A Hierarchical Hybrid System Model and Its Simulation

Jie Liu, Xiaojun Liu, Tak-Kuen J. Koo, Bruno Sinopoli, Shankar Sastry, and Edward A. Lee
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720, USA
{liuj, liuxj, koo, sinopoli, sastry,eal}@eecs.berkeley.edu

Abstract

This paper presents a hierarchical hybrid system modeling and simulation framework using the Ptolemy II environment. Ptolemy II is a system-level design tool that supports the integration of multiple models of computation. The modeling of hierarchical hybrid systems is achieved by combining continuous-time models with finite state automata. Breakpoint handling, event detection and invariant monitoring techniques are studied. A hybrid helicopter control system is simulated as an example.

1. Introduction

Hybrid systems have been intensively studied in the past few years both for their rigorous mathematical foundations [2], [7], [12] and for engineering designs [3], [8]. There is a strong demand for computer aided design and simulation tools that can help validate hybrid systems. A few hybrid system simulation tools are available [1], [4], [17], [18], but most of them only support a subset of the requirements for hybrid simulation [14], [10]. Our approach is based on a hierarchical assembly of heterogeneous components, which can efficiently capture the hybrid automata model of hybrid systems, and the approach extensively supports the interaction of continuous and discrete dynamics.

We start with the hybrid I/O automaton [13] view of hybrid systems, by defining a "*open*" *hybrid automaton* H as

$$H = (\boldsymbol{Q}, \boldsymbol{X}, \boldsymbol{U}, \boldsymbol{Y}, Init, f, h, Inv, E, G, R), \qquad (1)$$

where

- $\boldsymbol{Q}$ is a set of discrete variables;
- $\boldsymbol{X}$ is a set of continuous variables;
- $\boldsymbol{U}$ is a set of input variables, continuous or discrete;
- $\boldsymbol{Y}$ is a set of output variables, continuous or discrete;
- $Init \subseteq \boldsymbol{Q} \times \boldsymbol{X}$ is a set of initial states;
- $f: \boldsymbol{Q} \times \boldsymbol{X} \times \boldsymbol{U} \to T\boldsymbol{X}$ is a vector field;
- $h: \boldsymbol{Q} \times \boldsymbol{X} \times \boldsymbol{U} \to \boldsymbol{Y}$ is an output map (note that we do not restrict h to be only a function of $\boldsymbol{Q}$ and $\boldsymbol{X}$);
- $Inv: \boldsymbol{Q} \to \wp(\boldsymbol{X} \times \boldsymbol{U})$ assigns to each discrete state $q \in \boldsymbol{Q}$ an invariant set;
- $E \subset \boldsymbol{Q} \times \boldsymbol{Q}$ is a collection of discrete transitions;
- $G: E \to \wp\boldsymbol{X}$ assigns to each $e = (q, q') \in E$ a guard;
- $R: E \times \boldsymbol{X} \times \boldsymbol{U} \to \wp\boldsymbol{X}$ assigns to each $e \in E$, $x \in \boldsymbol{X}$, and $u \in \boldsymbol{U}$ a reset relation.

In the definition, $\wp\boldsymbol{X}$ is the power set of $\boldsymbol{X}$, and $T\boldsymbol{X}$ is the tangential space of $\boldsymbol{X}$.

We refer to $(q, x) \in \boldsymbol{Q} \times \boldsymbol{X}$ as the (hybrid) state of H, $u \in \boldsymbol{U}$ as the input of H, and $y \in \boldsymbol{Y}$ as the output of H. For the sake of simplicity, we sometimes use $f_q(x, u)$ to denote $f(q, x, u)$. And we assume $f_q(x, u)$ is globally Lipschitz continuous in its arguments.

The composition of two hybrid I/O automata is, roughly, to connect some inputs/outputs of one hybrid automaton with some outputs/inputs of another. The remaining inputs and outputs are the inputs and outputs of the composed automaton. For a formal definition of the composability and composition of hybrid I/O automata, please refer to [13].

A *hybrid time trajectory* τ is defined [12] to be a finite or infinite sequence of intervals of real, $\tau = \{I_i\}$, where i is a natural number. It satisfies the following conditions:

- I_i is closed for each i, unless τ is a finite sequence and I_i is the last interval, in which case it is left closed but can be right open.
- Let $I_i = [\tau_i, \tau_i']$. Then for all i, $\tau_i \le \tau_i'$, and for all $i > 0$, $\tau_i = \tau'_{i-1}$.

We denote by Υ the set of all hybrid time trajectories.

An *execution* χ of a hybrid automaton H is defined [12] as a collection $\chi = (\tau, q, x, u, y)$ with $\tau \in \Upsilon$, $q: \tau \to \boldsymbol{Q}$, $x: \tau \to \boldsymbol{X}$, $u: \tau \to \boldsymbol{U}$, and $y: \tau \to \boldsymbol{Y}$ satisfying:

- (*initial condition*) $(q(\tau_0), x(\tau_0)) \in Init$;
- (*continuous evolution*) $\forall i$, with $\tau_i < \tau_i'$, x, u, y, and q are continuous over $[\tau_i, \tau_i']$, and $\forall t \in [\tau_i, \tau_i')$, $(x(t), u(t)) \in Inv(q(t))$, and $\dot{x}(t) = f(q(t), x(t), u(t))$;
- (*discrete evolution*) $\forall i$, $e_i = (q(\tau_i'), q(\tau_{i+1})) \in E$, $(x(\tau_i'), u(\tau_i')) \in G(e_i)$, and $x(\tau_{i+1}) \in R(e_i, x(\tau_i'), u(\tau_i'))$; and
- (*output evaluation*) $\forall t \in \tau, y(t) = h(q(t), x(t), u(t))$.

Intuitively, an execution of a hybrid automaton starts from an initial state, runs the continuous dynamic for a while, makes a discrete state transition, and then runs (another) continuous dynamic for another period of time, and so on. To simulate hybrid systems, we want to compute the execution χ (or its approximation within an error tolerance).

In this paper, we introduce Ptolemy II [5], a system-level design environment, and show how hybrid system simula-

tion can be performed in it. In the environment, hybrid I/O automata are modeled using hierarchical composition of finite automata (FA) and continuous-time (CT) systems. The arbitrarily deep hierarchical nesting of continuous systems and discrete automata makes the model handled by Ptolemy II somewhat more general than that given in (1). The continuous simulation techniques implemented in Ptolemy II has the idea of breakpoint handling integrated. The breakpoint handling mechanism significantly empowers the continuous simulation such that requirements of hybrid simulation, like integration step control, event detection, and invariant monitoring are easily supported.

We give the hierarchical model of hybrid systems in section 2. The continuous-time simulation techniques are discussed in section 3. Invariant monitoring and event generation mechanisms are presented in section 4. The simulation control of continuous dynamics and discrete state transitions is described in section 5. As a case study, a 2-D helicopter with a hybrid controller is simulated.

2. Hierarchical Hybrid Automata Modeling

Ptolemy II models the hierarchical organization of a system using the container-containee relationship. The top-level system model consists of a set of executable entities called *actors*. Each actor models a sub-system. Actors can be atomic or composite, where a composite actor can in turn contain a set of actors. This hierarchical nesting can be extended to arbitrary levels. An actor communicates with the rest of the system through a set of input and output ports. The messages passing among the ports are encapsulated in *tokens*. This modeling mechanism is close to the intuitive representations of systems and maximizes the information hiding of components.

2.1. Modeling Automata

A finite state automaton in Ptolemy II is specified using the usual bubble-and-arc graph as illustrated in Figure 1. The guard condition and reset relation on a transition can refer to the inputs to the hybrid automaton and the outputs from the continuous dynamics of the source state of the transition. Ptolemy II provides a powerful and extensible expression language for specifying guards and reset relations.

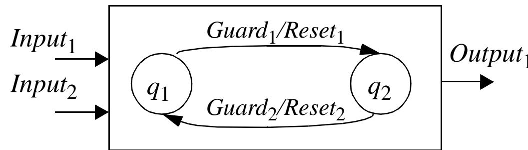

Figure 1. A finite state I/O automaton.

2.2. Modeling Continuous Dynamics

In each discrete state q of a hybrid automaton, there is an “open” continuous subsystem with the form of a set of ordinary differential equations (ODEs):

$$\begin{aligned} \dot{x} &= f_q(x, u_q, t) \\ y_q &= g_q(x, u_q, t). \end{aligned} \quad (2)$$

In Ptolemy II, we use a signal-flow model to represent a continuous time (sub)system [9], which means that each component in the system is a function that maps input signals to output signals, and the components communicate via continuous-time signals. For example, the system in (2) is built by integrators with feedback, as shown in Figure 2. The states of the system are the outputs of the integrators.

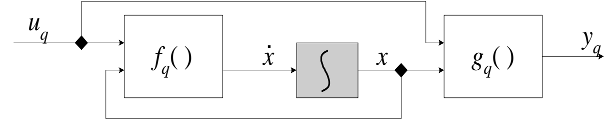

Figure 2. Signal flow model of a continuous-time system.

2.3. Hierarchical Hybrid Automata

In our approach, a hybrid automaton model is a composite actor containing a finite state automaton actor modeling the discrete dynamics, and a set of composite actors modeling the continuous dynamics of the states. The top level is a continuous-time system, where one or more components are composite actors. These composite actors implement finite state automata internally. Each state of the automaton is further refined by either another layer of automaton or a continuous-time subsystem. This hierarchy can be further nested until all subsystems are refined by a continuous-time subsystem, as illustrated in Figure 3.

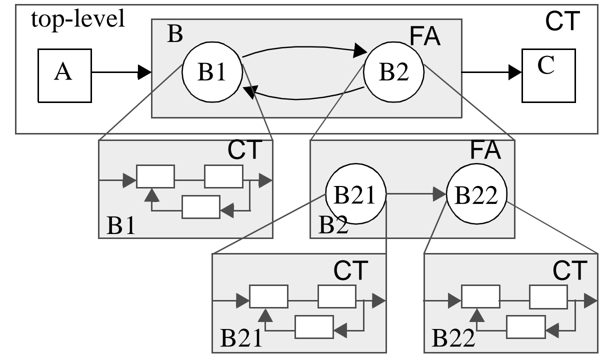

Figure 3. A hierarchical hybrid automaton.

3. Simulating Continuous Dynamics

In a hybrid system, when there is no discrete state transition, the entire system can be flattened into a CT system, captured by a set of ODEs. The task of a simulator is to solve the set of ODEs numerically, that is, discretizing time into discrete points, and finding the behavior of the system (values of all state variables) at those points. How time is discretized depends largely on the speed and accuracy requirements of the simulation, and, in hybrid systems, on the occurrence of state transitions as well.

In the signal-flow representation, numerical ODE solving methods can be performed by executing certain (chain of)

actors in a particular order. For example, at a discrete state q the forward Euler method has the form

$$x_{t+h} = x_t + h \cdot f_q(x_t, u_t, t), \quad (3)$$

where x_t is the (computed) state at time t (distinguishing itself from the "real" state $x(t)$) and h is the integration step size. So at time t, the integrator emits its state x_t. This token, together with the input token u_t, are consumed by the actor $f(\ )$, and it produces the derivative of x_t, i.e. $f_q(x_t, u_t, t)$. The derivative is consumed by the integrator, and by using (3), the new state x_{t+h} is obtained. Other integration methods, like linear multistep (LMS) methods and Runge-Kutta (RK) methods, are similarly accomplished.

3.1. Integration Step Size Control

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

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

$$E_{t_n} = \| x_{t_n} - x(t_n) \|. \quad (4)$$

  It can be shown [6] that by carefully choosing the parameters in the integration methods, the local error can be made such that $E_{t_n} \sim O((t_n - t_{n-1})^r)$, where r, an integer closely related to the number of function evaluation in one integration step, is called the *order* of the integration method. Therefore, in order to achieve an accurate solution, we want high order integration methods and small step sizes. But, a high order method means more function evaluations per step, and small step sizes lead to more integration steps, both resulting in a longer simulation time. In general, the choice of integration methods and step sizes reflects the trade-off between speed and accuracy of a simulation.
- *Convergence*. Implicit ODE solving methods convert the ODEs to a set of algebraic equations and solve them using Newton-Raphson method or contraction mapping iterations [15]. Both methods are fixed-point iteration based, and the convergence requires the step size to be small and the initial guess to be relatively accurate.
- *Breakpoints*. Breakpoints are the time points where the vector field f is not continuous. This may be the result of f itself, the discontinuity of the input signals u, or discrete state transitions. In general, the solutions at these points are not well defined. But the left and right limits are. So instead of solving the ODE at those points, we would actually try to find the left and right limit. In addition, the numerical integration formula is not applicable when the integration step crosses breakpoints. After each breakpoint, the integration process should be reinitialized as if it is the beginning of the simulation. How the breakpoints are detected and handled is key for hybrid system simulation.

3.2. Breakpoints Handling

We classify two kinds of breakpoints, predictable ones and unpredictable ones. *Predictable breakpoints* are the breakpoints that are known (exactly) beforehand. For example, a square wave source actor can predict its next flip time. This information is used to control the discretization of time. Predictable breakpoints are stored chronologically in a *breakpoint table*. Before each integration step, say from t to $t+h$ (h is determined by error control and convergence concerns), the breakpoint table will be examined. If there is a breakpoint at $t+\delta$, where $\delta < h$, then the step size will be reduced to δ. After the integration is finished, we have obtained the left limit of x at $t+\delta$.

An *unpredictable breakpoint* is unknown until the time it occurs. For example, an actor that varies its functionality when the input signal crosses a threshold can only report a "missed" breakpoint after an integration step has finished. Unpredictable breakpoints are handled by querying actors after each integration step. If all the actors report that this step is acceptable, then the integration continues; otherwise, the actors are asked for a refined step size, which is the step size that the actor estimates to locate the breakpoint. This process is iterated until the breakpoint is found "accurately" within an error tolerance.

4. Invariant Monitors and Event Generation

A hybrid automaton will take a transition from one discrete state p to another discrete state q at time τ' if one of the followings is true:

1.) τ' is the first time in some I_i such that the invariant at state p is violated. I.e. $\exists \varepsilon > 0$, such that $\forall 0 < \delta \le \varepsilon$, $(x(\tau' - \delta), u(\tau' - \delta)) \in Inv(p(\tau' - \delta))$, but

$$(x(\tau'), u(\tau')) \notin Inv(p(\tau')); \quad (5)$$

2.) one of the guards from p is enabled, i.e.

$$\exists e = (p(\tau'), q(\tau')) \in E, \text{ s.t. } (x(\tau'), u(\tau')) \in G(e). \quad (6)$$

Note that when condition (5) is true, the transition is forced to be taken, and if (6) is true, the transition is optional. In the latter case, the system has a nondeterministic behavior.

Conditions (5) and (6) are detected in a continuous subsystem, and it relies on the automaton super-system to decide whether to take the transition. The test of the conditions is achieved by invariant monitors. *Invariant monitors* are actors that consume continuous waveforms and produce events that can trigger state transitions. Consider the invariant condition with the form

$$\varphi(x(t), u(t), t) > 0 \quad (7)$$

which has the "boundary" condition:

$$\varphi(x(t), u(t), t) = 0. \quad (8)$$

(More complex conditions can be modeled by the logical combination of inequalities like (7).) We need to find the exact time point where (8) is true, given that the previous trajectory satisfies (7). This is handled by the breakpoint mechanism.

4.1. Predictable Event Generation

If φ in (8) is only a function of time, (the event of this type is called a *time event*) then we know the exact state transition time before the simulation actually reaches that time. In this case, the state transition time is simply registered as a predictable breakpoint. Since the breakpoint table mechanism can guarantee that the simulation will not miss any predictable breakpoint during the execution, the invariant monitor can emit the triggered event at the desired time.

4.2. Unpredictable Event Detection

If φ in (8) is also a function of $x(t)$ and $u(t)$, (the event of this type is called a *state event*) then it is, in general, impossible to know the exact time that (8) is true beforehand. This is the situation of an unpredictable breakpoint. The approach we take here is to make the invariant monitor report a missed event to the simulator if one integration step has crossed the invariant set boundary. After iteratively refining the step sizes by numerical root finding techniques, the accurate event time is found.

4.3. Nondeterministic Event Monitoring

A hybrid automaton may take a discrete state transition whenever the guard expression evaluates to true. Since all the outputs of a continuous subsystem are in the scope of the expressions language of the guards, the automaton system can evaluate the expression after each integration step. In this case, it can take the state transition whenever a guard is evaluated to true, but we do not specifically aim to find the first time point that makes the guard true.

One situation that must be taken care of occurs when the guard has the form

$$|\varphi(x(t), u(t), t)| \leq \varepsilon . \quad (9)$$

Denote by $\psi(t)$ the value of $\varphi(x(t), u(t), t)$ at time t. We want to make sure that $\psi(t)$ does not leap over this region in one integration step. This can be done by a threshold monitor that calculates $\psi(t)$ after each integration step. If after one integration step, it changes from $\psi(t) < -\varepsilon$ to $\psi(t+h) > \varepsilon$, or from $\psi(t) > \varepsilon$ to $\psi(t+h) < -\varepsilon$, then it should report a missed event and try to refine the step size such that the end of the integration step falls into the region.

Notice that the state trajectory and invariant/guard conditions could be complicated functions of time. In general, the numerical methods do not guarantee that they will find all the possible discrete events. This is related to how the models are built to support a good simulation.

5. Hybrid Execution Control

As discussed in the previous two sections, CT simulation in Ptolemy II is capable of generating events and monitoring invariants of continuous dynamics. This capability enables the interaction of discrete and continuous dynamics and makes the correct simulation of hybrid systems possible.

When simulating a hybrid system in Ptolemy II, the interaction of discrete and continuous dynamics goes through the following steps:

1.) During continuous evolution, the system is simulated as a CT system where the hybrid automaton is replaced by the continuous dynamics of its current state. The discretization of time during the simulation is controlled such that the time when the invariants is violated is located, and interval conditions of the form (9) on the guards are not missed.
2.) At each discrete time point where the behavior of the system is found, the guards on the transitions starting from the current state are evaluated.
3.) If a transition is enabled, the hybrid automaton makes a state transition. The continuous dynamics of the destination state is initialized by the reset relation on the transition. The simulation continues from 1) with the current time point treated as a breakpoint.
4.) If the invariant of the current state is violated or is going to be violated at the current time, and no guard on an outgoing transition is enabled, the simulation is blocked.

For a deterministic hybrid automaton, the above simulation scheme will calculate the execution for each initial condition within the precision of continuous-time simulation. For non-deterministic automata, the execution also depends on many other factors, which include the user's choice of integration methods, the maximum and minimum step sizes, and the error tolerance.

6. Case Study: A Helicopter Control System

In this section, we present a hybrid system which models a high-attitude take-off process of a 2-D helicopter. The hybrid control sequence is motivated by helicopter pilot flight instructions [16].

6.1. A 2-D Helicopter Model

A 2-D model of a helicopter is extracted from [8]. The motion along longitudinal and vertical axes is considered. The x, z-axes of the spatial frame are pointing north and down. The body x-axis is defined from the center of gravity to the nose of the helicopter, and body z-axis is pointing down from the center of gravity (CG). The motion of the helicopter is controlled by T_M, the main rotor thrust, and ε, the longitudinal tilt path angle. The state variables are p_x, p_z, and θ, which are the position on the x-axis, z-axis, and the pitch angle, respectively. The equations of the motion can be expressed as:

$$\begin{bmatrix}\ddot{p}_x\\ \ddot{p}_z\end{bmatrix} = -\frac{1}{m}\begin{bmatrix}\cos\theta & -\sin\theta\\ \sin\theta & \cos\theta\end{bmatrix}\begin{bmatrix}T_M\sin\varepsilon\\ T_M\cos\varepsilon\end{bmatrix} + \begin{bmatrix}0\\ g\end{bmatrix} \qquad (10)$$

$$\ddot{\theta} = \frac{1}{I_y}(M_M\varepsilon + h_M T_M \sin\varepsilon) \qquad (11)$$

where I_y is the moment of inertia about body y-axis; M_M is the hub pitching moment stiffness; and h_M is the vertical distance between the main rotor and the CG. The state and input vectors are defined as $x = [p_x, \dot{p}_x, p_z, \dot{p}_z, \theta, \dot{\theta}]^T$ and $u = [T_M, \varepsilon]^T$, respectively.

6.2. Flight Modes

Flight modes [9] represent different modes of operation of the helicopter and they correspond to controlling different variables in the dynamic. We define the following flight modes: Hover, Cruise, Acc/ALH, Dec/ALH, Climb, and Descend, where ALH stands for "ALtitude Hold."

6.3. Flight Mode Controller

The flight mode controller is based on approximate feedback linearization. We assume that full states are accessible for control purpose. The controller has the following form:

$$\begin{bmatrix}T_M^{(3)}\\ \dot{\varepsilon}\end{bmatrix} = A^{-1}(x)\left[-b(x) + \begin{bmatrix}v_x\\ v_z\end{bmatrix}\right]. \qquad (12)$$

Hence, the resulting closed-loop system becomes

$$\begin{bmatrix}p_x^{(5)}\\ p_z^{(5)}\end{bmatrix} = \begin{bmatrix}v_x\\ v_z\end{bmatrix}. \qquad (13)$$

In each flight mode, there is a set of control outputs defined and a corresponding regulator is designed. Given a setpoint for each output, we have the corresponding controllers:

$$\begin{aligned} v_i^p &= -\alpha_0(p_i - c_i^p) - \alpha_1\dot{p}_i - \ldots - \alpha_4 p_i^{(4)} \qquad (14)\\ v_i^v &= -\alpha_1(\dot{p}_i - c_i^v) - \alpha_2\ddot{p}_i - \ldots - \alpha_4 p_i^{(4)}\\ v_i^a &= -\alpha_2(\ddot{p}_i - c_i^a) - \alpha_3 p_i^{(3)} - \alpha_4 p_i^{(4)} \end{aligned}$$

for $i = x, z$. Thus the feedback linearization controller has the form, $u_{fb}(x, v_x^j, v_z^j)$, where $j = p, v, a$. The inputs and outputs of the controllers in each mode are summarized in the following table.

Mode	q	Outputs	Inputs
Hover	q_1	$y_1 = [p_x, p_z]^T$	$u_1 = u_{fb}(x, v_x^p, v_z^p)$
Cruise	q_2	$y_2 = [\dot{p}_x, p_z]^T$	$u_2 = u_{fb}(x, v_x^v, v_z^p)$
Acc/ALH	q_3	$y_3 = [\ddot{p}_x, p_z]^T$	$u_3 = u_{fb}(x, v_x^a, v_z^p)$
Dec/ALH	q_4	$y_4 = [\ddot{p}_x, p_z]^T$	$u_4 = u_{fb}(x, v_x^a, v_z^p)$
Climb	q_5	$y_5 = [\dot{p}_x, \dot{p}_z]^T$	$u_5 = u_{fb}(x, v_x^v, v_z^v)$
Descend	q_6	$y_6 = [\dot{p}_x, \dot{p}_z]^T$	$u_6 = u_{fb}(x, v_x^v, v_z^v)$

6.4. Flight Mode Switching

We simulate a high-attitude take-off process where the helicopter climbs from 2m to 10m with maximum speed 5m/s and climbing angle 20°. It has successively the following flight modes: Hover, Acc, Cruise, Climb, and Cruise. An automaton for mode switching is shown in Figure 4.

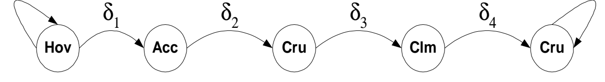

Figure 4. Flight mode switches.

In the automaton, δ_1 is the pilot input event that starts the climbing action;

$$\delta_2 \text{ is } ((V > 5) \wedge (|p_z - 2| < \varepsilon_z)); \qquad (15)$$

$$\delta_3 \text{ is } ((|V - 5| < \varepsilon_v) \wedge (|\gamma| < \varepsilon_\gamma)); \qquad (16)$$

$$\delta_4 \text{ is } |p_z - 10| < \varepsilon_z \quad , \qquad (17)$$

where total velocity $V = \sqrt{\dot{p}_x^2 + \dot{p}_z^2}$, and flight path angle $\gamma = \tan^{-1}(\dot{p}_z / \dot{p}_x)$.

6.5. Modeling The Helicopter in Ptolemy II

The hybrid system is modeled in Ptolemy II as Figure 5. The system has three levels of hierarchy. The top level is a continuous time system, with a hybrid controller. The controller implements the automaton that controls the switch-

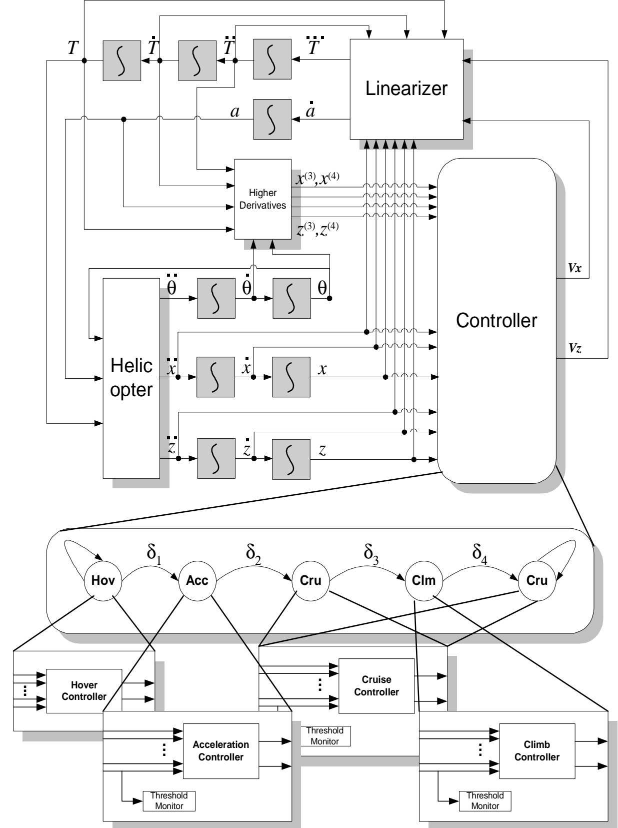

Figure 5. The helicopter model in Ptolemy II

ing of flight modes. In each flight mode, there is a concrete controller that computes the control output given the state of the helicopter. The simulation runs as a Java applet, and the result is shown in Figure 6.

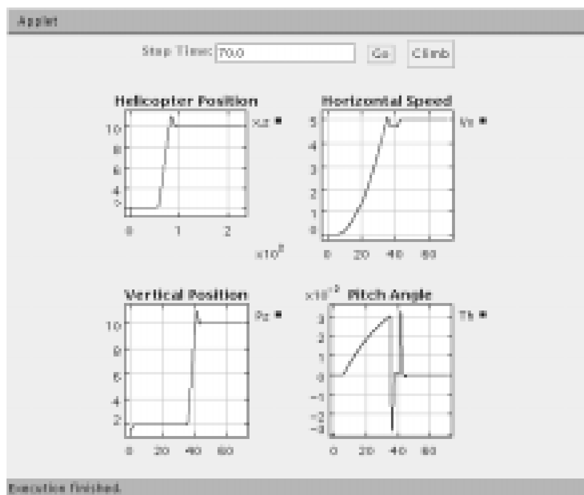

Figure 6. The result of the simulation.

7. Conclusion

This paper describes hybrid system modeling in the Ptolemy II environment. Hybrid systems are modeled hierarchically in Ptolemy II using finite state automata and continuous dynamical systems. The simulation techniques of both domains are studied. Event detection and invariant monitoring are achieved by breakpoint handling and integration step size control. A hierarchical helicopter control system is simulated as a case study.

Acknowledgments

This work is supported by the DARPA under grant F33615-98-C-3614. It is also part of the Ptolemy project, which is also supported by the State of California MICRO program, and the following companies: The Cadence Design Systems, Hewlett Packard, Hitachi, Hughes Space and Communications, Motorola, NEC, and Philips.

References

[1] M. Andersson, *OmSim and Omola Tutorial and User's Manual*, Dept. of Automatic Control, Lund Institute of Technology, Sweden, March 1995.

[2] P. Antsaklis, W. Kohn, A. Nerode, and S. Sastry (Eds.), *Hybrid Systems II*, LNCS 999, Springer Verlag, 1995.

[3] A. Balluchi, M. Di Benedetto, C. Pinelli, C. Rossi, and A. Sangiovanni-Vincentelli, "Hybrid Control for Automotive Engine Management: The Cut-Off Case." *Proc. of the 1st International Workshop on Hybrid Systems: Computation and Control* (HSCC'98), LNCS 1386, Springer 1998, pp. 13-32.

[4] B. Carlson and V. Gupta, "Hybrid cc with Interval Constraints." *Proc. of the 1st International Workshop on Hybrid Systems: Computation and Control* (HSCC'98), LNCS 1386, Springer 1998, pp. 80-95.

[5] J. Davis etc., "Overview of the Ptolemy Project." *ERL Technical Report* UCB/ERL No. M99/37, University of California, Berkeley, CA 94720, July 1999.

[6] C. W. Gear, *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice Hall Inc. 1971.

[7] T. A. Henzinger, "The Theory of Hybrid Automata." *Proc. of the 11th Annual Symposium on Logic in Computer Science*, IEEE Computer Press, 1996, pp 278-292.

[8] T.J. Koo and S. Sastry, "Output Tracking Control Design of a Helicopter Model Based on Approximate Linearization." *Proc. of IEEE Conference on Decision and Control*, Florida, Dec. 1998, pp 3635-3640.

[9] T.J. Koo, F. Hoffmann, H. Shim, B. Sinopoli, and S. Sastry, "Hybrid Control of Model Helicopter," *Proc. of IFAC Workshop on Motion Control*, Grenoble, France, Oct. 1999, pp 285-290.

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

[11]J. Liu, "Continuous Time and Mixed-Signal Simulation in Ptolemy II." *ERL Technical Report* UCB/ERL No. M98/74, University of California, Berkeley, CA 94720, Dec. 15 1998.

[12]J. Lygeros, C. Tomlin, and S. Sastry, "Controllers for Reachability Specifications for Hybrid Systems." *Automatica*, March 1999.

[13]N. Lynch, R. Segala, F. Vaandrager, and H.B. Weinburg, "Hybrid I/O automata," *Hybrid System III*, Springer Verlag, 1996, pp 496-510.

[14]P.J. Mosterman, "An overview of hybrid simulation phenomena and their support by simulation packages," *Proc. of the 2nd International Workshop on Hybrid Systems: Computation and Control* (HSCC'99), LNCS 1569, Springer, 1999, pp165-177.

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

[16]R.R. Bedeviled, *Learning to Fly Helicopters*, McGraw-Hill, 1992.

[17]T. Simsek, *SHIFT Tutorial: A first course for SHIFT programmers.* EECS, University of California, Berkeley. http://www.path.berkeley.edu/shift/tutorials.html

[18]J.H. Taylor, D. Kebede, "Modeling and Simulation of Hybrid Systems in MATLAB." *Proc. of IFAC World Congress*, Vol. J, San Francisco, USA, July 1996, pp 275-280.