification to the models so that this undesirable behavior is avoided (hint: what happens if the plant P_1 sends its outputs every two slots while all other components stay unchanged?). How will this modification affect the closed-loop behavior of the two control loops? ■

9.3 Linear Hybrid Automata *

In chapter 7, we studied timed automata as a subclass of timed processes: a timed automaton restricts how clock variables are tested and updated, and these restrictions allow the development of symbolic reachability analysis using the data structure of difference-bounds matrices. With a similar motivation, we now consider the restriction of hybrid processes to a subclass known as *linear hybrid automata*. A linear hybrid automaton can be viewed as a generalization of a timed automaton. While a clock variable in a timed automaton can only be

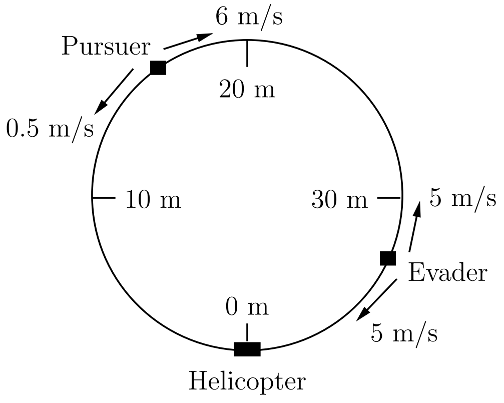


Figure 9.25: Pursuit Game

compared with a constant and reset to 0, continuously updated variables of a linear hybrid automaton are tested and updated using *affine* constraints. While a clock variable increases at the rate 1 during a timed action, a continuously updated variable in a linear hybrid automaton increases at a constant rate and, more generally, at a rate chosen from an interval with constant bounds. This structure then allows symbolic reachability analysis based on the representation of the sets of states by *polyhedra*.

Before we develop this model, it is worth emphasizing that the adjective "linear" in this context has a different meaning from its use in the classical model of "linear systems" that we studied in chapter 6. In a linear system, the rate of change of a state variable is a linear function of the system state, whereas in a linear hybrid automaton, the rate of change of a state variable is a constant or is bounded by a constant, which results in a state signal that is a linear function of time.

9.3.1 Example Pursuit Game

We illustrate the model with a two-player game of pursuit-evasion shown in figure 9.25. There is a pursuer in a golf cart chasing an evader on a circular track 40 meters long. The cart can travel upto 6 m/s in the clockwise direction but only upto 0.5 m/s in the counter-clockwise direction since it must use its reverse gear to travel counter-clockwise. The evader is on a bicycle and travels at 5 m/s in either direction. However, the evader makes a decision whether to change its direction only at fixed instances in time, separated by exactly 2 seconds. The goal of the evader is to avoid the pursuer. The evader has the added advantage that there is a rescue car at a fixed position on the track.

The evader uses a simple strategy: determine if the evader will win the race to the car if both players proceed clockwise at their respective full speeds, and if so, head clockwise and otherwise choose to move counterclockwise. The game, with this specific strategy for the evader, is shown as an extended-state machine in figure 9.26.

The continuously updated variable p models the position of the pursuer on the track measured in meters in a clockwise direction relative to the stationary car at position 0. Similarly, the continuously updated variable e models the position of the evader. The clock variable x measures the delay with respect to the most recent time instance when the evader chose the direction.

There are three modes: in the mode `ClkW`, the evader is moving in the clockwise direction on the track; in the mode `CntrClkW` the evader is moving in the counter-clockwise direction on the track; and in the mode `Rescued`, the evader has reached the car thus bringing the game to an end.

In the mode `ClkW`, the motion of the evader is described by the differential equation $\dot{e} = 5$. Regarding the evolution of the variable p that specifies the pursuer's position, we only know the bounds on the pursuer's speed in the two directions, and this is captured by the *differential inequality* $-0.5 \leq \dot{p} \leq 6$. This means that during a timed action of duration δ, the value of the variable p changes by an amount equal to $\delta\, c$ for a constant c belonging to the interval $[-0.5, 6]$. The constraint $(0 \leq p \leq 40) \wedge (0 \leq e \leq 40) \wedge (x \leq 2)$ labeling the mode `ClkW` is the continuous-time invariant and ensures that all the three continuously updated variables stay within their respective ranges during a timed action.

The specification of the mode `CntrClkW` is similar. The only difference is that the evader moves in the counter-clockwise direction, and its evolution is captured by the differential equation $\dot{e} = -5$. In the mode `Rescued`, the game has ended, so all the position variables stay unchanged.

The decision logic is captured by the mode-switches. Consider the mode `ClkW`. Whenever the value of the variable e equals 0 or 40, the evader has reached the car, and the switch to the mode `Rescued` is enabled. When the clock variable x equals 2, the evader compares the times needed by the two players to reach the car moving clockwise at their respective maximum speeds. If the time $(40 - e)/5$ the evader needs is less than the time $(40 - p)/6$ the pursuer needs, then the guard condition $(6e - 5p > 40)$ of the self-loop from the mode `ClkW` is enabled, and the mode continues to be `ClkW`; otherwise the guard condition $(6e - 5p \leq 40)$ of the mode-switch from the mode `ClkW` to the mode `CntrClkW` is enabled. Note that if the pursuer is between the evader and the car along the clockwise direction, that is, the condition $(e < p)$ holds, the evader is guaranteed to choose to switch to mode `CntrClkW`. In either case, the clock x is reset to 0. Note that the track is circular, and thus the positions 0 and 40 coincide. To capture this, if the value of the variable p is increasing and reaches 40, then it must be reset to 0; symmetrically, if the value of the variable p is decreasing and reaches 0, then it must be updated to 40. This explains the left self-loop on the mode `ClkW`.

The mode-switches originating from the mode `CntrClkW` are symmetric.

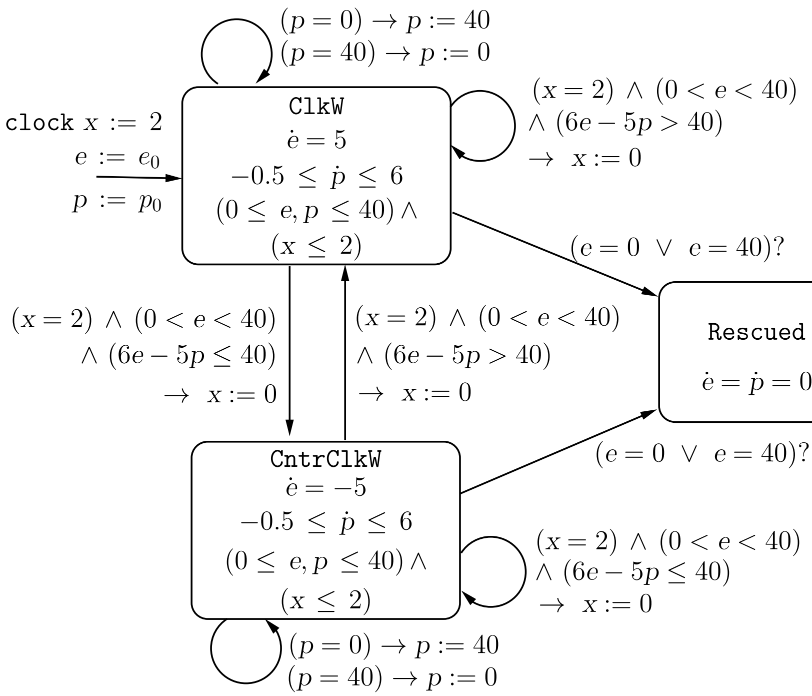


Figure 9.26: Linear Hybrid Automaton for the Pursuit Game

The initial position of the pursuer is p_0 and that of the evader is e_0. The clock is initialized to 2 so that the evader gets to make a decision about which direction to move at initial time.

During an execution, if the mode of the system ever becomes `Rescued`, then the evader wins by reaching the car. If the property $(e = p)$ becomes true at some time, then the evader loses. An execution may keep switching between the modes `ClkW` and `CntrClkW` forever with the property $(e \neq p)$ being true in all states belonging to the execution, and in such a case, the evader wins.

To illustrate the behavior of the model, consider the scenario with the initial positions $e_0 = 20$ and $p_0 = 1$. One possible execution from this initial state resulting in a win for the pursuer is shown below. A state is shown by listing the mode, followed by the values of the variables e, p, and x in that order:

$$(\texttt{ClkW}, 20, 1, 2) \xrightarrow{\varepsilon} (\texttt{ClkW}, 20, 1, 0) \xrightarrow{2} (\texttt{ClkW}, 30, 0, 2) \xrightarrow{\varepsilon}$$
$$(\texttt{ClkW}, 30, 40, 2) \xrightarrow{\varepsilon} (\texttt{CntrClkW}, 30, 40, 0) \xrightarrow{2} (\texttt{CntrClkW}, 20, 40, 2) \xrightarrow{\varepsilon}$$
$$(\texttt{CntrClkW}, 20, 40, 0) \xrightarrow{2} (\texttt{CntrClkW}, 10, 39, 2) \xrightarrow{\varepsilon} (\texttt{CntrClkW}, 10, 39, 0) \xrightarrow{0.17}$$
$$(\texttt{CntrClkW}, 9.17, 40, 0.17) \xrightarrow{\varepsilon} (\texttt{CntrClkW}, 9.17, 0, 0.17) \xrightarrow{0.83} (\texttt{CntrClkW}, 5, 5, 1).$$

In this scenario, the evader first moves clockwise for 2 seconds, during which time the pursuer moves counter-clockwise, resulting in a position where $e = 30$ and $p = 40$. Then the evader reverses direction moving counter-clockwise, during which time the pursuer stays stationary, resulting in a position where $e = 20$ and $p = 40$. The evader keeps moving counter-clockwise, during which time the pursuer moves counter-clockwise, resulting in a position where $e = 10$ and $p = 39$. The evader now still keeps moving counter-clockwise, but now the pursuer moves clockwise at full speed, and the two meet at 5 meters.

9.3.2 Formal Model

The formal definition of a linear hybrid automaton is a variation of the corresponding definitions of timed and hybrid processes. It consists of an asynchronous process some of whose state variables are of type `cont` and are updated continuously during a timed action. As in a timed process, we assume that all the input and output variables are updated only discretely. The dynamics of the continuously updated state variables is specified using a continuous-time invariant and a rate constraint.

The continuous-time invariant is a Boolean expression over the state variables, and a timed action is allowed only if all the states visited during the timed action satisfy the invariant. For the system of figure 9.26, the continuous-time invariant is:

$$(\mathit{mode} = \mathtt{Rescued}) \;\vee\; [\,(0 \leq\, e \,\leq 40) \;\wedge\; (0 \leq\, p \,\leq 40) \;\wedge\; (x \,\leq 2)\,].$$

The *linearity* restriction means that all the tests and updates involving the variables of type `cont` are affine expressions. More precisely, given variables $x_1, x_2, \ldots x_n$, an *affine test* is of the form $a_1\, x_1 + a_2\, x_2 + \cdots + a_n\, x_n \;\sim\; a_0$, where $a_0, a_1, \ldots a_n$ are (integer or real) constants, and $\sim$ is a comparison operation and can be either $<$, $\leq$, $=$, $>$, or $\geq$. An *affine assignment* is of the form $x_i \;:=\; a_0 + a_1\, x_1 + a_2\, x_2 + \cdots + a_n\, x_n$, where $a_0, a_1, \ldots a_n$ are (integer or real) constants. In a linear hybrid automaton, whenever an expression involving continuously updated variables appears in a guard, in the update code of a task, or in the continuous-time invariant, it must be an affine test, and every assignment to a continuously updated variable must be an affine assignment.

The rate constraint is a Boolean expression over the derivatives of the continuously updated variables and the discrete state variables. The expressions involving the derivatives must be affine. For the pursuit game of figure 9.26, the rate constraint is

$$\begin{aligned}
& (\,\mathit{mode} = \mathtt{ClkW}\,) \;\wedge\; (\,\dot{e} = 5\,) \;\wedge\; (\,-0.5 \,\leq\, \dot{p} \,\leq\, 6\,) \;\wedge\; (\,\dot{x} = 1\,) \\
\vee\;\; & (\,\mathit{mode} = \mathtt{CntrClkW}\,) \wedge (\,\dot{e} = -5\,) \wedge (\,-0.5 \,\leq\, \dot{p} \,\leq\, 6\,) \wedge (\,\dot{x} = 1\,) \\
\vee\;\; & (\,\mathit{mode} = \mathtt{Rescued}\,) \;\wedge\; (\,\dot{e} = 0\,) \;\wedge\; (\,\dot{p} = 0\,) \;\wedge\; (\,\dot{x} = 1\,).
\end{aligned}$$

To execute a timed action in a state s, we choose a rate vector r, that is, a constant r_x, for every continuously updated variable x so that the rate constraint is satisfied when evaluated in the state s using the values r_x for the derivatives $\dot{x}$. Given a time value t, let $s + t\,r$ denote the state s' such that the value of a discrete variable in the state s' coincides with its value in the state s, and the value of a continuously updated variable x in the state s' equals $s(x) + t\,r_x$. Then a timed action of duration δ can be executed if the state $s + t\,r$ satisfies the continuous-time invariant for every time value t in the interval $[0, \delta]$, and the state resulting from the timed action is the state $s + \delta\,r$.

The formal definition is summarized below:

LINEAR HYBRID AUTOMATON

A *linear hybrid automaton HP* consists of (1) an asynchronous process P, where some of its state variables can be of type `cont`, and appear only in affine tests and affine assignments in the guards and updates of the tasks of P; (2) a *continuous-time invariant CI*, which is a Boolean expression over the state variables S, where the variables of type `cont` appear only in affine tests; and (3) a *rate constraint RC*, which is a Boolean expression over the discrete state variables and the derivatives of the continuously updated state variables that appear only in affine tests. Inputs, outputs, states, initial states, internal actions, input actions, and output actions of the linear hybrid automaton *HP* are the same as that of the asynchronous process P. Given a state s and a real-valued time $\delta > 0$, $s \xrightarrow{\delta} s + \delta\,r$ is a *timed action* of *HP*, for a rate vector r consisting of a constant r_x for every continuously updated state variable x, if (1) the expression *RC* is satisfied when for every continuously updated variable x, the derivative $\dot{x}$ is assigned the value r_x, and every discrete variable x is assigned the value $s(x)$; and (2) the state $s + t\,r$ satisfies the expression *CI* for all values $0 \leq t \leq \delta$.

Note that during a timed action, each continuously updated variable x evolves at a constant rate, and thus, during the duration of the timed action, the signal $\overline{x}$ is a linear function of time. In contrast, in a linear system, a typical differential equation is of the form $\dot{x} = a\,x$, and the corresponding signal $\overline{x}(t) = x_0 e^{at}$ is an exponential function of time.

As already seen in the example of the pursuit game, a rate constraint can be used to specify bounds on the rate of change of a variable. The definition of linear hybrid automata also allows constraints involving rates of two variables. For example, if the variables (x, y) denote the position of a robot in a plane, then the rate constraint

$$(1 \leq \dot{x} \leq 2) \;\wedge\; (\dot{x} = \dot{y})$$

specifies that the robot is moving along the diagonal (due to the constraint $\dot{x} = \dot{y}$) at a constant speed for which we know a lower and an upper bound (due to the constraint $1 \leq \dot{x} \leq 2$).

Notions such as executions and reachable states for a linear hybrid automaton are defined in the same way as for a hybrid process.

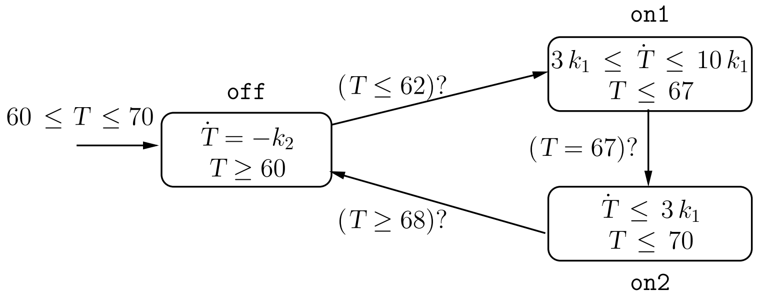


Figure 9.27: Linear Hybrid Automaton Model of a Thermostat

Note that since the rate constraint in the description of a linear hybrid automaton allows differential inequalities, syntactically it is not a hybrid process. However, the model of hybrid processes is strictly more expressive than linear hybrid automata as it is straightforward to capture the same behavior as that of a given linear hybrid automaton by introducing auxiliary variables corresponding to the rates that are updated discretely and nondeterministically. For example, to capture the differential constraint $(-0.5 \le \dot{p} \le 6)$ in the description of the automaton of figure 9.26, we introduce a discrete variable r_p of type `real`, set its value using the nondeterministic assignment $r_p :=$ `choose`$\{v \mid -0.5 \le v \le 6\}$ during a mode-switch, and specify the dynamics of the state variable p by the linear differential equation $\dot{p} = r_p$.

Exercise 9.11: Consider the linear hybrid automaton shown in figure 9.27 that models the behavior of a thermostat. Describe the possible executions of this model and explain how this model differs from the hybrid process of figure 9.1. ■

Exercise 9.12*: Design an alternative strategy for the evader that can still be described as a linear hybrid automaton but is "better" than the strategy shown in figure 9.26. It is still required that the evader makes its decisions only every two seconds; once it decides to move clockwise or counter-clockwise, it moves in that direction at full speed. Thus, the only permissible change in the linear hybrid automaton modeling the revised strategy compared to the automaton of figure 9.26 is in the test $(6e - 5p > 40)$ that is used to determine whether to move clockwise or counter-clockwise. Your alternative strategy should be such that the evader wins the game starting from the initial position $e_0 = 20$ and $p_0 = 1$. Is your strategy optimal for the evader (that is, is it the case that for every initial position, if your strategy results in a loss for the evader, then in every alternative strategy also results in a loss for the evader from this initial position)? ■

9.3.3 Symbolic Reachability Analysis

Now we develop an algorithm for the invariant verification problem for linear hybrid automata based on symbolic reachability analysis. The high-level algorithm is the same as the one discussed in section 3.4. To implement the symbolic search algorithm, we need a representation for regions, that is, for sets of states, that is suitable in the context of linear hybrid automata. For this purpose, let us assume that all the discrete variables have enumerated types as is the case for the pursuit game of figure 9.26. Then the linear hybrid automaton has only finitely many discrete states. The reachability algorithm then analyzes the discrete variables by enumerating their values and the continuously updated variables in a symbolic manner using affine constraints.

Affine Formulas

Let V be a set of variables that is partitioned into two sets: the set V_d of discrete variables of enumerated types and the set V_c of continuously updated variables of type `real`. A state over V then consists of a discrete state that assigns values to all the variables of enumerated types and an $|V_c|$-dimensional real-valued vector. A region A over (V_c, V_d) is then represented by a formula in the *disjunctive normal form* built using affine constraints over V_c and equality constraints over V_d.

Formally, the type `AffForm` is parameterized by the variable sets (V_c, V_d) and consists of affine formulas defined by the following rules:

- An atomic affine formula over (V_c, V_d) is an equality of the form $(x = d)$, where x is a discrete variable in V_d and d is a constant belonging to the type of the variable x, or an affine constraint of the form $(a_1x_1 + a_2x_2 + \cdots + a_nx_n \sim a_0)$, where $x_1, x_2, \ldots x_n$ are real-valued variables in V_c, $a_0, a_1, \ldots a_n$ are real numbers, and $\sim$ is a comparison operator from the set $\{<, \leq, =, >, \geq\}$.

- A conjunctive affine formula φ over (V_c, V_d) is a conjunction $\varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_k$, where the conjuncts $\varphi_1, \varphi_2, \ldots \varphi_k$ are atomic affine formulas over (V_c, V_d).

- An affine formula A over (V_c, V_d) is a disjunction $\varphi_1 \vee \varphi_2 \vee \cdots \vee \varphi_l$, where the disjuncts $\varphi_1, \varphi_2, \ldots \varphi_l$ are conjunctive affine formulas over (V_c, V_d).

Note that the syntax of affine formulas does not allow negation explicitly, but it can be expressed. For example, the constraint $\neg(x = d)$, where x is a variable in V_d and d is a constant, is equivalent to the affine formula $(x = d_1) \vee (x = d_2) \vee \cdots \vee (x = d_a)$, if the constants $d_1, d_2, \ldots d_a$ are all the values belonging to the type of the variable x other than the value d. The constraint $\neg(a_1x_1 + \cdots + a_nx_n \leq a_0)$, where $x_1, x_2, \ldots x_n$ are real-valued variables in V_c and $a_0, a_1, \ldots a_n$ are real numbers, is equivalent to the affine formula $(a_1x_1 + \cdots + a_nx_n > a_0)$.

Symbolic Representation of a Linear Hybrid Automaton

Having fixed the representation of affine formulas, now we can encode the various components of a linear hybrid automaton using this symbolic representation. For a given linear hybrid automaton *HP*, let S_d denote the set of its discrete variables and let S_c be the set of its continuously updated real-valued variables. We are restricting attention to the case where each variable in the set S_d has an enumerated type. For the pursuit game of figure 9.26, the set S_d contains the variable *mode*, and the set S_c contains the variables e, p, and x.

The set of initial states of the automaton *HP* is represented by a formula *Init* of type `AffForm` over (S_c, S_d). For the pursuit game, assuming the initial position e_0 of the evader is 20 and the initial position p_0 of the pursuer is 10, the formula *Init* equals

$$(\,\mathit{mode} = \texttt{ClkW}\,) \;\wedge\; (\,x = 2\,) \;\wedge\; (\,e = 20\,) \;\wedge\; (\,p = 10\,).$$

The transition relation of the asynchronous process corresponding to the automaton *HP* is represented by a formula *Trans* of type `AffForm` over $(S_c \cup S_c', S_d \cup S_d')$. Here, for every variable x, its primed version x' denotes the value of the variable x after executing the transition as explained in section 3.4. This transition relation captures all the input, output, and internal actions of the underlying asynchronous process and, thus, the set of all discrete transitions of the automaton *HP*. For the pursuit game, the formula *Trans* has a disjunct for every mode-switch of the automaton of figure 9.26. For instance, the disjunct corresponding to the mode-switch from the mode `ClkW` to the mode `CntrClkW` is

$$\begin{aligned}&(\,\mathit{mode} = \texttt{ClkW}\,) \;\wedge\; (\,x = 2\,) \;\wedge\; (\,e > 0\,) \;\wedge\; (\,e < 40\,) \;\wedge\; (\,6e - 5p \leq 40\,) \;\wedge\\ &\qquad (\,\mathit{mode}' = \texttt{CntrClkW}\,) \;\wedge\; (\,x' = 0\,) \;\wedge\; (\,e' - e = 0\,) \;\wedge\; (\,p' - p = 0\,).\end{aligned}$$

The continuous-time invariant of the automaton *HP* is represented by a formula *CI* of type `AffForm` over (S_c, S_d). The continuous-time invariant for the pursuit game is captured by:

$$(\,\mathit{mode} = \texttt{Rescued}\,) \vee [\,(\,e \geq 0\,) \wedge (\,e \leq 40\,) \wedge (\,p \geq 0\,) \wedge (\,p \leq 40\,) \wedge (\,x \leq 2\,)\,].$$

The rate constraint of the automaton *HP* is represented by a formula *RC* of type `AffForm` over $(\dot{S}_c, S_d)$. Here, for every real-valued variable x, the real-valued variable $\dot{x}$ represents its rate of change with time. The rate constraint for the pursuit game is given by:

$$\begin{aligned}&[(\,\mathit{mode} = \texttt{ClkW}\,) \;\wedge\; (\,\dot{e} = 5\,) \;\wedge\; (\,\dot{p} \geq -0.5\,) \;\wedge\; (\,\dot{p} \leq 6\,) \;\wedge\; (\,\dot{x} = 1\,)]\\ \vee\;\; &[(\,\mathit{mode} = \texttt{CntrClkW}\,) \wedge (\,\dot{e} = -5\,) \wedge (\,\dot{p} \geq -0.5\,) \wedge (\,\dot{p} \leq 6\,) \wedge (\,\dot{x} = 1\,)]\\ \vee\;\; &[(\,\mathit{mode} = \texttt{Rescued}\,) \;\wedge\; (\,\dot{e} = 0\,) \;\wedge\; (\,\dot{p} = 0\,) \;\wedge\; (\,\dot{x} = 1\,)].\end{aligned}$$

In summary, the symbolic representation of a linear hybrid automaton *HP* with discrete state variables S_d and continuously updated state variables S_c consists of (1) an initialization formula *Init* of type `AffForm` over (S_c, S_d), (2) a transition formula *Trans* of type `AffForm` over $(S_c \cup S'_c, S_d \cup S'_d)$, (3) a continuous-time invariant *CI* of type `AffForm` over (S_c, S_d), and (4) a rate constraint *RC* of type `AffForm` over $(\dot{S}_c, S_d)$. Such a description can be compiled from the source language used to describe linear hybrid automata automatically.

Operations on Affine Formulas

The symbolic search techniques discussed in section 3.4 require the following operations on the data type for regions: union `Disj`, intersection `Conj`, set difference `Diff`, emptiness test `IsEmpty`, existential quantification `Exists`, and renaming of variables `Rename`. All these operations can be effectively implemented for the data type `AffForm` of affine formulas as discussed below.

For two affine formulas A and B, the union `Disj`(A, B) is simply the formula $A \vee B$, as it is guaranteed to be of type `AffForm`.

Consider two affine formulas A and B. The formula corresponding to `Conj`(A, B) cannot simply be the conjunction $A \wedge B$ since it need not be an affine formula in the required disjunctive normal form. However, the distributivity properties of the logical disjunction and conjunction operations can be used to implement the desired operation. If the formula A equals $\varphi_1 \vee \cdots \vee \varphi_a$, where each φ_i is a conjunctive affine formula, and the formula B equals $\psi_1 \vee \cdots \vee \psi_b$, where each ψ_j is a conjunctive affine formula, then `Conj`(A, B) is the affine formula

$$\bigvee_{1 \leq i \leq a, 1 \leq j \leq b} (\varphi_i \ \wedge \ \psi_j).$$

Note that each disjunct $(\varphi_i \ \wedge \ \psi_j)$ is a conjunctive affine formula, and the size of the resulting formula grows quadratically as it has $a \cdot b$ number of disjuncts.

The set-difference operation `Diff`(A, B) for two affine formulas can be implemented with some rewriting and is left as an exercise.

The renaming operation can be implemented by simple textual substitution. For an affine formula A, to rename a variable x in A to another name y, which does not occur in A, the result `Rename`(A, x, y) is obtained by replacing every occurrence of the variable x in the formula A by y.

To implement the operation `IsEmpty`, given a formula A of type `AffForm` over (V_c, V_d), we need to check if the variables can be assigned values so that the formula A is satisfied. Suppose the formula A is the disjunction $\varphi_1 \vee \cdots \vee \varphi_a$, where each φ_i is a conjunctive affine formula. Then the formula A is satisfiable exactly when one of the formulas φ_i is satisfiable, and satisfiability of each of these subformulas can be checked independently. Thus, it suffices to focus on testing satisfiability of a conjunctive affine formula. If it contains two conjuncts of the form $(x = d)$ and $(x = d')$, for a discrete variable x and two *distinct* constants d and d', then the formula cannot be satisfied. Otherwise the constraints involving discrete variables do not influence satisfiability. Thus, the core computational problem for checking satisfiability of affine formulas reduces to checking satisfiability of a conjunction of atomic affine constraints, that is, checking satisfiability of a formula of the form $\varphi_1 \wedge \cdots \wedge \varphi_k$, where each φ_i is an affine constraint of the form $(a_1x_1 + a_2x_2 + \cdots + a_nx_n \sim a_0)$. Checking satisfiability of a conjunction of such affine constraints is a classical problem in linear programming with a well-understood theoretical foundation and a variety of efficient implementations.

Quantifier Elimination

Finally, let us focus on the operation of existential quantification: given an affine formula A and a variable x, we want to compute the result $B = \texttt{Exists}(A, x)$ so that the formula B is of type `AffForm`, does not involve the variable x, and a state s satisfies the formula B exactly when there exists a value c for the variable x such that the state $s[x \mapsto c]$ satisfies the formula A. Let us assume that the variable x to be quantified is a real-valued variable, and the formula A is a conjunction $\varphi_1 \wedge \cdots \wedge \varphi_k$, where each φ_i is an affine constraint of the form $(a_1x_1 + a_2x_2 + \cdots + a_nx_n \sim a_0)$. This case captures the computational essence of the problem, and handling the general case is left as an exercise.

Consider a conjunct φ_i of the form $(a_1x_1 + a_2x_2 + \cdots + a_nx_n \sim a_0)$, where the variable x equals x_1. If the coefficient a_1 equals 0 (that is, the variable to be eliminated does not appear in the conjunct φ_i), then φ_i directly appears as a conjunct in the result B. If the coefficient a_1 is non-zero, then consider the expression e_i given by $(a_0 - a_2x_2 - \cdots - a_nx_n)/a_1$. If the comparison operation $\sim$ is $\leq$ and the coefficient a_1 is positive, then we can rewrite the constraint φ_i as $(x \leq e_i)$, and in such a case, the expression e_i is an upper bound on the value of x. If the comparison operation $\sim$ is $\leq$ and the coefficient a_1 is negative, then we can rewrite the constraint φ_i as $(x \geq e_i)$, and in such a case, the expression e_i is a lower bound on the value of x. The other cases are similar but can lead to strict lower bound constraints of the form $(x > e_i)$ and strict upper bound constraints of the form $(x < e_i)$. Now we can eliminate the variable x from the constraints if we simply assert that every lower bound on x must be less than every upper bound on x. For instance, the constraints $(x \geq e_i)$ and $(x \leq e_j)$ lead to the implied constraint $(e_i \leq e_j)$, and the constraints $(x \geq e_i)$ and $(x < e_j)$ lead to the implied constraint $(e_i < e_j)$. If the conjunction of all such implied constraints is satisfiable, then the maximum of the lower bounds on x does not exceed the minimum of the upper bounds on x, and in such a case, it is possible to find a value of x that satisfies all the original constraints in A. Each implied constraint of the form $(e_i \leq e_j)$ or $(e_i < e_j)$ can be easily rewritten so that it is an atomic affine formula and contributes a conjunct to the desired result B.

To illustrate the quantifier elimination procedure, consider the conjunctive affine formula A given by:

$$(\,2x+3y-5z\ < 7\,) \wedge (\,6y+8z\ \geq -2\,) \wedge (\,-x+y-7z\ \leq 10\,) \wedge (\,3x+z\ \leq 0\,).$$

The first conjunct gives the strict upper bound constraint $x < (7-3y+5z)/2$, the second conjunct does not constrain x, the third conjunct gives the lower bound constraint $x \geq (-10+y-7z)$, and the fourth conjunct gives the upper bound constraint $x \leq -z/3$. We eliminate x by requiring every lower bound not to exceed every upper bound and retaining the second conjunct. This leads to:

$$(6y{+}8z \geq -2) \wedge [(-10{+}y{-}7z) \leq -z/3] \wedge [(-10{+}y{-}7z) < (7{-}3y{+}5z)/2].$$

We then rewrite the last two conjuncts so that the formula is in the desired affine form:

$$(6y+8z \geq -2) \wedge (3y-20z \leq 30) \wedge (5y-19z < 27).$$

Note that the number of atomic constraints in the result B can be quadratic in the number of atomic constraints in the input formula A. If we apply the existential quantification repeatedly, the number of constraints grows exponentially.

Image Computation: Discrete Transitions

The core of the symbolic search is *image computation*: given a region A over the state variables, we want to compute the region that contains all the states that can be reached from the states in A using one transition. If we focus on the discrete transitions, then the algorithm for image computation is identical to the one discussed in section 3.4. Given an affine formula A, we first conjoin it with *Trans*, a region over unprimed and primed state variables containing all the discrete transitions. The intersection $\texttt{Conj}(A, \mathit{Trans})$ is a region over $S \cup S'$ and contains all the discrete transitions that originate in the states in A. Then we project the result onto the set S' of primed state variables by existentially quantifying the variables in S. Renaming each primed variable x' to x gives us the desired region. Thus, the discrete post-image of the region A of a linear hybrid automaton *HP* with the transition formula *Trans* is defined by

$$\texttt{DiscPost}(A, \mathit{Trans}) = \texttt{Rename}(\texttt{Exists}(\texttt{Conj}(A, \mathit{Trans}), S), S', S).$$

For the pursuit-game example, the discrete post-image of the initial region can be obtained using the above formula. The result of this computation is the affine formula A_1:

$$\texttt{DiscPost}(\mathit{Init}, \mathit{Trans}) = A_1 = (\mathit{mode} = \texttt{ClkW}) \wedge (x = 0) \wedge (e = 20) \wedge (p = 10).$$

Image Computation: Timed Transitions

Our next goal is to compute a symbolic representation of the set of all states resulting from a timed action starting from a state in a given region of the hybrid automaton *HP*. Let A be a region of the hybrid automaton, let *CI* be its continuous-time invariant, and let RC be its rate constraint. The timed post-image B of the region A consists of all states s', such that there exists a state s belonging to the region A and a time duration δ and a rate vector r such that (1) the state s together with the rates r satisfy the formula RC; (2) for every $0 \leq t \leq \delta$, the state $s + r\,t$ satisfies the invariant CI; and (3) the final state s' equals $s + r\,\delta$. For simplicity, let us assume that the continuous-time invariant CI defines a *convex* set: if two states satisfy the formula CI, then so does every state that lies on the segment joining the two states. This is a typical case and holds for the pursuit-game example. In such a case, the condition (2) can be replaced by the simpler condition that the states s and s' at the beginning and the end of the timed action both satisfy CI.

The formula A uses the discrete variables S_d and continuously updated variables S_c. To get the desired result, we use auxiliary variables S'_c that denote the values of the continuously updated variables at the end of the timed action and the variable *inc* that denotes the duration of the timed action (assume that *inc* is not a variable of the hybrid automaton). We will first construct a formula B' over the variables S_d, S_c, S'_c, and *inc*. This formula captures all the timed transitions starting in the region A.

Since the region B' should capture all timed actions starting in the region A, A is a conjunct in B'. To ensure that the continuous-time invariant holds at the beginning of the timed action, the formula CI is also a conjunct of B'. To ensure that the continuous-time invariant holds at the end of the timed action, the formula $\texttt{Rename}(CI, S_c, S'_c)$ obtained by renaming every continuously updated variable x to x' is also a conjunct of B'.

We now need a constraint that says the rates for variables are chosen according to the rate constraint, and the increment $(x' - x)$ in the value of a continuously updated variable x equals the value of the duration variable *inc* multiplied by the rate of change of x. A direct encoding of this constraint uses the multiplication operation and thus results in a non-linear constraint. We avoid this by exploiting the special structure of the rate formula RC: an atomic affine constraint of the form $(a_1\dot{x}_1 + \cdots + a_n\dot{x}_n \sim a_0)$ is replaced by the constraint $[a_1(x'_1 - x_1) + \cdots + a_n(x'_n - x_n) \sim a_0\, inc]$. Observe that this modified constraint is equivalent to the original constraint since the constant rate of change $\dot{x}_i$ of a variable equals the change in its value divided by the duration, which is captured by the expression $(x'_i - x_i)/inc$. More precisely, given the affine formula RC over $(\dot{S}_c, S_d)$, let RC' be the affine formula over $(S'_c \cup \{inc\}, S_d)$ obtained by replacing an atomic affine constraint of the form

$$(a_1\dot{x}_1 \;+\; \cdots \;+\; a_n\dot{x}_n) \;\sim\; a_0$$

by the atomic affine constraint

$$(a_1x'_1 - a_1x_1 \;+\; \cdots \;+\; a_nx'_n - a_nx_n \;-\; a_0\, inc) \;\sim\; 0.$$

The formula B' is the conjunction of the affine formulas A, CI, $\texttt{Rename}(CI, S_c, S'_c)$, and RC'.

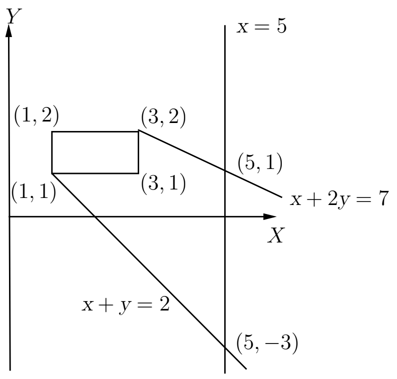


Figure 9.28: Example of Timed Post-image Computation

To obtain the desired timed post-image B from the formula B', we can use quantifier elimination to project out the starting values of the continuously updated variables along with the time duration and rename every primed variable x' back to x. Thus, the timed post-image $\texttt{TimedPost}(A, CI, RC)$ of the region A is defined by:

$$\texttt{Rename}(\texttt{Exists}(\texttt{Conj}(A, CI, \texttt{Rename}(CI, S_c, S'_c), RC'), S_c \cup \{ inc \}), S'_c, S_c).$$

To illustrate the computation of the timed post-image, suppose there are two continuously updated variables x and y and no discrete variables. Suppose the region A is given by $(1 \le x \le 3) \ \wedge \ (1 \le y \le 2)$ (see the rectangle in figure 9.28). Suppose the continuous-time invariant is $(x \le 5)$, and the rate constraint is $(\dot{x} = 1) \ \wedge \ (-1 \le \dot{y} \le -0.5)$. This means that during a timed action, a state evolves along a line within the cone bounded by lines with slopes -1 and -0.5 as long as the value of x does not exceed 5. To compute the timed post-image, we first introduce the primed variables x' and y' and the duration variable inc. The transformed rate constraint RC' is

$$[(x' - x) = inc] \ \wedge \ [-inc \le (y' - y) \le -0.5\, inc].$$

The formula B' is the conjunction

$$\begin{array}{l}(1 \le x \le 3) \ \wedge \ (1 \le y \le 2) \ \wedge \ (x \le 5) \ \wedge \ (x' \le 5) \ \wedge \\ (x' - x = inc) \ \wedge \ [-inc \le (y' - y) \le -0.5\, inc].\end{array}$$

The desired region B is obtained by eliminating the variables x, y, and inc from the above formula and renaming x' to x and y' to y in the result. The final result is equivalent to the affine formula

$$(x \ge 1) \ \wedge \ (y \le 2) \ \wedge \ (x \le 5) \ \wedge \ (x + y \ge 2) \ \wedge \ (x + 2y \le 7).$$

In figure 9.28, the timed post-image of the rectangle A is the pentagon with the vertices $(1,1)$, $(1,2)$, $(3,2)$, $(5,1)$, and $(5,-3)$.

For the pursuit-game example, no timed action of a positive duration is possible from the initial state (since the clock x is 2), and the timed post-image of the region *Init* is *Init* itself. The timed post-image of the region A_1, which was obtained by applying the discrete post-image operation to *Init*, is the affine formula

$$(\, mode = \mathtt{ClkW}\,) \wedge (\, 0 \leq\, x \,\leq 2\,) \wedge (\, e - 5x = 20\,) \wedge (\, 10 - 0.5x \,\leq\, p \,\leq\, 10 + 6x\,).$$

Iterative Image Computation

The post-image of a region A of a linear hybrid automaton is simply the union (or disjunction) of its discrete post-image and timed post-image:

$$\mathtt{Post}(A, \mathit{Trans}, \mathit{CI}, \mathit{RC}) = \mathtt{Disj}(\mathtt{DiscretePost}(A, \mathit{Trans}), \mathtt{TimedPost}(A, \mathit{CI}, \mathit{RC})).$$

To check whether a property φ over the state variables S is an invariant (or, equivalently, whether the negated property $\neg\varphi$ is reachable), we can now apply the symbolic breadth-first search algorithm of figure 3.18 based on the representation of regions as affine formulas.

The complexity of the representation, that is, the size of the affine formula representing the current set of reachable states, grows significantly with the number of iterations. Tools for symbolic reachability analysis of linear hybrid automata such as HyTech and SpaceEx incorporate a number of optimizations to improve the computational efficiency. When a linear hybrid automaton has n continuously updated variables, each atomic affine constraint is a hyperplane in the n-dimensional space of real-valued vectors. A conjunctive affine formula then is a polyhedron in the n-dimensional space. The optimizations for manipulating affine formulas are based on employing alternative representations for polyhedra and algorithms for simplifying such representations. For example, given the polyhedron as a conjunction of atomic affine constraints, one of the constraints can be omitted if it does not represent a bounding facet of the polyhedron.

Even when the linear hybrid automaton has only finitely many discrete states, the symbolic reachability algorithm may not terminate. As an example, consider the linear hybrid automaton of figure 9.29 with a single mode and two continuously updated variables. Both variables are initialized to 0. The variable x evolves at the rate 1, whereas the variable y changes at the rate 2. Initially, both variables are 0. The set of reachable states discovered in the first iteration of the symbolic search is the line segment joining $(0,0)$ and $(1/2,1)$. When y reaches 1, the discrete mode-switch updates the state to $(0,1/2)$. In the next iteration, when y reaches 1, the value of x is $1/4$, and the discrete mode-switch updates the state to $(0,1/4)$. This pattern repeats forever (see figure 9.29). The set of reachable states of this system contains infinitely many disconnected line segments: the kth line segment connects the states $(0, 1-1/2^k)$ and $(1/2^{k+1}, 1)$.

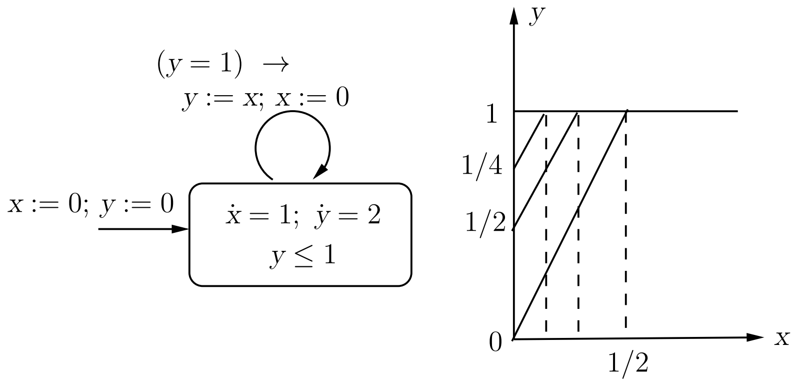


Figure 9.29: Example of Non-terminating Symbolic Image Computation

It is clear that the symbolic breadth-first search algorithm keeps iterating forever discovering shorter and shorter such segments. In contrast, as studied in section 7.3, for timed automata, all continuously updated variables increase at the same rate, and if the timed automaton has finitely many discrete states, the symbolic reachability algorithm based on the symbolic representation of clock-zones (or clock-regions) is guaranteed to terminate.

Analysis of the Pursuit Game

The symbolic reachability analysis can be used to determine whether the evader's strategy of figure 9.26 is a winning strategy for a given initial position. For this purpose, the symbolic breadth-first search algorithm of figure 3.18 is executed to check whether the property $(e - p = 0)$ is reachable: if the property is discovered to be reachable, then the pursuer wins the game, or else the evader wins the game.

For the initial position $e_0 = 20$ and $p_0 = 1$, after four iterations of the algorithm, the property $(e - p = 0)$ is found to be reachable. Thus, the pursuer wins in this case.

For the initial position $e_0 = 20$ and $p_0 = 10$, after five iterations, the algorithm terminates having computed all the reachable states, without finding any state where the property $(e - p = 0)$ holds. The set of reachable states is given by the formula:

$$\begin{array}{lllll} & mode = \texttt{ClkW} & \wedge\ e = 20 & \wedge\ p = 10 & \wedge\ x = 2 \\ \vee & mode = \texttt{ClkW} & \wedge\ e = 20 + 5x & \wedge\ 10 - 0.5x \le p \le 10 + 6x & \wedge\ 0 \le x \le 2 \\ \vee & mode = \texttt{ClkW} & \wedge\ e = 30 + 5x & \wedge\ 9 - 0.5x \le p \le 22 + 6x & \wedge\ 0 \le x \le 2 \\ \vee & mode = \texttt{Rescued} & \wedge\ e = 0 & \wedge\ 8 \le p \le 34 & \wedge\ x \ge 2\,. \end{array}$$

Now suppose, having fixed the initial position e_0 of the evader to be 20, we want to compute the set of initial positions p_0 of the pursuer for which the evader wins. This can be achieved using the same symbolic reachability algorithm but treating the initial position p_0 as another symbolic variable. Formally, we modify the linear hybrid automaton of figure 9.26 by adding a fourth continuously updated variable called p_0. The value of this variable does not change: in every mode, modify the rate constraint by adding the conjunct $(\dot{p_0} = 0)$. The initial condition is now the formula

$$(\,mode = \mathtt{ClkW}\,)\ \wedge\ (\,x = 2\,)\ \wedge\ (\,e = 20\,)\ \wedge\ (\,p - p_0 = 0\,).$$

Then we apply the iterative image computation algorithm until all the reachable states are computed. If the affine formula *Reach* over the variables *mode*, x, e, p, and p_0 represents all the reachable states, then the formula

$$\mathtt{Exists}(\mathtt{Conj}(\mathtt{Reach}, (\,e - p = 0\,)), \{mode, p, x, e\})$$

is an affine formula containing only the variable p_0 that gives the constraint on the initial position of the pursuer for which the pursuer wins. Based on the computation using the tool HyTech, this formula turns out to be $(0 \leq p_0 \leq 2)\ \wedge\ (16 \leq p_0 \leq 40)$. This tells us that when the evader's initial position is 20, the evader wins exactly when the pursuer's initial position belongs to the interval $(2, 16)$.

Exercise 9.13 : Given two formulas A and B of type `AffForm`, show how to implement the operation $\mathtt{Diff}(A, B)$, that is, how to compute a formula that is (1) equivalent to the logical formula $A \wedge \neg B$, and (2) of type `AffForm`. ■

Exercise 9.14 : Describe the procedure for implementing the existential quantification for the data type of affine formulas in its full generality: given an affine formula A over (V_c, V_d) and a variable $x \in V_c \cup V_d$, show how to obtain the affine formula for $\mathtt{Exists}(A, x)$. ■

Exercise 9.15 : Consider the region A given by the conjunctive affine formula

$$(\,-x_1 + 6x_4\ \leq\ 17\,)\ \wedge\ (\,3x_1 + 12x_3\ <\ 1\,)\ \wedge\ (\,2x_1 - 3x_2 + 5x_4\ \leq\ 7\,)\ \wedge$$
$$(\,7x_2 - x_3 - 8x_4\ >\ 0\,)\ \wedge\ (\,5x_1 + 2x_2 - x_3\ >\ -5\,).$$

Calculate the region $\mathtt{Exists}(A, x_1)$ as an affine formula. ■

Exercise 9.16 : Suppose a linear hybrid automaton has two continuously updated variables x and y and no discrete variables. Consider the triangular region A with vertices $(0,0)$, $(1,2)$, and $(2,1)$. Suppose the continuous-time invariant is $(x - y \leq 4)$ and the rate constraint is $(\dot{x} = 1)\ \wedge\ (0.5 \leq \dot{y} \leq 1)$. Calculate the timed post-image of the region A and describe it as an affine formula. ■

Bibliographic Notes

The study of formal models for hybrid systems by combining discrete transition systems and differential/algebraic equations started in the early 1990s [MMP91]. Our model of hybrid processes is based on hybrid automata [ACH$^+$95]. As examples of well-developed formal approaches to specification and verification of hybrid systems, see [Tab09] and [Pla10]. Modeling of embedded control systems using hybrid automata is now also supported by commercial modeling software; for example, Mathworks (see `mathworks.com`) supports modeling using the combination of STATEFLOW and SIMULINK.

Mathematical analysis of stability and control design for hybrid dynamical systems is an active topic of research in control theory (see [Bra95] and [LA14]).

The case study of the automated guided vehicle is based on [LS11], the modeling of obstacle avoidance appears in [AEK$^+$99], the analysis of multi-hop time-triggered control network is based on [ADJ$^+$11], and the pursuit game of figure 9.25 is from [AHW97].

The model of linear hybrid automata was introduced in [ACH$^+$95], and the corresponding symbolic analysis algorithm was first implemented in the model checker HYTECH [AHH96, HHW97] (see [FLGD$^+$11] for recent and efficient techniques for analysis of linear hybrid automata).