Foundations and Trends® in
Electronic Design Automation
Vol. 1, No 1/2 (2006) 1–193

Languages and Tools for Hybrid Systems Design

Luca P. Carloni[1], Roberto Passerone[2], Alessandro Pinto[3] and Alberto L. Sangiovanni-Vincentelli[4]

[1] *Department of Computer Science, Columbia University, 1214 Amsterdam Avenue, Mail Code 0401, New York, NY 10027, USA, luca@cs.columbia.edu*
[2] *Cadence Berkeley Laboratories, 1995 University Ave Suite 460, Berkeley, CA 94704, USA, robp@cadence.com*
[3] *Department of EECS, University of California at Berkeley, Berkeley, CA 94720, USA, pinto@eecs.berkeley.edu*
[4] *Department of EECS, University of California at Berkeley, Berkeley, CA 94720, USA, alberto@eecs.berkeley.edu*

Abstract

The explosive growth of embedded electronics is bringing information and control systems of increasing complexity to every aspects of our lives. The most challenging designs are safety-critical systems, such as transportation systems (e.g., airplanes, cars, and trains), industrial plants and health care monitoring. The difficulties reside in accommodating constraints both on functionality and implementation. The correct behavior must be guaranteed under diverse states of the environment and potential failures; implementation has to meet cost, size, and power consumption requirements. The design is therefore subject to extensive mathematical analysis and simulation. However, traditional models of information systems do not interface well to the continuous evolving nature of the environment in which these devices operate. Thus, in practice, different mathematical representations have to be mixed to analyze the overall behavior of the system. *Hybrid systems* are a particular class of mixed models that focus on the combination of discrete and continuous subsystems. There is a wealth of tools and languages that have been proposed over the years to handle hybrid systems. However, each tool makes different assumptions on the environment, resulting in somewhat different notions of hybrid system. This makes it difficult to share information among tools. Thus, the community cannot maximally leverage the substantial amount of work that has been directed to this important topic. In this paper, we review and compare hybrid system tools by highlighting their differences in terms of their underlying semantics, expressive power and mathematical mechanisms. We conclude our review with a comparative summary, which suggests the need for a unifying approach to hybrid systems design. As a step in this direction, we make the case for a *semantic-aware interchange format*, which would enable the use of joint techniques, make a formal comparison between different approaches possible, and facilitate exporting and importing design representations.

1 Introduction

With the rapid advances in implementation technology, designers are given the opportunity of building systems whose complexity far exceeds the increase in rate of productivity afforded by traditional design paradigms. Design time has thus become the bottleneck for bringing new products to market. The most challenging designs are in the area of safety-critical embedded systems, such as the ones used to control the behavior of transportation systems (e.g., airplanes, cars, and trains) or industrial plants. The difficulties reside in accommodating constraints both on functionality and implementation. Functionality has to guarantee correct behavior under diverse states of the environment and potential failures; implementation has to meet cost, size, and power consumption requirements.

When designing embedded systems of this kind, it is essential to take all effects, including the interaction between environment (plant to be controlled) and design (digital controller) into consideration. This calls for methods that can deal with heterogeneous components exhibiting a variety of different behaviors. For example, digital controllers can be represented mathematically as discrete event systems, while plants are mostly represented by continuous time systems whose behavior is captured by partial or ordinary differential equations. In addition, the complexity of the plants is such that representing them at the detailed level is often impractical or even impossible. To cope with this complexity, abstraction is a very powerful method. Abstraction consists in eliminating details that do not affect the behavior of the system that we may be interested in. In both cases, different mathematical representations have to be mixed to analyze the overall behavior of the controlled system.

There are many difficulties in mixing different mathematical domains. *In primis*, the very meaning of interaction may be challenged. In fact, when heterogeneous systems are interfaced, interface variables are defined in different mathematical domains that may be incompatible. This aspect makes verification and synthesis impossible, unless a careful analysis of the interaction semantics is carried out.

In general, pragmatic solutions precede rigorous approaches to the solution of engineering problems. This case is no exception. Academic institutions and private software companies started developing computational tools for the simulation, analysis, and implementation of control systems (e.g., SIMULINK, STATEFLOW and MATLAB from The Mathworks), by first deploying common sense reasoning and then trying a formalization of the basic principles. These approaches focused on a particular class of heterogeneous systems: systems featuring the combination of discrete-event and continuous-time subsystems. Recently, these systems have been the subject of intense research by the academic community because of the interesting theoretical problems arising from their design and analysis as well as of the relevance in practical applications [2, 92, 133]. These systems are called *hybrid systems* [12, 14, 17, 18, 19, 20, 63, 80, 98, 131, 132, 134, 140, 163, 168].

Hybrid systems have proven to be powerful design representations for system-level design. While SIMULINK, STATEFLOW and MATLAB together provide excellent practical modeling and simulation capability for the design capture and the functional verification via simulation of embedded systems, there is a need for a more rigorous and domain-specific analysis as well as for methods to refine a high-level description into an implementation. There is a wealth of tools and languages that have been proposed over the years to handle hybrid systems. Each tool or language is based on somewhat different notions of hybrid systems and on assumptions that make a fair comparison difficult. In addition, sharing information among tools is almost impossible at this time, so that the community cannot leverage maximally the substantial amount of work that has been directed to this important topic.

In this survey, we collected data on available languages, formalism and tools that have been proposed in the past years for the design and verification of hybrid systems. We review and compare these tools by highlighting their differences in the underlying semantics, expressive power and solution mechanisms. Table 1.1 lists tools and languages reviewed in this survey with information on the institution that supports the development of each project as well as pointers to the corresponding web site[1] and to some relevant publications.

The tools are covered in two main sections: one dedicated to simulation-centric tools including commercial offerings, one dedicated to formal verification-centric tools. The simulation-centric tools are the most popular among designers as they pose the least number of constraints on the systems to be analyzed. On the other hand, their semantics are too general to be amenable to formal analysis or synthesis. Tools based on restricted expressiveness of the description languages (see, for example, the synthesizable subset of RTL languages as a way of allowing tools to operate on a more formal way that may yield substantial productivity gains) do have an appeal as they may be the ones to provide the competitive edge in terms of quality of results and cost for obtaining them. The essence is to balance the gains in analysis and synthesis power versus the loss of expressive power.

We organized each section describing a tool in

(1) a brief introduction to present the tool capabilities, the organizations supporting it and how to obtain the code;

[1] George Pappas research group at the Univ. of Pennsylvania is maintaining a WikiWiki-Web site at `http://wiki.grasp.upenn.edu/ graspdoc/hst/` whose objective is to serve as a community depository for software tools that have been developed for modeling, verifying, and designing hybrid and embedded control systems. It provides an "evolving" point of reference for the research community as well as potential users of all available technology and it maintains updated links to online resources for most of the tools listed on Table 1.1.

Table 1.1 References for the various modeling approaches, toolsets.

Name	Institution	Web Page	References	Section
Charon	Univ. of Pennsylvania	`www.cis.upenn.edu/mobies/charon/`	[3, 4, 8]	3.6
CheckMate	Carnegie Mellon Univ.	`www.ece.cmu.edu/~webk/checkmate/`	[151]	4.4
d/dt	Verimag	`www-verimag.imag.fr/~tdang/Tool-ddt/ddt.html`	[53, 21, 22]	4.8
Dymola	Dynasim AB	`www.dynasim.se/`	[67]	3.2
Ellipsoidal Toolbox	UC Berkeley	`www.eecs.berkeley.edu/~akurzhan/ellipsoids/`	[113, 120, 119]	4.7
HSolver	Max-Planck-Institut	`www.mpi-inf.mpg.de/~ratschan/hsolver/`	[147]	4.6
Hysdel	ETH Zurich	`www.control.ee.ethz.ch/~hybrid/hysdel/`	[166, 165]	4.9
HyTech	Cornell, UC Berkeley	`www-cad.eecs.berkeley.edu/~tah/HyTech`	[11, 88, 95]	4.2
HyVisual	UC Berkeley	`ptolemy.eecs.berkeley.edu/hyvisual`	[103]	3.3
Masaccio	UC Berkeley	`www.eecs.berkeley.edu/~tah`	[97]	4.3
Modelica	Modelica Association	`www.modelica.org`	[71, 162, 70]	3.2
PHAVer	VERIMAG	`www.cs.ru.nl/~goranf/`	[69]	4.5
Scicos	INRIA	`www.scicos.org`	[64, 143]	3.4
Shift	UC Berkeley	`www.path.berkeley.edu/shift`	[60, 61]	3.5
Simulink	The MathWorks	`www.mathworks.com/products/simulink`	[15, 52, 148]	3.1
Stateflow	The MathWorks	`www.mathworks.com/products/stateflow`	[15, 52, 148]	3.1
SynDEx	INRIA	`www-rocq.inria.fr/syndex`	[78, 79]	3.4

(2) a section describing the syntax of the language that describes the system to be analyzed;
(3) a section describing the semantics of the language;
(4) the application of the language and tool to two examples that have been selected to expose its most interesting features;
(5) a discussion on its pros and cons.

In the last part of the survey we provide a comparative summary of the hybrid system tools that we have presented. The resulting landscape appears rather fragmented. This suggests the need for a unifying approach to hybrid systems design. As a step in this direction, we make the case for a *semantic-aware interchange format*. Today, re-modeling the system in another tool's modeling language, when (at all) possible, requires substantial manual effort and maintaining consistency between models is error-prone and difficult in the absence of tool support. The interchange format, instead, would enable the use of joint techniques, make a formal comparison between different approaches possible, and facilitate exporting and importing design representations. The popularity of MATLAB, SIMULINK, and STATEFLOW implies that significant effort has already been invested in creating a large model-base in SIMULINK/STATEFLOW. It is desirable that application developers take advantage of this effort without foregoing the capabilities of their own analysis and synthesis tools. We believe that the future will be in automated semantic translators that, for instance, can interface with and translate the SIMULINK/STATEFLOW models into the models of different analysis and synthesis tools.

Survey organization. In Section 2, we lay the foundation for the analysis. In particular, we review the formal mathematical definition of hybrid systems (Section 2.1) and we define two examples (Section 2.2), a system of three point masses and a full wave rectifier, which will be used to compare and explain the tools and languages presented in this survey. In Section 3 we introduce and discuss the most relevant tools for simulation and design of hybrid and embedded systems. With respect to the industrial offering, we present the SIMULINK/STATEFLOW design environment, the MODELICA language, and the modeling and simulation tool DYMOLA based on it. Among the academic tools, we summarize the essential features of Scicos, Shift, HyVisual and Charon, a tool that is the bridge between the simulation tools and the formal verification tools as it supports both (although the verification component of Charon is not publicly available). In Section 4, we focus on tools for formal verification of hybrid systems. In particular, we discuss HyTech, PHAVer, HSolver, Masaccio, CheckMate, d/dt and Hysdel. The last two can also be used to synthesize a controller that governs the behavior of the system to follow desired patterns. We also summarize briefly tools based on the ellipsoidal calculus like Ellipsoidal Toolbox. In Section 5 we give a comparative summary of the design approaches, languages, and tools presented throughout this survey. To end in Section 6, we offer a discussion and a plan on the issues surrounding the construction of the interchange format.

2

Foundations

In this section, we discuss a general formal definition of hybrid systems as used in the control community. Most models used in the control community can be thought of as special cases of this general model. Then, we present two examples, which will be used in the rest of this survey to evaluate and compare different tools and languages for hybrid systems.

2.1 Formal definition of hybrid systems

The notion of a hybrid system traditionally used in the control community is a specific composition of discrete and continuous dynamics. In particular, a hybrid system has a continuous evolution and occasional jumps. The jumps correspond to the change of state in an automaton that transitions in response to external events or to the continuous evolution. A continuous evolution is associated to each state of the automaton by means of ordinary differential equations. The structure of the equations and the initial condition may be different for each state. While this informal description seems rather simple, the precise definition of the evolution of the system is quite complex.

Early work on formal models for hybrid systems includes *phase transition systems* [2] and *hybrid automata* [133]. These somewhat simple models were further generalized with the introduction of compositionality of parallel hybrid components in *hybrid I/O automata* [130] and *hybrid modules* [9]. In the sequel, we follow the classic work of Lygeros et al. [129] to formally describe a hybrid system as used in the control literature. We believe that this model is sufficiently general to form the basis of our work in future sections.

We consider subclasses of continuous dynamical systems over certain vector fields X, U and V for the continuous state, the input and disturbance, respectively. For this purpose, we denote with $\mathcal{U}_C$ the class of measurable input functions $u : \mathbb{R} \to U$, and with $\mathcal{U}_d$ the class of measurable disturbance functions $\delta : \mathbb{R} \to V$. We use the symbol $\mathbf{S}_C(X,U,V)$ to denote the class of continuous time dynamical systems defined by the equation

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

where $t \in \mathbb{R}$, $x(t) \in X$ and f is a function such that for all $u \in \mathcal{U}_C$ and for all $\delta \in \mathcal{U}_d$, the solution $x(t)$ exists and is unique for a given initial condition. A hybrid system can then be defined as follows.

Definition 2.1. (Hybrid System) A continuous time hybrid system is a tuple $\mathcal{H} = (\mathbf{Q}, \mathbf{U}_D, E, X, U, V, \mathcal{S}, Inv, R, G)$ where:

- $\mathbf{Q}$ is a set of states;
- $\mathbf{U}_D$ is a set of discrete inputs;
- $E \subset \mathbf{Q} \times \mathbf{U}_D \times \mathbf{Q}$ is a set of discrete transitions;
- X, U and V are the continuous state, the input and the disturbance, respectively;
- $\mathcal{S} : \mathbf{Q} \to \mathbf{S}_C(X,U,V)$ is a mapping associating to each discrete state a continuous time dynamical system;
- $Inv : \mathbf{Q} \to 2^{X \times \mathbf{U}_D \times U \times V}$ is a mapping called *invariant*;
- $R : E \times X \times U \times V \to 2^X$ is the reset mapping;
- $G : E \to 2^{X \times U \times V}$ is a mapping called *guard*.

Note that we can similarly define *discrete time* hybrid systems by simply replacing $\mathbb{R}$ with $\mathbb{Z}$ for the independent variable, and by considering classes of discrete dynamical systems underlying each state. The triple $(\mathbf{Q}, \mathbf{U}_D, E)$ can be viewed as an automaton having state set $\mathbf{Q}$, inputs $\mathbf{U}_D$ and transitions defined by E. This automaton characterizes the structure of the discrete transitions. Transitions may occur because of a discrete input event from $\mathbf{U}_D$, or because the invariant in Inv is not satisfied. The mapping $\mathcal{S}$ provides the association between the continuous time definition of the dynamical system in terms of differential equations and the discrete behavior in terms of states. The mapping R provides the initial conditions for the dynamical system upon entering a state.

The transition and dynamical structure of a hybrid system determines a set of *executions*. These are essentially functions over time for the evolution of the continuous state, as the system transitions through its discrete structure. To highlight the discrete structure, we introduce the concept of a hybrid time basis for the temporal evolution of the system, following [129].

Definition 2.2. (Hybrid Time Basis) A hybrid time basis τ is a finite or an infinite sequence of intervals

$$I_j = \{t \in \mathbb{R} : t_j \leq t \leq t'_j\}, \quad j \geq 0$$

where $t_j \leq t'_j$ and $t'_j = t_{j+1}$.

Let $\mathcal{T}$ be the set of all hybrid time bases. An execution of a hybrid system can then be defined as follows.

Definition 2.3. (Hybrid System Execution) An execution χ of a hybrid system $\mathcal{H}$, with initial state $\widehat{q} \in \mathbf{Q}$ and initial condition $x_0 \in X$, is a collection $\chi = (\widehat{q}, x_0, \tau, \sigma, q, u, \delta, \xi)$ where $\tau \in \mathcal{T}$, $\sigma : \tau \to \mathbf{U}_D$, $q : \tau \to \mathbf{Q}$, $u \in \mathcal{U}_C$, $\delta \in \mathcal{U}_d$ and $\xi : \mathbb{R} \times \mathbb{N} \to X$ satisfying:

(1) Discrete evolution:
- $q(I_0) = \widehat{q}$;
- for all j, $e_j = (q(I_j), \sigma(I_{j+1}), q(I_{j+1})) \in E$;

(2) Continuous evolution: the function ξ satisfies the conditions

- $\xi(t_0, 0) = x_0$;
- for all j and for all $t \in I_j$,

$$\xi(t, j) = x(t)$$

  where $x(t)$ is the solution at time t of the dynamical system $\mathcal{S}(q(I_j))$, with initial condition $x(t_j) = \xi(t_j, j)$, given the input function $u \in \mathcal{U}_C$ and disturbance function $\delta \in \mathcal{U}_d$;
- for all j, $\xi(t_{j+1}, j+1) \in R\left(e_j, \xi(t'_j, j), u(t'_j), v(t'_j)\right)$
- for all j and for all $t \in \left[t_j, t'_j\right]$,

$$(\xi(t, j), \sigma(I_j), u(t), v(t)) \in Inv\,(q(I_j))$$

- if τ is a finite sequence of length $L + 1$, and $t'_j \neq t'_L$, then

$$\left(\xi(t'_j, j), u(t'_j), v(t'_j)\right) \in G\,(e_j)$$

We say that the behavior of a hybrid system consists of all the executions that satisfy Definition 2.3. The constraint on discrete evolution ensures that the system transitions through the discrete states according to its transition relation E. The constraints on the continuous evolution, on the other hand, require that the execution satisfies the dynamical system for each of the states, and that it satisfies the invariant condition. Note that when the invariant condition is about to be violated, the system must take a transition to another state where the condition is satisfied. This implies the presence of an appropriate discrete input. Because a system may not determine its own inputs, this definition allows for executions with blocking behavior. When this is undesired, the system must be structured appropriately to allow transitions under any possible input in order to satisfy the invariant.

Note also that the same input may induce different valid executions. This is possible because two or more trajectories in the state machine may satisfy the same constraints. When this is the case, the system is non-deterministic. Non-determinism is important when specifying incomplete systems, or to model choice or don't care situations.

However, when describing implementations, it is convenient to have a deterministic specification. In this case, one can establish priorities among the transitions to make sure that the behavior of the system under a certain input is always well defined. Failure to take all cases of this kind into account is often the cause of the inconsistencies and ambiguities in models for hybrid systems.

Definition 2.4. A hybrid system execution is said to be (i) trivial if $\tau = \{I_0\}$ and $t_0 = t'_0$; (ii) finite if τ is a finite sequence; (iii) infinite if τ is an infinite sequence and $\sum_{j=0}^{\infty} t'_j - t_j = \infty$; (iv) Zeno, if τ is infinite but $\sum_{j=0}^{\infty} t'_j - t_j < \infty$.

In this survey, we are particularly concerned with Zeno behaviors and with simultaneous events and non-determinism, since different models often differ in the way these conditions are handled.

2.2 Examples

Comparing tools and languages is always difficult. To make the comparison more concrete, we selected two examples that are simple enough to be handled yet complex enough to expose strength and drawbacks.[1] This section describes in detail the two examples (a system of three point masses and a full wave rectifier) by using the notation introduced in Section 2.1.

2.2.1 Three-mass system

We consider a system (Figure 2.1) where three point masses, m_1, m_2 and m_3, are disposed on a frictionless surface (a table) of length L and height h. Mass m_1 has initial velocity $v_{1,0}$ while the other two masses are at rest. Mass m_1 eventually collides with m_2 which, in turn, collides with m_3. Consequently, mass m_3 falls from the table and starts bouncing on the ground. This system is not easy to model exactly [141], therefore we make some simplifying assumptions. Each collision is governed by the Newton's collision rule and the conservation of momentum. Let

[1] Links to the models for the example developed for the present survey are available at `http://embedded.eecs.berkeley.edu/hyinfo/`

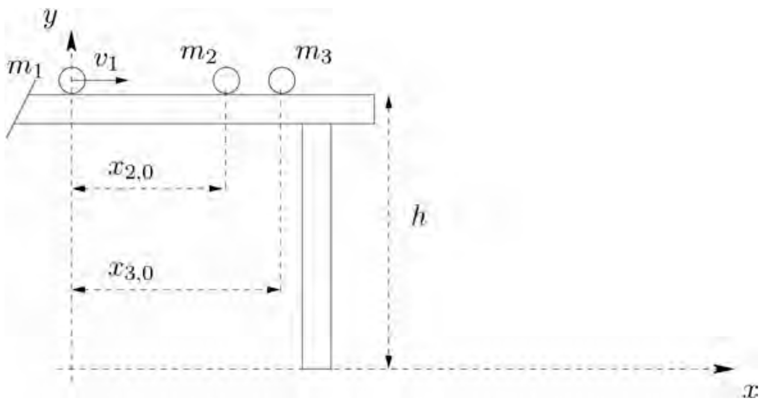

Fig. 2.1 The system with three point masses.

m_1 and m_2 be two colliding masses. Let v_i and v_i^+ denote the velocity before and after the collision, respectively. Then, Newton's rule states that $v_1^+ - v_2^+ = -\epsilon(v_1 - v_2)$, where ϵ is called the *coefficient of restitution*, which describes the loss of kinetic energy due to the collision. The conservation of momentum is the other equation that determines the velocities after the impact: $m_1(v_1^+ - v_1) = m_2(v_2 - v_2^+)$. A collision between m_1 and m_2 happens when $x_1 \geq x_2$ and $v_1 > v_2$, in which case the velocities after collisions are:

$$v_1^+ = v_1 \frac{(m_1 - \epsilon m_2)}{m_1 + m_2} + v_2 \frac{m_2(1+\epsilon)}{m_1 + m_2}$$
$$v_2^+ = v_1 \frac{(1+\epsilon) m_1}{m_1 + m_2} + v_2 \frac{(m_2 - \epsilon m_1)}{m_1 + m_2}$$

We assume that $x_{2,0} < x_{3,0}$.

Different tools provide different features to model hybrid systems and there are many ways of modeling this particular system. For instance, each point mass could be modeled as an independent system that only implements the laws of motion. A discrete automaton could be superimposed to the three dynamics to force discrete jumps in the state variables due to collisions and bounces. A possible hybrid system model is shown in Figure 2.2, where the position and velocity of each mass are chosen as state variables. Labels Cij represent guards and reset maps in the case of a collision between mass i and mass j.

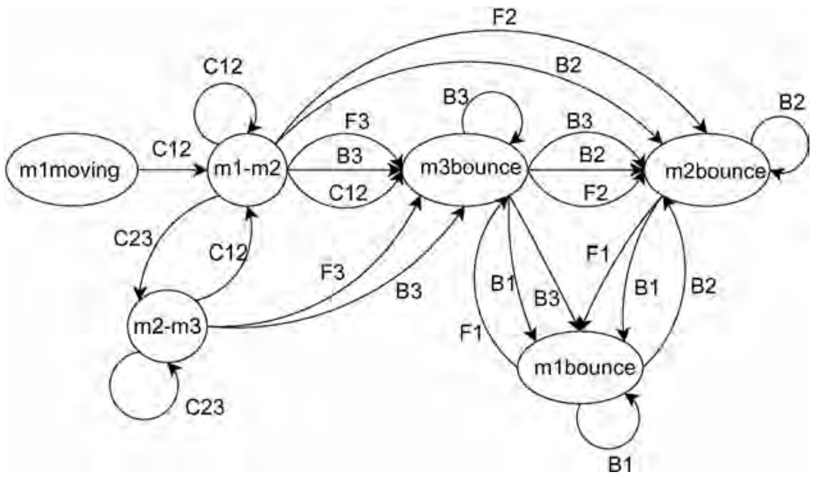

Fig. 2.2 The hybrid system modeling the three point masses.

Table 2.1 Guard conditions and reset maps for the hybrid system of Figure 2.2.

Label	Guard	Reset
C12	$x_1 \geq x_2 \wedge vx_1 > vx_2$	$vx_1 = vx_1^+ \wedge vx_2 = vx_2^+$
C23	$x_2 \geq x_3 \wedge vx_2 > vx_3$	$vx_2 = vx_2^+ \wedge vx_3 = vx_3^+$
F1	$x_1 \geq L \wedge y_1 > 0 \wedge vx_1 > 0$	$ay_1 = -g$
F2	$x_2 \geq L \wedge y_2 > 0 \wedge vx_2 > 0$	$ay_2 = -g$
F3	$x_3 \geq L \wedge y_3 > 0 \wedge vx_3 > 0$	$ay_3 = -g$
B1	$y_1 \leq 0 \wedge vy_1 < 0$	$vx_1 = \gamma_x vx_1 \wedge vy_1 = -\gamma_y vy_1$
B2	$y_2 \leq 0 \wedge vy_2 < 0$	$vx_2 = \gamma_x vx_2 \wedge vy_2 = -\gamma_y vy_2$
B3	$y_3 \leq 0 \wedge vy_3 < 0$	$vx_3 = \gamma_x vx_3 \wedge vy_3 = -\gamma_y vy_3$

Labels Fi represent guards and reset maps when mass i falls from the table. Finally, labels Bi represent guards and reset maps when mass i bounces on the ground. The coefficients γ_x and γ_y model the loss of energy on the x and y directions due to the bounce. We assume that in each state the invariant is the conjunction of the complement of the guards on the output transitions (or, equivalently, that guards have an "as is" semantics). Guard conditions and reset maps for each transition are listed in Table 2.1.

The system behavior starts with all the masses on the table. All accelerations are set to zero, $y_i = h$, $i = 1,2,3$ (all masses on the table top), $x_i = x_{i,0}$, $i = 2,3$ and $x_1 = 0$. Also, m_1 is initially moving with velocity $v_{1,0} > 0$ while the other two masses have zero initial velocity. Mass m_1 moves to the right and collides with m_2 (state $m_1 - m_2$). Mass m_2, after collision, moves to the right and collides with m_3 (state $m_2 - m_3$). Eventually m_3 falls off the table (transition $F3$) and starts bouncing (state $m_3 - bounce$ and transitions $B3$). We consider both a vertical and horizontal loss of energy in the bounce as to denote that the surface at $y = 0$ manifest some friction. While m_3 bounces on the ground, the other two masses (depending on the values of m_1, m_2 and m_3) can either stop on the table or eventually fall off and bounce. In each state, the dynamics is captured by a set of linear differential equations. If we denote the horizontal and vertical components of the velocity and of the acceleration by vx, ax and vy, ay, then the equations are: $dvx_i/dt = ax_i$, $dx_i/dt = vx_i$, $dvy_i/dt = ay_i$, $dy_i/dt = vy_i$.

The three-mass system shows interesting simulation phenomena. When $x_{3,0} = L$ (mass number 3 is positioned at the very edge of the table), three events occur at the same time: m_2 collides with m_3 and then both masses fall (event iteration). Even if events happen simultaneously, they are sequentially ordered. This is the main reason for having several states with the same dynamics. A hybrid system with only one state would be non-deterministic and incapable of ordering events in the proper way. When m_2 and m_3 fall at the same time, they also bounce at the same time, which makes the hybrid automaton non-deterministic since the bouncing events can be arbitrarily ordered. Finally, this systems is Zeno because at least m_3 will eventually fall and it's behavior becomes the one of a bouncing ball.

2.2.2 Full wave rectifier

Our second example, shown in Figure 2.3, is a full wave rectifier, which is used to obtain a constant voltage source starting from a sinusoidal one. Let $v_{in} = A\sin(2\pi f_0 t)$ be the input voltage. The idea behind this circuit is very simple: when $v_{in} > 0$, diode D_1 is in forward polarization while D_2 is in reverse polarization; when $v_{in} < 0$, diode D_2 is in forward polarization while D_1 is in reverse polarization. In both cases the current flows in the load in the same direction. Diodes are modeled by two states. In the off state, i.e., $va_i - vk_i \leq v_\gamma$, the current flowing through them is equal to $-I_0$. In the on state, i.e., $va_i - vk_i \geq v_\gamma$, the current

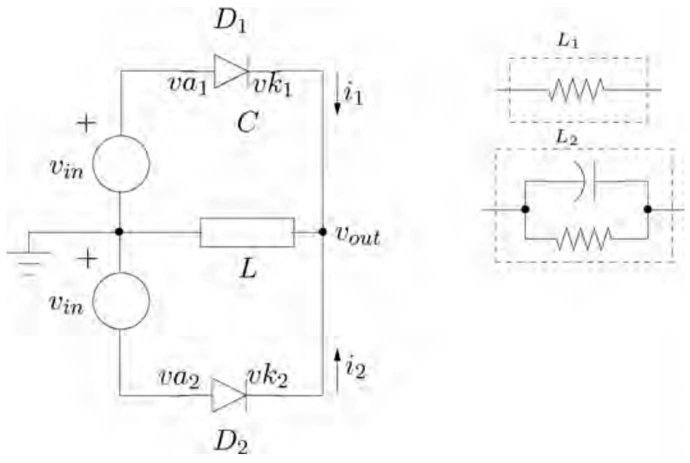

Fig. 2.3 A full wave rectifier circuit.

is equal to $I_0 e^{\frac{va_i - vk_i}{V_T}}$. The currents in the two diodes depend on v_{out}, which depends on the sum of the two currents. We model the diode as a resistor of value 0.1Ω in forward polarization and as an independent current source of value $-I_0 A$ in backward polarization. We have two candidates for the load L: L_1 is a pure resistor while L_2 is the parallel connection of a resistor and a capacitor. When the load is the pure resistor L_1 we observe the algebraic loop $v_{out} \to i_i \to v_{out}$. In order to determine $v_{out}(t)$ at time t the values of $i_1(t)$ and $i_2(t)$ must be known but they depends on the value $v_{out}(t)$ at the very same time. If the load is the parallel composition of a resistor and a capacitor L_2, then v_{out} is the solution of a differential equation and the algebraic loop problem disappears because the derivative operator acts as a delay in a loop of combinational operators.

Figure 2.4 shows the discrete automaton representing the full-wave rectifier system. There are fours states, representing the different working condition combinations of the two diodes. In all four cases, the continuous dynamics for the voltages is described by the following equations:

$$\begin{aligned} v_{in} &= \sin(2\pi f t) \\ \dot{v}_{out} &= -\frac{v_{out}}{RC} + \frac{i_1 + i_2}{C} \end{aligned}$$

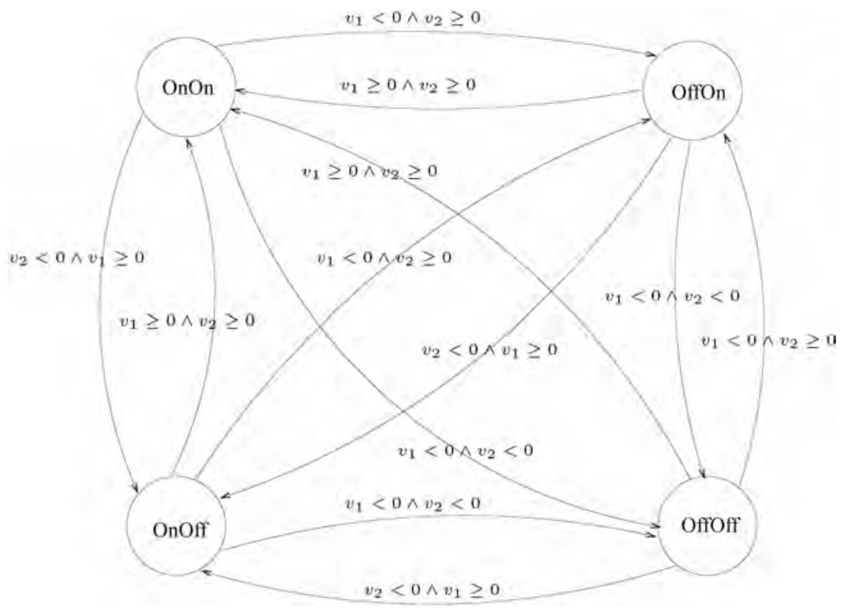
$v_1 < 0 \wedge v_2 \geq 0$
OnOn
$v_1 \geq 0 \wedge v_2 \geq 0$
OffOn
$v_1 \geq 0 \wedge v_2 \geq 0$
$v_2 < 0 \wedge v_1 \geq 0$
$v_1 < 0 \wedge v_2 \geq 0$
$v_1 < 0 \wedge v_2 < 0$
$v_1 \geq 0 \wedge v_2 \geq 0$
$v_1 < 0 \wedge v_2 \geq 0$
$v_2 < 0 \wedge v_1 \geq 0$
$v_1 < 0 \wedge v_2 < 0$
$v_1 < 0 \wedge v_2 < 0$
OnOff
OffOff
$v_2 < 0 \wedge v_1 \geq 0$

Fig. 2.4 A full wave rectifier hybrid system model.

$$v_1 = v_{in} - v_{out}$$
$$v_2 = -v_{in} - v_{out}$$

The dynamics for the currents i_1 and i_2 and the invariant conditions for each state are as follows:

- OnOn: both diodes are on. The continuous dynamics is described by the additional equations:

  $$i_1 = v_1/R_f$$
  $$i_2 = v_2/R_f$$

  and the invariant is $v_1 \geq 0 \wedge v_2 \geq 0$.
- OnOff: d_1 is on and d_2 is off. The continuous dynamics is described by the additional equations:

  $$i_1 = v_1/R_f$$
  $$i_2 = -I_0$$

  and the invariant is $v_1 \geq 0 \wedge v_2 < 0$.

- OffOn: d_2 is on and d_1 is off. The continuous dynamics is described by the additional equations:

$$\begin{aligned} i_1 &= -I_0 \\ i_2 &= v_2/R_f \end{aligned}$$

  and the invariant is $v_1 < 0 \wedge v_2 \geq 0$.
- OffOff: both diodes are off. The continuous dynamics is described by the additional equations:

$$\begin{aligned} i_1 &= -I_0 \\ i_2 &= -I_0 \end{aligned}$$

  and the invariant is $v_1 < 0 \wedge v_2 < 0$.

3

Tools for Simulation

Historically, the first computer tool to be used for designing complex systems has been simulation. Simulation substitutes extensive testing after manufacturing and, as such, it can reduce design costs and time. Of course, the degree of confidence in the correctness of the design is limited as unpredicted interactions with the environment go unchecked since the input size is too large to allow for exhaustive analysis.

The design of hybrid systems is no exception and the most used and popular tools are indeed simulation based. In this domain, there are strong industrial offerings that are widely used: first and foremost the SIMULINK/STATEFLOW toolset that has become the *de facto standard* in industry for system design capture and analysis. The MODELICA language with the DYMOLA simulation environment is also popular and offers a solid toolset. Together with these industrial tools, there are freely available advanced tools developed in academia that are getting attention from the hybrid system community. HYVISUAL recently developed at U.C. Berkeley, SCICOS developed at INRIA, SHIFT also developed at U.C. Berkeley and CHARON developed at University of Pennsylvania are reviewed here. CHARON is actually a bridge to the formal verification domain as it offers not only simulation but also formal verification tools based on the same language.

Each of the tools under investigation in this section is characterized by the language used to capture the design. While SIMULINK/STATEFLOW, MODELICA and SCICOS offer a general formalism to capture hybrid systems (hence their expressive power is large), the properties of the systems captured in these languages are difficult to analyze. The CHARON language is more restrictive but, because of this, offers an easier path to verification and, in fact, the same input mechanism is used for the formal verification suite.

3.1 Simulink and Stateflow

In this section, we describe the data models of SIMULINK and STATEFLOW. The information provided below is derived from the SIMULINK documentation as well as by "reverse engineering" SIMULINK/STATEFLOW models.[1]

SIMULINK and STATEFLOW are two interactive tools that are integrated within the popular MATLAB environment for technical computing marketed by The MathWorks. MATLAB integrates computation, visualization, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation. SIMULINK is an interactive tool for modeling and simulating nonlinear dynamical systems. It can work with linear, nonlinear, continuous-time, discrete-time, multi-variable, and multi-rate systems. STATEFLOW is an interactive design and development tool for complex control and supervisory logic problems. STATEFLOW supports visual modeling and simulation of complex reactive systems by simultaneously using finite state machine (FSM) concepts, STATECHARTS formalisms [84], and flow diagram notations. A STATEFLOW model can be included in a SIMULINK model as a subsystem.

Together with SIMULINK and STATEFLOW, MATLAB has become the *de facto* design-capture standard in academia and industry for control and data-flow applications that mix continuous and discrete-time

[1] We have also drawn from a technical report by S. Neema [142].

domains. The graphical input language together with the simulation and symbolic manipulation tools create a powerful toolset for system design. The tools are based on a particular mathematical formalism, a language, necessary to analyze and simulate the design. Unfortunately, the semantics of the language is not formally defined. For this reason, we discuss the aspects of the Simulink/Stateflow semantics as data models. As discussed below, the behavior of the design depends upon the execution of the associated simulation engine and the engine itself has somewhat ambiguous execution rules.

3.1.1 Simulink/Stateflow Syntax

Both Simulink and Stateflow are graphical languages. Simulink graphical syntax is very intuitive (and this is also the reason why this language is so popular). A system in Simulink is described as a collection of *blocks* that compute the value of their outputs as a function of their inputs. Blocks communicate through connectors that are attached to their *ports*. A subsystem can be defined as the interconnection of primitive blocks or of other subsystems, and by specifying its primary input and output ports. Once defined, subsystems can be used to specify other subsystems in a hierarchical fashion. Simulink has a rich library of primitive components that can be used to describe a system. The library is composed of six fundamental block sets:

- Continuous: blocks for processing continuous signals such as the Derivative and Integrator blocks; more complex continuous time operators, like State-Space blocks that can be used to model dynamical systems described by state equations; Zero-Pole blocks that can be used to describe transfer functions in the s domain.
- Discrete: blocks for processing discrete signals; most of these are descriptions of transfer functions in the z domain; Discrete Zero-Pole, Discrete State-Space, and Discrete-Time Integrator are examples of blocks that can be instantiated and parameterized in a Simulink model. Discrete blocks have a *Sample Time* parameter that specifies the rate of a periodic execution. This library also includes Unit Time and Zero-Order Hold,

which are important "interface blocks" in modeling multi-rate systems with SIMULINK. Specifically, a Unit Delay blocks must be inserted when moving from a slow-rate to a fast-rate block and a Zero-Order Hold is necessary in the other case [45, 125].

- Math Operations: general library of blocks representing mathematical operations like Sum, Dot Product, and Abs (absolute value).
- Sinks: signal consumers that can be used to display and store the results of the computation or to define the boundaries of the hierarchy. There are several types of display blocks for run time graph generation. It is possible to store simulation results in a MATLAB workspace variable for post-processing. Output ports are special type of Sinks.
- Sources: various signal generators that can be used as stimuli for test-benches; input ports are a special type of Sources.
- Discontinuities: non-linear transformations of signals such as Saturation and Quantizers; the Hit Crossing block is very useful for modeling hybrid systems: this block has a *threshold* parameter and it generates an output event when the threshold is hit.

The SIMULINK syntax supports the definition of subsystems that can be instantiated in a SIMULINK model allowing designers to use hierarchy in the organization of their designs. A STATEFLOW model can be instantiated as a block within a SIMULINK model. The syntax of STATEFLOW is similar to that of STATECHARTS. A STATEFLOW model is a set of states connected by arcs. A state is represented by a rounded rectangle. A state can be refined into a STATEFLOW diagram, thus creating a hierarchical state machine. A STATEFLOW model can have data input/output ports as well as event input/output ports. Both data and events can be defined as local to the STATEFLOW model or external, i.e., coming from the SIMULINK parent model in which case, data and events are communicated through ports.

Each arc, or transition, has a label with the following syntax:

```
event[condition]{condition_action}/transition_action
```

Transitions can join states directly, or can be joined together using *connective junctions* to make composite transitions that simulate if ... then ... else and loop constructs. Each segment of a composite transition is called a *transition segment*. A transition is "attempted" whenever its event is enabled and the condition is true. In that case, the condition action is executed. If the transition connects directly to a destination state, then control is passed back to the source state that executes its exit action (see below), then the transition executes its transition action, and finally the state change takes place by making the destination state active. On the other hand, if the transition ends at a connective junction, the system checks if any of the outgoing transition segments is enabled, and further attempts to reach a destination state. If no path through the transition segments can be found to reach a destination state, then the source state remains active and no state change takes place. Note, however, that, in the process, some of the condition actions might have been executed. This is essential to simulate the behavior of certain control flow constructs over the transitions, and at the same time distinguish with the actions to be taken upon a state change.

A state has a label with the following syntax:

```
name/
entry:entry action
during:during action
exit:exit action
on event_name:on event_name action
```

The identifier name denotes the name of the state; the entry action is executed upon entering the state; the during action is executed whenever the model is evaluated and the state cannot be left; the exit action is executed when the state is left; finally, the event_name action is executed each time the specified event is enabled.

3.1.2 Simulink/Stateflow Semantics

The Simulink Data Model. Simulink is a simulation environment that supports the analysis of mixed discrete-time and continuous-time models. Different simulation techniques are used according to whether continuous blocks and/or discrete blocks are present. We discuss only the case in which both components are present.

A SIMULINK project[2] is stored in an ASCII text file in a specific format referred to as Model File Format in the SIMULINK documentation. The SIMULINK project files are suffixed with ".mdl" and therefore we may occasionally refer to a SIMULINK project file as an "mdl file". There is a clear decoupling between the SIMULINK and the STATEFLOW models. When a SIMULINK project contains STATEFLOW models, the STATEFLOW models are stored in a separate section in the mdl file. We present STATEFLOW models separately in the next section. The data models presented here capture only the information that is being exposed by SIMULINK in the mdl file. Note that a substantial amount of semantics information that is sometimes required for the effective understanding of the SIMULINK models is hidden in the MATLAB simulation engine, or in the SIMULINK primitive library database.

The SIMULINK simulation engine deals with the components of the design by using the continuous-time semantic domain as a unifying domain whenever both continuous and discrete-time components are present. In fact, discrete-time signals are just piecewise-constant continuous-time signals. In particular, the inputs of discrete block is sampled at multiples of its *Sample Time* parameter while its outputs are piecewise-constant signals.

The simulation engine includes a set of integration algorithms, called *solvers*, which are based on the MATLAB ordinary differential equation (ODE) suite. A sophisticated ODE solver uses a variable time-step algorithm that adaptively selects a time-step tuned to the smallest time constant of the system (i.e., its fastest mode). The algorithm allows for errors in estimating the correct time-step and it back-tracks whenever the truncation error exceeds a bound given by the user. All signals of the system must be evaluated at the time-step dictated by the integration algorithm even if no event is present at these times. A number of multirate integration algorithms have been proposed for ODEs to improve the efficiency of the simulators but they have a serious overhead that

[2] In order to avoid any ambiguity, a complete model of a system in SIMULINK will be referred to as a "SIMULINK project".

may make them even slower than the original conservative algorithm. MATLAB provides a set of solvers that the user can choose from to handle either stiff (e.g., ODE15S) or non-stiff (e.g., ODE23) problems.

The most difficult part for a mixed-mode simulator that has to deal with discrete-events as well as continuous-time dynamics is managing the interaction between the two domains. In fact, the evolution of the continuous-time dynamics may trigger a discrete event at a time that is not known *a priori*. The trigger may be controlled by the value of a continuous variable, in which case detecting when the variable assumes a particular value is of great importance as the time at which the value is crossed is essential to have a correct simulation. This time is often difficult to obtain accurately. In particular, simulation engines have to use a sort of bisection algorithm to bracket the time value of interest. Numerical noise can cause serious accuracy problems. SIMULINK has a predefined block called *zero-crossing* that forces the simulator to accurately detect the time when a particular variable assumes the zero value.

In SIMULINK, there is the option of using fixed time-step integration methods. The control part of the simulator simplifies considerably, but there are a few problems that may arise. If the system is *stiff*, i.e., there are substantially different time constants, the integration method has to use a time step that, for stability reasons, is determined by the fastest mode. This yields an obvious inefficiency when the fast modes die out and the behavior of the system is determined only by the slower modes. In addition, an *a priori* knowledge of the time constants is needed to select the appropriate time step. Finally, not being able to control the time step may cause the simulation to be inaccurate in estimating the time at which a jump occurs, or even miss the jump altogether!

The computations of the value of the variables are scheduled according to the time step. Whenever there is a static dependency among variables at a time step, a set of simultaneous algebraic equations must be solved. Newton-like algorithms are used to compute the solution of the set of simultaneous equations. When the design is an aggregation of subsystems, the subsystems may be connected in ways that result in ambiguity in the computation. For example, consider a subsystem A with two outputs: one to subsystem B and one to subsystem C.

Subsystem B has an output that feeds C. In this case, we may evaluate the output of C whenever we have computed one of its inputs. Assuming that A has been processed, then we have the choice of evaluating the outputs of B or of C. Depending on the choice of processing B or C, the outputs of C may have different values! Simultaneous events may yield a non-deterministic behavior. In fact, both cases are in principle correct behaviors unless we load the presence of connections among blocks with causality semantics. In this case, B *has* to be processed before C. Like many other simulators, SIMULINK deals with non-determinism with scheduling choices that cannot be but arbitrary unless a careful (and often times expensive) causality analysis is carried out. Even when a causality analysis is available, there are cases where the non-determinism cannot be avoided since it is intrinsic in the model. In this case, scheduling has to be somewhat arbitrary. If the user knows what scheme is used and has some control on it, he/she may adopt the scheduling algorithm that better reflects what he/she has in mind. However, if the choice of the processing order is done *inside* the simulator according, for example, to a lexicographical order, changing the name of the variables (or of the subsystems) may change the behavior of the system itself! Since the inner workings of the simulation engines are often not documented, unexpected results and inconsistencies may occur. This phenomenon is well known in hardware design when Hardware Description Languages (HDLs) are used to represent a design at the register-transfer level (RTL) and a RTL simulator is used to analyze the system. For example, two different RTL simulators may give two different results even if the representation of the design is identical, or if it differs solely on the names of the subsystems and on the order in which the subsystems are entered.

The STATEFLOW **Data Model.** STATEFLOW models the behavior of dynamical systems based on finite state machines. The STATEFLOW modeling formalism is derived from STATECHARTS developed by Harel [84]. The essential differences from STATECHARTS are in the action language. The STATEFLOW action language has been extended primarily to reference MATLAB functions, and MATLAB workspace variables. Moreover, the concept of condition action has been added to the transition expression.

The interaction between SIMULINK and STATEFLOW occurs at the event and data boundaries. The simulation of a system consisting of SIMULINK and STATEFLOW models is carried out by alternatively releasing the control of the execution to the two simulation engines embedded in the two tools. In the hardware literature, this mechanism is referred to as *co-simulation*. Since control changes from one engine to the other, there is an overhead that may be quite significant when events are exchanged frequently. An alternative simulation mechanism would consist of a unified engine. This, however, would require a substantial overhaul of the tools and of the underlying semantic models.

3.1.3 Examples

A moving point mass can be modeled in SIMULINK as the subsystem shown in Figure 3.1. The two accelerations ax and ay are integrated to obtain the two velocities vx and vy, which are integrated to obtain the

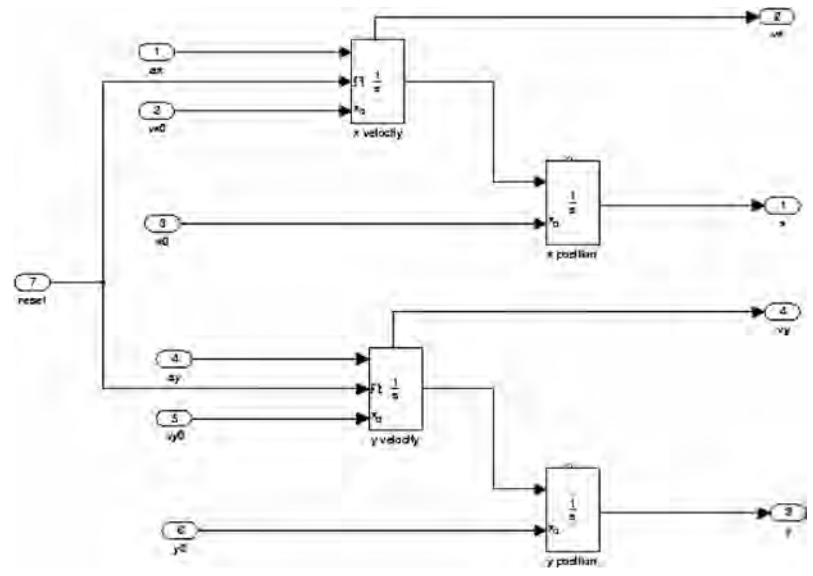

Fig. 3.1 Model of single moving mass in SIMULINK/STATEFLOW.

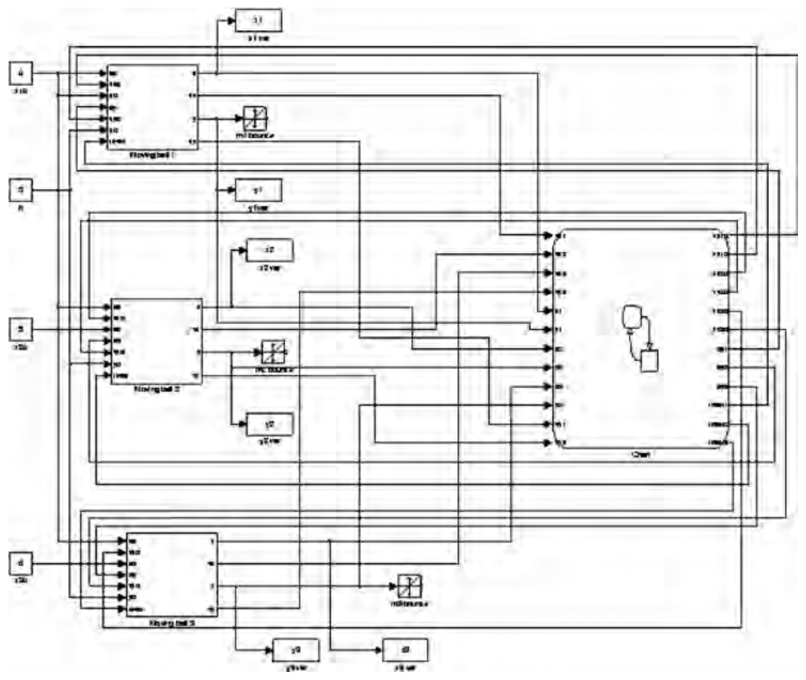

Fig. 3.2 Model of the three point masses in SIMULINK/STATEFLOW.

positions x and y. The subsystem has a reset input that forces the integrators to be loaded with the initial conditions vx_0, vy_0, x_0, y_0 provided externally. In order to avoid algebraic loops through the STATEFLOW model, outputs are taken from the integrators' state ports which represent the outputs of the integrators at the previous time stamp. The system discussed in Section 2.2.1 can be modeled by instantiating and coordinating three of the point mass subsystems. The entire system is shown in Figure 3.2. The *Chart* block is a STATEFLOW model describing the discrete automaton that is shown in Figure 3.3. We assume $m_1 = m_2 = m_3$. The STATEFLOW chart is a hierarchical state machine. There are four states:

- allon in which all point masses are on the table. The entry state is m1moving in which only m_1 is moving to the right. The first mass that falls off the table is m_3 because masses are not allowed to make vertical jumps. In this state two

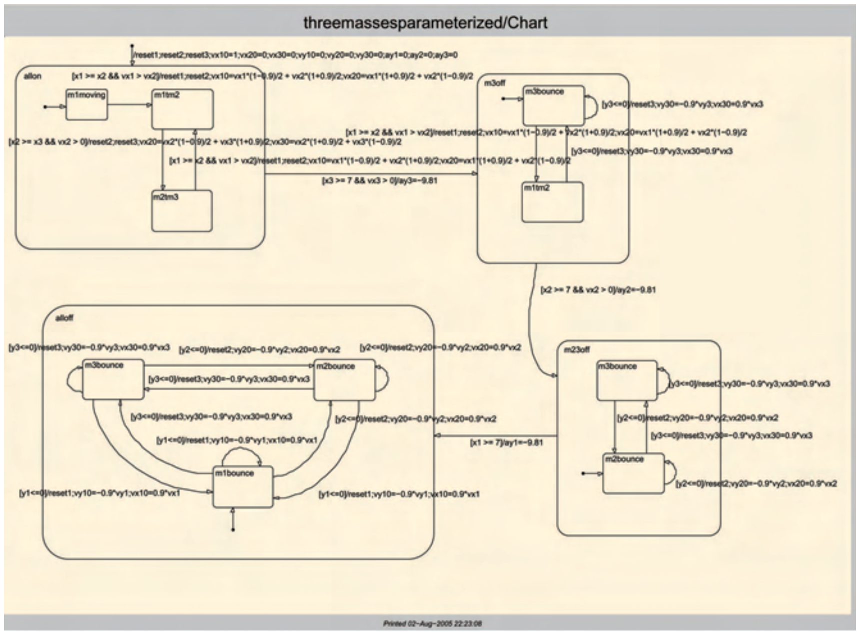
threemassesparameterized/Chart
/reset1;reset2;reset3;vx10=1;vx20=0;vx30=0;vy10=0;vy20=0;vy30=0;ay1=0;ay2=0;ay3=0
allon
[x1 >= x2 && vx1 > vx2]/reset1;reset2;vx10=vx1*(1-0.9)/2 + vx2*(1+0.9)/2;vx20=vx1*(1+0.9)/2 + vx2*(1-0.9)/2
m1moving
m1tm2
[x2 >= x3 && vx2 > 0]/reset2;reset3;vx20=vx2*(1-0.9)/2 + vx3*(1+0.9)/2;vx30=vx2*(1+0.9)/2 + vx3*(1-0.9)/2
[x1 >= x2 && vx1 > vx2]/reset1;reset2;vx10=vx1*(1-0.9)/2 + vx2*(1+0.9)/2;vx20=vx1*(1+0.9)/2 + vx2*(1-0.9)/2
m2tm3
[x3 >= 7 && vx3 > 0]/ay3=-9.81
m3off
m3bounce
[y3<=0]/reset3;vy30=-0.9*vy3;vx30=0.9*vx3
[x1 >= x2 && vx1 > vx2]/reset1;reset2;vx10=vx1*(1-0.9)/2 + vx2*(1+0.9)/2;vx20=vx1*(1+0.9)/2 + vx2*(1-0.9)/2
[y3<=0]/reset3;vy30=-0.9*vy3;vx30=0.9*vx3
m1tm2
[x2 >= 7 && vx2 > 0]/ay2=-9.81
alloff
[y3<=0]/reset3;vy30=-0.9*vy3;vx30=0.9*vx3
[y2<=0]/reset2;vy20=-0.9*vy2;vx20=0.9*vx2
[y2<=0]/reset2;vy20=-0.9*vy2;vx20=0.9*vx2
m3bounce
m2bounce
[y3<=0]/reset3;vy30=-0.9*vy3;vx30=0.9*vx3
[y3<=0]/reset3;vy30=-0.9*vy3;vx30=0.9*vx3
[y2<=0]/reset2;vy20=-0.9*vy2;vx20=0.9*vx2
[y1<=0]/reset1;vy10=-0.9*vy1;vx10=0.9*vx1
m1bounce
[y1<=0]/reset1;vy10=-0.9*vy1;vx10=0.9*vx1
[y1<=0]/reset1;vy10=-0.9*vy1;vx10=0.9*vx1
[x1 >= 7]/ay1=-9.81
m23off
m3bounce
[y3<=0]/reset3;vy30=-0.9*vy3;vx30=0.9*vx3
[y2<=0]/reset2;vy20=-0.9*vy2;vx20=0.9*vx2
[y3<=0]/reset3;vy30=-0.9*vy3;vx30=0.9*vx3
m2bounce
[y2<=0]/reset2;vy20=-0.9*vy2;vx20=0.9*vx2
Printed 02-Aug-2005 22:23:08

Fig. 3.3 Model of the three point masses automata in SIMULINK/STATEFLOW.

events can take place: m_1 collides with m_2 or m_2 collides with m_3.

- m3off in which m_3 has fallen. The transition to this state sets the vertical acceleration ay_3 to $-9.81m/s^2$ but does not reset the integrators' states. In this state either m_1 collides with m_2 or m_3 touches the ground.
- m23off in which m_2 has also fallen. The transition to this state sets the vertical acceleration ay_2 to $-9.81m/s^2$ but does not reset the integrators' states. In this state either m_3 touches the ground or m_2 does.
- alloff in which m_1 has fallen too. In this state any mass can touch the ground.

The simulation result is shown in Figure 3.4. We set $x_{2,0} = L - 0.5$, $x_{3,0} = L$, $\epsilon = 0.9$, $L = 7$ and $h = 3$.

The simulation result highlights how discrete and continuous states are updated. There is one integration step between the time when a

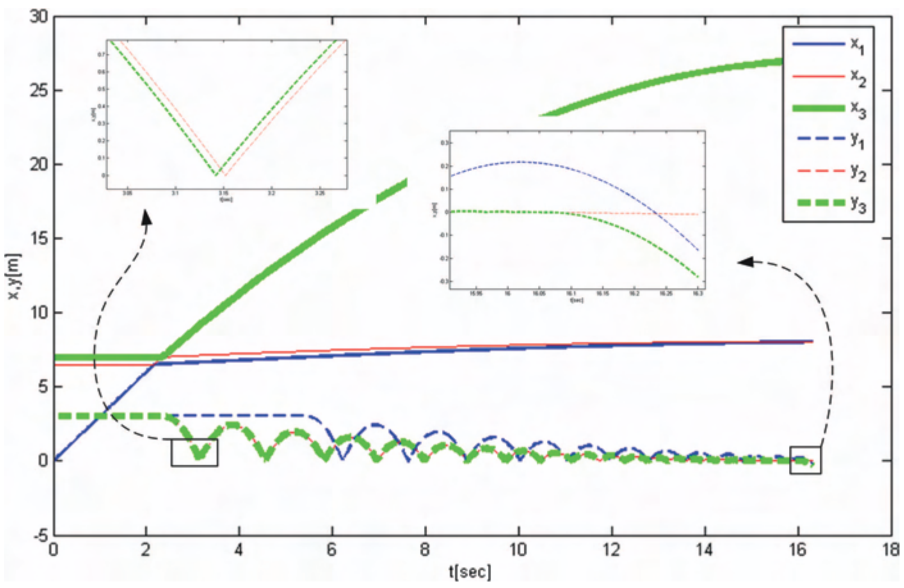

Fig. 3.4 Simulation result for the three-mass system.

guard becomes enabled and the time when a transition is taken. The delay is due to the fact that when a reset of the integrators is needed, SIMULINK blocks are executed for at least one integration step before passing the control back to the STATEFLOW chart. Things are different for the change in the values of the vertical accelerations. This change requires no reset and transitions can be taken in the STATEFLOW model in zero time. The time shift due to the reset of velocity propagates to the bounces of the two masses that occur at two different times, as shown in the enlarged inset on the left of Figure 3.4. Another simulation artifact is shown in the second inset at the right of Figure 3.4 at the end of the simulation. There, we see that masses m_1 and m_3 fall below the floor. This is because transitions are always interleaved with the integration step, and one of two events that occur simultaneously may therefore be lost. In this case, the system reacts to the bouncing of mass m_2, by taking the corresponding transition in state alloff shown in Figure 3.3. Subsequently, control is passed to the continuous time subsystem, which performs an integration step. Recall that an event is enabled when the evaluation of the condition changes from false to true. During the integration, the vertical position of masses m_1 and m_2

remains negative, thus disabling the corresponding event. Hence, the event, which was enabled at the previous step, is lost. This problem could be resolved by a more elaborate discrete model that takes into account the possibility of simultaneous events.

The Full Wave Rectifier Example. Figure 3.5 illustrates a SIMULINK model of the full wave rectifier system presented in Section 2.2.2. The bottom part of the figure shows a linearized model

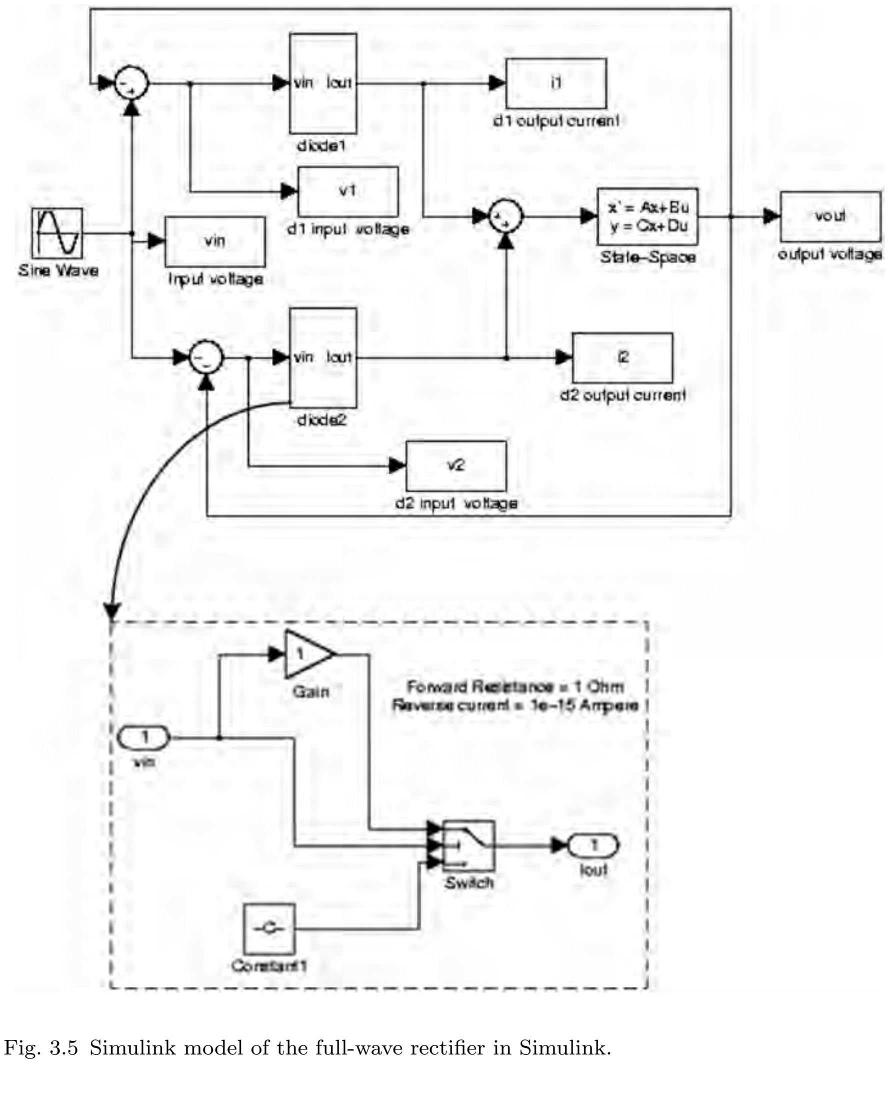

Fig. 3.5 Simulink model of the full-wave rectifier in Simulink.

of a diode. The *switch* block has three inputs: the middle pin controls which of the two other inputs is routed to the output. If the value of the control input is greater than zero, the output is proportional to the input voltage by a constant that represents the forward resistance. If the control input is less than zero then the current is equal to the reverse bias current. The sum of the currents in the two diodes is equal to the current through the load which is modeled as a linear dynamical system.

The simulation results are shown in Figure 3.6, where the correct functionality of the model can be validated. When the load is substituted with a simple constant (that models a pure resistive load), SIMULINK reports an error due to an algebraic loop. There are two possible solutions to this problem. The easiest one is to add a delay in the loop (right before or after the constant) so that the algebraic loop is eliminated. This solution is not always possible especially when adding a delay changes the stability properties of a feedback system. The other

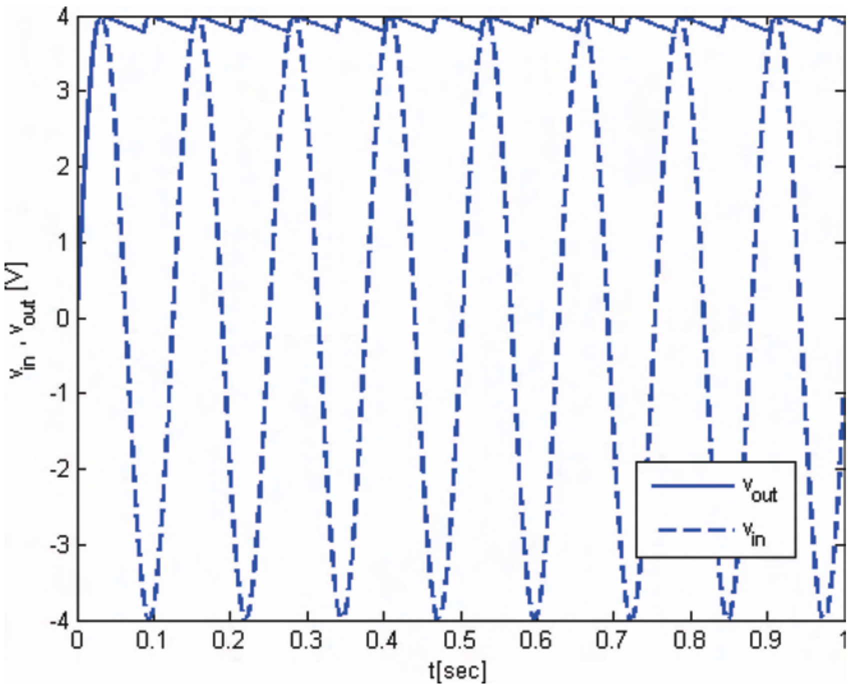

Fig. 3.6 Simulation results of the rectifier model.

solution is to use an *Algebraic Constraint* block that can be found in the *Math Operations* SIMULINK library. This block has an input called $f(z)$ and an output called z. The simulator computes z such that $f(z)$ is equal to zero (for index 1 differential algebraic systems).

3.1.4 Discussion

The MATLAB toolbox with SIMULINK and STATEFLOW provides excellent modeling and simulation capabilities for control and data-flow applications mixing continuous- and discrete-time domains. SIMULINK interfaces very well with the MATLAB environment allowing the use of powerful visualization functions for plotting graphs and, more generally, for the post-elaboration of simulation results. The SIMULINK library is very rich making the language very expressive. The expressiveness is even enhanced by the possibility of calling MATLAB functions and compiled C code.

However, often there is a need to subject the models (developed in SIMULINK) to a more complex, rigorous, and domain-specific analysis. In fact, we have seen in Section 3.1.2 that the behavior of the system is sensitive to the inner working of the simulation engines. Consequently, fully understanding what takes place inside the tools would be important to prevent unpleasant surprises. On the other hand, in most cases users ignore these details and may end up with an erroneous result without realizing it. Indeed, the lack of formal semantics of the models used inside this very successful tool set has been considered a serious drawback in academic circles[3] thus motivating an intense activity in formalizing the semantics of hybrid systems and a flurry of activities aimed at providing translation to and from SIMULINK/STATEFLOW.

A strong need has been expressed for *automatic semantic translators* that can interface with and translate the SIMULINK/STATEFLOW models into the models of different analysis and synthesis tools. In [45] Caspi *et al.* discuss a method for translating a discrete-time subset of SIMULINK models into LUSTRE programs.[4] The proposed method

[3] Some authors dispute the fact that "SIMULINK has no semantics" by arguing instead that SIMULINK has a multitude semantics (depending on user-configurable options) which, however, are informally and sometimes partially documented [45].

[4] While doing so, they also attempt to formalize the typing and timing mechanisms of such discrete-time subset of SIMULINK.

consists of three steps (type inference, clock inference, and hierarchical bottom-up translation) and has been implemented in a prototype tool called S2L.

3.2 Modelica

Modelica is an object-oriented language for hierarchical physical modeling [70, 162] targeting efficient simulation. One of its most important features is *non-causal modeling*. In this modeling paradigm, users do not specify the relationship between input and output signals directly (in terms of a function), but rather they define variables and the equations that they must satisfy. Modelica provides a formal type system for this modeling effort. Two commercial modeling and simulation environments for Modelica are currently available: Dymola [67] (Dynamic Modeling Laboratory) marketed by Dynasim AB and MathModelica, a simulation environment integrated into Mathematica and Microsoft Visio, marketed by MathCore Engineering.

3.2.1 Modelica Syntax

The syntax of the Modelica language is described in [139]. Readers familiar with object-oriented programming will find some similarities with Java and C++. However, there are also fundamental differences since Modelica is oriented to mathematical programming. This section describes the syntactic statements of the language and gives some intuition on how they can be used in the context of hybrid systems. This, of course, is not a complete reference but only a selection of the basic constructs of the language. A complete reference can be found in [139]. The book by Tiller [162] is an introduction to the language and provides also the necessary background to develop Modelica models for various physical systems.

Modelica is a typed language. It provides some primitive types like Integer, String, Boolean and Real. As in C++ and Java, it is possible to build more complicated data types by defining classes. There are many types of classes: records, types, connectors, models, blocks, packages and functions. Classes, as well as models, have fields (variables they act on) and methods.[5] In MODELICA, class methods are represented by equation and algorithm sections. An equation is syntactically defined as <expression = expression> and an equation section may contain a set of equations. The syntax supports the ability to describe a model as a set of equations on variables (non-causal modeling), as opposed to a method of computing output values by operating on input values. In non-causal modeling there is no distinction between input and output variables; instead, variables are involved in equations that must be satisfied. The algorithm sections are simply sequential blocks of statements and are closer to JAVA or C++ programming from a syntactic and semantic viewpoints. MODELICA also allows the users to specify causal models by defining *functions*. A function is a special class that can have inputs, outputs, and an algorithm section which specifies the model behavior.

Before going into the details of variable declaration, it is important to introduce the notion of *variability* of variables. A variable can be continuous-time, discrete-time, a parameter or a constant depending on the modifier used in its instantiation. The MODELICA variability modifiers are discrete, parameter and constant (if no modifier is specified then the variable is assumed to be continuous). The meaning is self-explanatory; the formal semantics is given in Section 3.2.2.

MODELICA also defines a connect operator that takes two variable references as parameters. Connections are like other equations. In fact, connect statements are translated into particular equations that involve the required variables. Variables must be of the same type (either continuous-time or discrete-time). The connect statement is a convenient shortcut for the users who could write their own set of equations to relate variables that are "connected".

MODELICA is a typed system. Users of the language can extend the predefined type set by defining new, and more complex, types. The MODELICA syntax supports the following classes:[6]

- **record**: it is just an aggregation of types without any method definition. In particular, no equations are allowed in the

[5] C++ or JAVA programmers are used to this terminology, where methods are functions that are part of a class definition.
[6] Some of the constructs mentioned below are explained in Section 3.2.2

definition or in any of its components, and they may not be used in connections. A record is a heterogeneous set of typed fields.

- **type**: it may only be an extension to the predefined types, records, or array of type. It is like a typedef in C++.
- **connector**: it is a special type for variables that are involved in a connection equation. Connectors are specifically used to connect models. No equations are allowed in their definition or in any of their components.
- **model**: it describes the behavior of a physical system by means of equations. It may not be used in connections.
- **block**: it describes an input-output relation. It has fixed causality. Each component of an interface must either have causality equal to input or output. It can not be used in connections.
- **package**: it may only contain declarations of classes and constants.
- **function**: it has the same restrictions as for blocks. Additional restrictions are: no equations, at most one algorithm section. Calling a function requires either an algorithm section or an external function interface which is a way of invoking a function described in a different language (for instance C). A function can not contain calls to the MODELICA built-in operators der, initial, terminal, sample, pre, edge, change, reinit, delay, and cardinality whose meaning is explained in Section 3.2.2.

Inheritance is allowed through the keyword extends like in JAVA. A class can extend another class thereby inheriting its parent class fields, equations, and algorithms. A class can be defined as partial, i.e., it cannot be instantiated directly but it has to be extended first. The MODELICA language provides control statements and loops. There are two basic control statements (if and when) and two loop statements (while and for).

```
if expression then
  equation/algorithm
else
```

```
    equation/algorithm
  end if
```

For instance, an expression can check the values of a continuous variable. Depending on the result of the Boolean expression, a different set of equations is chosen. It is not possible to mix equations and algorithms. If one branch has a model described by equations, so has to have the other branch. Also the number of equations has to match. The syntax of the for statement is as follows:

```
for IDENT in expression loop
  { equation/algorithm; }
end for
```

IDENT is a valid MODELICA identifier. A for loop can be used to generate a vector of equations, for instance. It is not possible to mix equations and algorithms. The while statement syntax is as follows:

```
while expression loop
{ equation/algorithm; }
end while
```

A while loop has the same meaning as in many programming languages. The body of the while statement is active as long as the expression evaluates to true. Finally, the when statement has the form:

```
when expression then
  { equation/algorithm; }
end when

when expression then
  { equation/algorithm; }
else when expression then
  { equation/algorithm; }
end when
```

The body of a when statement is active when the expression changes from false to true. Real variables assigned in a when clause must be discrete time. Also, equations in a when clause must be of the form $v = \mathit{expression}$, where v is a variable. Expressions use relation operators like $\leq$, $\geq$, $==$, $\ldots$ on continuous time variables, but can be any other valid expression whose result is a Boolean.

3.2.2 MODELICA **Semantics**

The MODELICA language distinguishes between discrete-time and continuous-time variables. Continuous-time variables are the only ones that can have a non-zero derivative. MODELICA has a predefined operator der(v) that indicates the time derivative of the continuous variable v. When v is a discrete time variable (specified by using the discrete modifier at instantiation time) the derivative operator should not be used even if we can informally say that its derivative is always zero and changes only at *event instants* (see below). Parameter and constant variables remain constant during transient analysis.

The second distinction to point out is between the algorithm and the equation sections. Both are used to describe the behavior of a model. An equation section contains a set of equations that must be satisfied. Equations are all concurrent and the order in which they are written is immaterial. Furthermore, an equation does not distinguish between input and output variables. For instance, an equation could be $i_1(t) + i_2(t) = 0$ which does not specify if i_1 is used to compute i_2 or vice-versa. The value of i_1 and i_2, at a specific time t_0, is set in such a way that all the equations of the model are satisfied. An algorithm section is a block of sequential statements. Here, order matters. In an algorithm section, the user should use the assignment operator $:=$ instead of the equality operator $=$. Only one variable reference can be used as left operand. The value of the variable to the left of the assignment operator is computed using the values of the variables to the right of it.

Causal models in MODELICA are described using functions. A function is a particular class that has input and output variables. A function has exactly one algorithm section that specifies the input/output behavior of the function. Non-causal models are described by means of equation sections defined in classes or models. Statements like if then else and for are quite intuitive. In the case of if clauses in equation sections, if the switching condition contains also variables that are not constants or parameters then the else branch cannot be omitted, otherwise the behavior will not be defined when a false expression is evaluated.

The when clause deserves particular attention. When the switching expression (see Section 3.2.1) evaluates to true the body of the when clause is active. The switching expression is considered a discrete-time predicate. If the body of the when clause is not active, all the variables assigned in the body should be held constant to their values at the last event instant. Hence, if the when clause is in an equation section, each equality operator must have only one component instance on the left-hand side (otherwise it is not clear which variable should be held). Such component instance is the one whose value is held while the switching expression evaluates to false. This condition can be checked by a syntax checker.

Finally, a connect statement is an alternative way of expressing certain equations. A connect statement can generate two kinds of equations depending on the nature of the variables that are passed as arguments. In the first case, the variables $v_1, \dots, v_n$ are declared *flows* at instantiation time (using the flow modifier) and the connection generates the equation $v_1 + \dots + v_n = 0$. Otherwise, the connection generates the equation $v_1 = ... = v_n$. Note that the term "flow" here should not be confused with the same term used to indicate a continuous evolution as opposed to a discrete jump (see, e.g., Section 3.5).

Equivalent Mathematical Description of a Modelica **Program.** A program written in the Modelica language can be interpreted by defining a one-to-one mapping between the program and a system of Differential Algebraic Equations (DAE). The first step is to translate a hierarchical Modelica model into a flat set of Modelica statements, consisting of the set of equation and algorithm sections of all the used components. The resulting system of equations looks like the following:

$$c := f_c(rel(v)) \tag{3.1}$$

$$m := f_m(v, c) \tag{3.2}$$

$$0 := f_x(v, c) \tag{3.3}$$

where $v := [\dot{x}; x; y; t; m; pre(m); p]$. Here, p is the set of parameters and constant variables, m is the set of discrete event variables, $pre(m)$ is the value of discrete events variables immediately before the current event occurred, x and y are continuous variables, $rel(v)$ is the set of relations on variables in v and c is the set of expressions in if statements (including expressions coming from the conversion of when statements into if). The variables x and y are distinguished because x variables appear differentiated while y variables do not. A DAE solver will iterate in the following way:

- Equation 3.3 is solved by assuming c and m constants, meaning that the system of equations is a continuous system of continuous variables;
- during integration of Equation 3.3, the conditions in Equation 3.1 are monitored. If a condition changes its status, an event is triggered at that specific time and the integration is halted.
- at the event instant, Equation 3.2 is a mixed set of algebraic equations which is solved for the Real, Boolean and Integer unknowns;
- after the event is processed, the integration is restarted with Equation 3.3.

3.2.3 Examples

We first describe the full wave rectifier example, which shows the usefulness of object orientation and non-causal modeling. The variables are currents through and voltages across each component, whose types are defined as follows:

```
type Voltage = Real;
type Current = Real;
```

Each component in a circuit has pins to connect to other components. A pin is characterized by a voltage (with respect to a reference voltage) and an input current. A pin is defined as follows:

```
connector Pin
  Voltage v;
  flow Current i;
end Pin;
```

The connector keyword is used to specify that pins are used in connection statements. The flow keyword is used to declare that the variable i is a flow, i.e., the sum of all *Current* fields of *Pins* in a connection must be equal to zero. A generic two-pin component can be described in the following way [71]:

```
partial class TwoPin
   Pin p, n;
   Voltage v;
   Current i;
   equation
      v = p.v - n.v;
      0 = p.i + n.i;
      i = p.i;
end TwoPin;
```

This class defines a positive and a negative pin. Kirchoff's equations for voltage and current are declared in the equation section. This class is partial and we extend it to specify two pins components like resistors and capacitors. A capacitor for instance can be described as follows:

```
class Capacitor
   extends TwoPin;
   parameter Real C(unit="F") "Capacitance";
   equation
      C * der(v) = i;
end Capacitor;
```

In the equation section, we need only declare the component constituent equation since the other equations are inherited from a two-pin component. A parameter is used for the value of capacitance. A diode is modeled as a component with two regions of operation: reverse bias for $v < 0$ and forward bias for $v \geq 0$:

```
class Diode
   extends TwoPin;
   equation
      if v ≥ 0 then i = v / 0.1;
      else i = -1e-15;
      end if;
end Diode;
```

In the forward-bias region, the diode is a resistor with a very small resistance while in reverse bias it is basically an open circuit (only a small reverse current flows through it). Each component can be instantiated and interconnected with others to build a netlist as in the following example:

```
class circuit
  Resistor R1(R = 10); Capacitor C1(C = 0.01);
  Vsin DCp(VA = 5); Vsin DCn(VA = 5);
  Diode d1; Diode d2;
  Ground G;
  equation
    connect( DCp.p, d1.p ); connect( d1.n, R1.p );
    connect( d1.n, C1.p ); connect( DCp.n, G.gpin );
    connect( DCn.p, G.gpin ); connect( DCn.n, d2.p );
    connect( d2.n, R1.p ); connect( C1.n, G.gpin );
    connect( R1.n, G.gpin );
end circuit;
```

where `Vsin` is the sinusoidal voltage source and `Ground` is a component that is used to fix the voltage of a node to $0V$. Figure 3.7 shows the simulation result for the two different types of load. The waveforms were obtained by simulating the Modelica models with Dymola. Dymola is able to solve the algebraic loop by performing a symbolic manipulation.

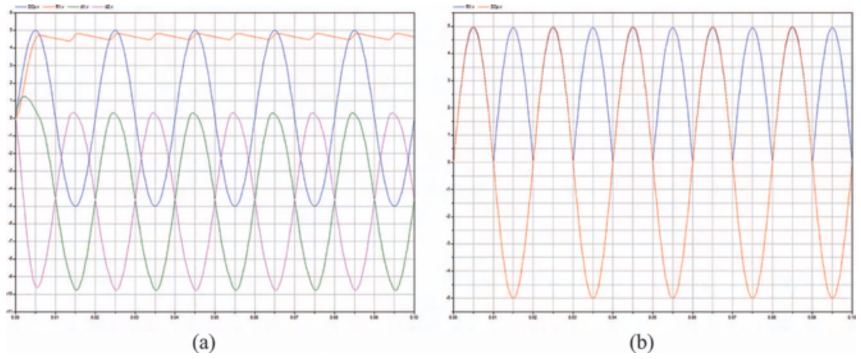

Fig. 3.7 Dymola simulation results of the Modelica rectifier example: (a) for an RC load and (b) for a pure resistive load.

The Three-Mass Example. A moving mass is a MODELICA class that defines a mass moving in a bi-dimensional space with vertical and horizontal accelerations equal to *ax* and *ay* respectively.

```
class MovingMass
  parameter Real x0, y0, vx0, vy0, ax0, ay0;
  Real x, y, vx, vy, ax, ay;
equation
  der( x ) = vx; der( vx ) = ax; der( y ) = vy; der( vy ) = ay;
algorithm
  when initial( ) then
    reinit( x, x0 ); reinit( y, y0 ); reinit( vx, vx0 );
    reinit( vy, vy0 ); reinit( ax, ax0 ); reinit( ay, ay0 );
  end when;
end MovingMass;
```

The equations are self-explicative. When the simulation starts, the call to `initial()` generates an event that executes the `when` clause. The `reinit` statements set each variable to its initial value that is passed as parameter. The system of three masses is a MODELICA class that instantiates three moving masses and defines guards conditions and resets maps. The model is described as follows:

```
class ThreeMasses
  parameter Real m1 "Mass1", m2 "Mass2", m3 "Mass3";
  parameter Real h "Height", L "Lenght", e "Restitution";
  MovingMass mass1(x0=0.0,y0=h,vx0=3.0,vy0=0.0,ax0=0.0,
    ay0=0.0);
  MovingMass mass2(x0=6.5,y0=h,vx0=0.0,vy0=0.0,ax0=0.0,
    ay0=0.0);
  MovingMass mass3(x0=7.0,y0=h,vx0=0.0,vy0=0.0,ax0=0.0,
    ay0=0.0);
equation
  if ( (mass1.x >= L) and (mass1.vx > 0 ) ) then
    mass1.ay = -9.81; mass1.ax = 0.0;
  else
    mass1.ay = 0.0; mass1.ax = 0.0;
  end if;
  if ( (mass2.x >= L) and (mass2.vx > 0 ) ) then
    mass2.ay = -9.81; mass2.ax = 0.0;
  else
    mass2.ay = 0.0; mass2.ax = 0.0;
  end if;
  if ( (mass3.x >= L) and (mass3.vx > 0 ) ) then
    mass3.ay = -9.81; mass3.ax = 0.0;
  else
    mass3.ay = 0.0; mass3.ax = 0.0;
  end if;
  when ( (mass1.y <= 0) and (mass1.vy < 0) ) then
    reinit(mass1.vx,e*pre(mass1.vx));reinit(mass1.vy,-
    e*pre(mass1.vy));
  end when;
  when ( (mass2.y <= 0) and (mass2.vy < 0) ) then
    reinit(mass2.vx,e*pre(mass2.vx));reinit(mass2.vy,-
    e*pre(mass2.vy));
  end when;
  when ( (mass3.y <= 0) and (mass3.vy < 0) ) then
    reinit(mass3.vx,e*pre(mass3.vx));reinit(mass3.vy,-
    e*pre(mass3.vy));
  end when;
algorithm
  when ( (mass1.x >= mass2.x) and ( mass1.vx >= mass2.vx) )
  then
    reinit(mass1.vx, pre(mass1.vx) * (m1 - e * m2) / (m1 + m2) +
    pre(mass2.vx) * m2 * (1 + e) / (m1 + m2));
    reinit(mass2.vx, pre(mass1.vx) * (1 + e) * m1 / (m1 + m2) +
    pre(mass2.vx) * (m2 - e * m1) / (m1 + m2));
  elsewhen ( (mass2.x >= mass3.x) and ( mass2.vx >= mass3.vx))
  then
    reinit(mass2.vx, pre(mass2.vx) * (m2 - e * m3) / (m2 + m3) +
    pre(mass3.vx) * m3 * (1 + e) / (m2 + m3));
    reinit(mass3.vx, pre(mass2.vx) * (1 + e) * m2 / (m2 + m3) +
    pre(mass3.vx) * (m3 - e * m2) / (m2 + m3));
  end when;
end ThreeMasses;
class ThreeMassSystem
  ThreeMasses tms(m1 = 1.0, m2 = 1.0, m3 = 1.0, h = 3, L = 7,
  e = 0.9);
end ThreeMassSystem;
```

The code shows two sections: one equation and one algorithm. The semantics is very different in the two cases: statements in an algorithm section are sequential while equations are constraints that must be satisfied concurrently. The if statements define regions where the masses are subject to vertical acceleration. Note that, in order to have the same number of equations independently of whether the condition holds or not, an if statement in an equation section must always have an else branch. A set of when statements takes care of resetting the vertical velocity when a mass hits the ground. The order in which velocities are re-initialized after they hit the ground is immaterial.

We describe the collisions in the algorithm section. The when-elsewhen statement imposes a priority between the collision of m_1 with m_2 and the collision of m_2 with m_3. In particular, if $x_{2,0} = x_{3,0}$ then the two collisions have the same time stamp and when the algorithm section runs, only the first branch of the when statement is executed while the second event is basically lost. The DYMOLA compiler warns the user that some variables are re-initialized in different parts of the source code which could lead to non-deterministic behaviors unless the events that are involved in the re-initialization are mutually exclusive.

Figure 3.8 shows the simulation result. The collision of m_2 and m_3 and the falling events are exactly located at the same point in time as it can be deduced by the fact that the two masses bounce together at the same time (see the larger inset at the left of Figure 3.8). Two effects can be noted. First, the simulation is non-Zeno. This is because MODELICA always introduces a delay when executing a transition. Second, the bouncing balls eventually fall below the floor, as indicated in the inset at the right of Figure 3.8 at the end of the simulation. This artifact, that we have already seen in SIMULINK in Section 3.1.3, is again due to the simulation strategy. However, unlike SIMULINK, the bouncing event in this case is not lost due to the simultaneity of two events. Instead, the ball bounces, but the following integration step is too large, in fact large

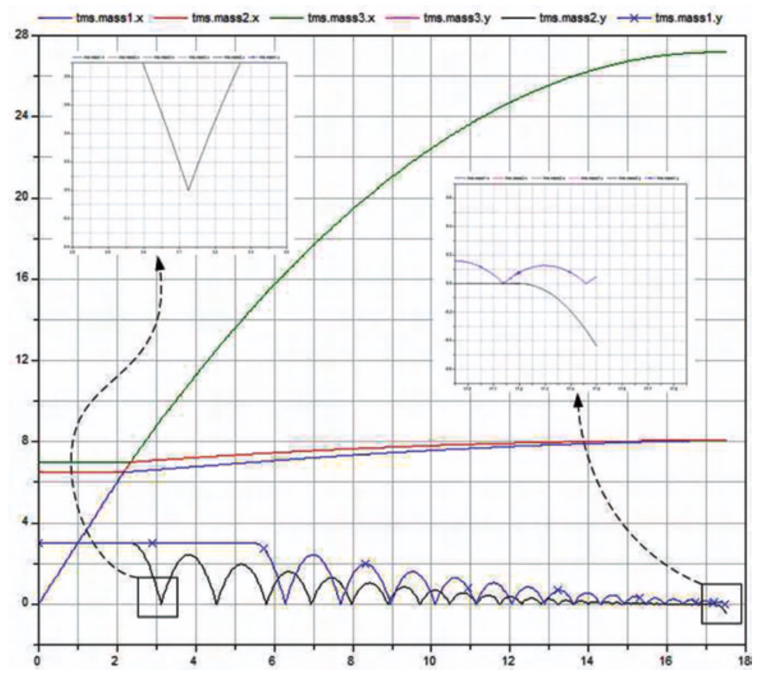

Fig. 3.8 Modelica simulation result for three-mass system example.

enough that the ball at the next iteration has already reached its highest point and fallen again below the floor level. Because the sign of the vertical position and of the vertical velocity remain negative between the two integration steps, a new bouncing event is not generated, and the ball keeps falling below the floor level.

3.2.4 Discussion

Modelica is an object-oriented language for mathematical programming. Object orientation is well understood in the software community and is certainly a well accepted programming paradigm. The language is very clean. There are important features that make building models easy. First of all, non-causal modeling allows designers to write model equations directly into the language syntax without any change. Designers do not have to explicitly define dependent and independent variables. This saves the potential effort of solving equations or making different models depending on which quantities are computed and which are used to compute others.

Object orientation helps write reusable models. Inheritance makes it possible to define a basic set of equations that are common to many dynamical systems and then specialize a model depending on the real application. In modeling a physical system, it is often important to distinguish quantities as *"through"* and *"across"*. MODELICA provides a special keyword to declare their type. Then, connections are automatically translated into the correct equation (zero-sum or equality) according to the type of variables involved. MODELICA doesn't specify the semantics of algebraic loops. This is left to the particular simulation tool, which could simply reject a program that contains them. For instance, a simple system $x = x$ cannot be simulated in DYMOLA, which reports a cyclic dependency on x, while the system $x^2 + x = x^2 + x$ can be simulated and gives the result $x = 0.0$. This is because the first is treated symbolically with algebraic manipulations, while the second, which is more complex, is solved using numerical techniques.

All these features make a MODELICA model very compact. Modeling hybrid systems in MODELICA, however, is not a trivial task. Guard conditions and reset maps can be specified in equation sections or algorithm sections and they have very different meanings. When described in equation sections, events cannot be sequentially scheduled because elsewhen are not allowed. When described in algorithm sections, simultaneous events could be lost.

When such languages are used to describe hybrid systems, the discrete state at time t is usually not explicit but it is represented by the sequence of events that happened until t. Continuous state and events are defined by a set of non-causal equations that model the physical system. These two peculiarities of the MODELICA modeling paradigm make debugging less intuitive than other tools like HYVISUAL where states and transitions are explicit and where models are causal.

3.3 HyVisual

The Hybrid System Visual Modeler (HYVISUAL) is a block-diagram editor and simulator for continuous-time dynamical systems and hybrid systems [103]. HyVisual is built on top of Ptolemy [66, 127], a framework that supports the construction of domain specific tools, and can be freely downloaded from http://ptolemy.eecs.berkeley.edu.

3.3.1 HyVisual Syntax

Like any Ptolemy model, a HyVisual model is specified graphically starting from a set of library actors. An actor is a block with typed ports and parameters. Output ports can be connected to input ports by means of relations. Types are organized in a partial order, where $t_1 \geq t_2$ if a variable of type t_1 can be converted into t_2 without loss of information. The type of an output port must be greater than or equal to the type of the input port it is connected to. While the actor library is rich enough to model most practical systems, users have the option to build new actors and redefine relations. A *composite actor* encapsulates a subsystem as an interconnection of other actors, thereby representing a level of the hierarchy. Hierarchy can also be expressed in terms of a *modal model*, which represents an actor that has modes of operation. A modal model is captured as a finite state machine that can be specified by drawing bubbles (states) and connecting them through arcs (transitions). Each bubble can be refined into a continuous time system representing a dynamical system or into another finite state machine.

A hybrid system can be described in HyVisual as follows. A modal model is instantiated and its ports are configured. The finite state machine that describes its mode of operations is represented as a graph. Each state has a name and each transition is characterized by the following elements:

guard expression: a Boolean expression involving inputs and outputs of the modal model as well as state variables;
output actions: an assignment of values to the output ports;
set actions: an assignment of values to the state variables;
reset: a Boolean value (either zero or one);
preemptive: a Boolean value (either zero or one);
non-deterministic: a Boolean value (either zero or one).

Each state can be refined into a dynamical system or into another finite state machine. The user describes a dynamical system by using actors from the built-in libraries. These include actors for standard computation (like addition, multiplication, etc.), as well as actors to model continuous dynamics (the *dynamics* library) like Integrator, LaplaceTransferFunction, LinearStateSpace, DifferentialSystem. When a modal model is created, its ports are propagated to the state machine diagram and to all its refinements.

A HyVisual model is saved in XML format. The XML file is a text file describing the actors used in the model, their ports and parameter configuration, and their graphical properties (shape and position).

3.3.2 HyVisual Semantics

A complete and clear explanation of the HyVisual semantics is given in [128]. Here we briefly summarize the main concepts.

In HyVisual, a *continuously evolving signal* is a function

$$x : T \times \mathbb{N} \to V$$

where $T \subset \mathbb{R}$ is a connected subset representing the time line, $\mathbb{N}$ is the set of non-negative integers representing an index within a time stamp, and V is the set of values that the signal can take on. For a fixed time t, the value of a signal depends on the index, which is used to model simultaneous events. In order to avoid chattering Zeno conditions, it is required that $\exists m \in \mathbb{N}$ such that $\forall n > m$, $x(t,n) = x(t,m)$. If the system is non-chattering Zeno, then the least m satisfying the condition above is called the *final index*. The value $x(t,m)$ is called the *final value* of x at t and the value $x(t,0)$ is called the *initial value* at time t. Accordingly, the *initial value function* $x_i : T \to V$ and and the *final value function* $x_f : T \to V$ are defined as

$$\forall t \in T, \quad x_i(t) = x(t,0) \quad \text{and} \quad x_f(t) = x(t,m)$$

where m is the final index. This representation is useful to express functions that are piecewise continuous, that is functions that are continuous except for a discrete subset of the timeline. A signal x is *piecewise continuous* if

(1) the initial value function x_i is left continuous;
(2) the final value function x_f is right continuous;
(3) x has only one value at all $t \in T \setminus D$, where D is a discrete subset of T.

The solution to the dynamical system

$$\dot{x}(t) = g(x(t),t), \quad x(t_0) = x_0 \tag{3.4}$$

can then be expressed as a piecewise continuous signal. This can be further discretized by letting $D \subset T$ be a discrete set that includes the times at which signals have more than one value and D' a superset that includes D. A *discrete trace* of the hybrid system is the set

$$\{x(t,n) | t \in D' \wedge n \in \mathbb{N}\} \tag{3.5}$$

To be a valid trace, it is required that, for each interval between times in D', Equation 3.4 have a unique and continuous solution, and that the endpoints of the solution in the interval be in the trace.

To obtain a discrete trace one can proceed as follows.

Init: $t^* = t_0$, $x(t^*,0) = x_0$;
Discrete phase: execute the model until $x_f(t^*)$ is computed;
Continuous phase: compute t_1 such that g is continuous and locally Lipschitz on $[t^*,t_1)$. Solve Equation 3.4 on the interval $[t^*,t_1)$ with initial condition $x_0 = x_f(t^*)$;
Iterate: Set $t^* = t_1$ and iterate from the discrete phase with $x(t^*,0)$ equal to the value of x at t_1 computed in the previous step.

Two issues remain open: how to compute t_1 and how to execute the model to compute $x_f(t^*)$. The first issue reduces to a proper selection of the step size while the second reduces to the definition of the discrete phase semantics.

To determine the step size, HyVisual implements both event detection as well as backtracking. In particular, backtracking is implemented by providing each actor with two functions:

$$f : V_d^n \times T \times \Sigma \to V_d^m \tag{3.6}$$

$$g : V_d^n \times T \times \Sigma \to \Sigma \tag{3.7}$$

where n is the number of input ports, m the number of output ports, V_d is the set of all possible values (including the absence of a signal ϵ which is fundamental for representing discrete signals), T is the time line and Σ is the state space of an actor. The function f is the output function and g is the state update function. In HyVisual, each actor can reject the current step size decided by the simulator, in which case a new step size must be decided. The simulator calls the state update function only after all actors have accepted the current step size.

The second issue is how to compute $x_f(t^*)$. HyVisual has a fixed point semantics to compute the values of signals and state. For an actor, let the input be $x : T \times \mathbb{N} \to V_d^n$, the output be $y : T \times \mathbb{N} \to V_d^m$ and the state be given by the function $\sigma : T \times \mathbb{N} \to \Sigma$. At time $t \in T$, execution proceeds as follows:

$$\begin{aligned}
y(t,0) &= f(x(t,0), t, \sigma(t,0)) \\
\sigma(t,1) &= g(x(t,0), t, \sigma(t,0)) \\
y(t,1) &= f(x(t,1), t, \sigma(t,1)) \\
\sigma(t,2) &= g(x(t,1), t, \sigma(t,1)) \\
&\dots
\end{aligned}$$

When (and if) all actors in the model have reached a point where their state no longer changes, then the final values have been reached for all signals and the execution at time t is complete.

3.3.3 Examples

The HyVisual model of the three-mass system is shown in Figure 3.9. Each state of the state machine is refined into a continuous time system that describes the dynamics of a point mass moving with a constant acceleration. The accelerations are integrated to obtain the velocities and the velocities are integrated to obtain the positions. Both horizontal and vertical positions are used to generate threshold events: the horizontal positions are monitored to check when a point mass falls off the table and the vertical position is monitored to check when a point mass hits the ground.

The initial state is named Init. From the initial state, the model makes a spontaneous transition to a state where m_1 starts moving

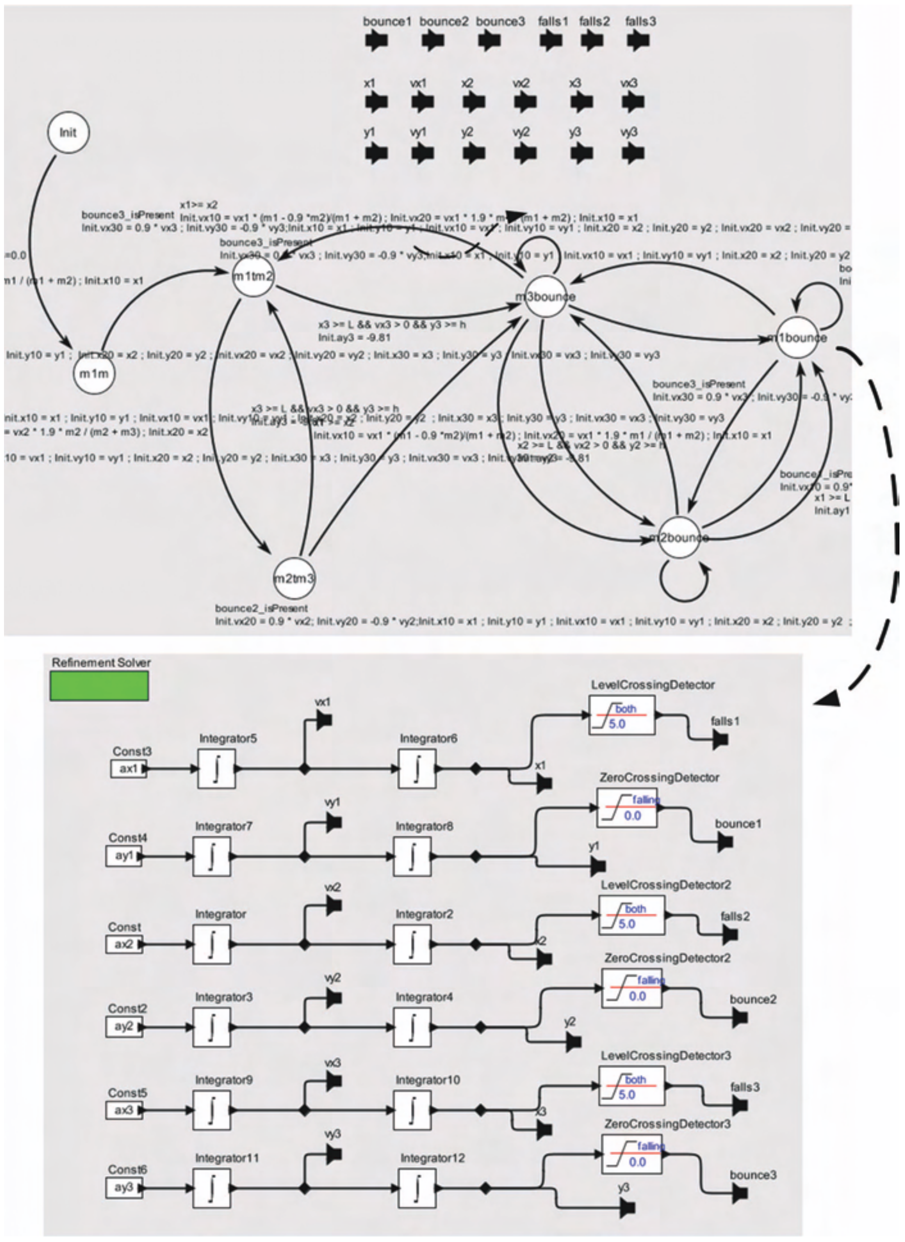

Fig. 3.9 HyVisual model of the three-mass system.

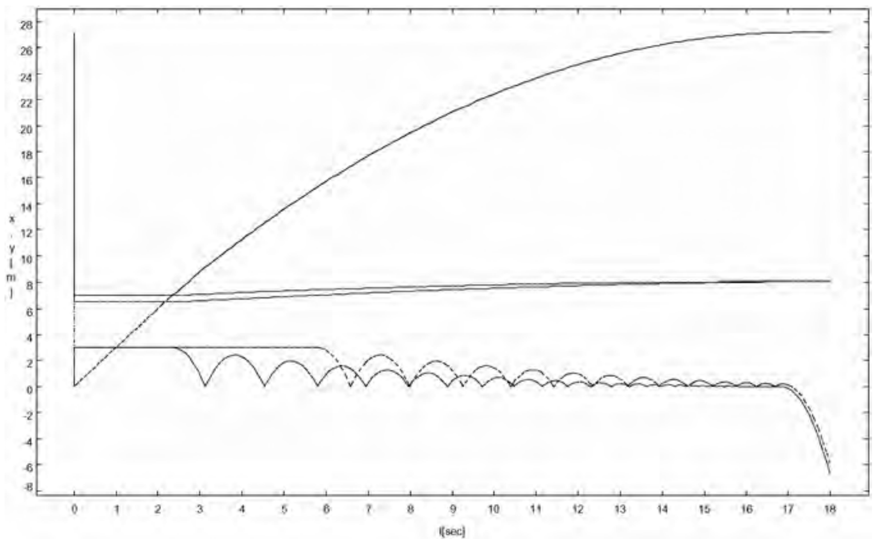

Fig. 3.10 HyVisual simulation result for the three-mass system with $x_{20} = 4.95$, $x_{30} = 4.98$, $L = 5$ and $h = 7$.

with initial velocity v_{10}. The state machines implements the one in Figure 2.2.

The simulation results are shown in Figure 3.10 where all three masses eventually bounce on the ground. In this simulation $L = 7$, $x_{2,0} = 6.5$, $x_{3,0} = 7$ and $v_{1,0} = 3$ while $y_i = 3$ for all three masses. When m_2 touches m_3, HyVisual correctly simulates the collision and the falling events of m_2 and m_3 that occur at the same time, but with different indices. When m_2 and m_3 touch the ground, multiple output transitions are enabled from state m_2*bounce* and the simulator reports an error saying that there are multiple transitions enabled but not all of them are marked non-deterministic. In the lastest version of HyVisual, each transition has, in fact, a flag non-deterministic that can be marked in situations where multiple transitions could be enabled at the same time. After this small change has been made to the model, the simulation can be successfully completed. At the end of the simulation, as shown in Figure 3.10, the three balls fall below the floor level. This effect is again due to the choice of the duration of the integration step, as already explained for Modelica in Section 3.2.3.

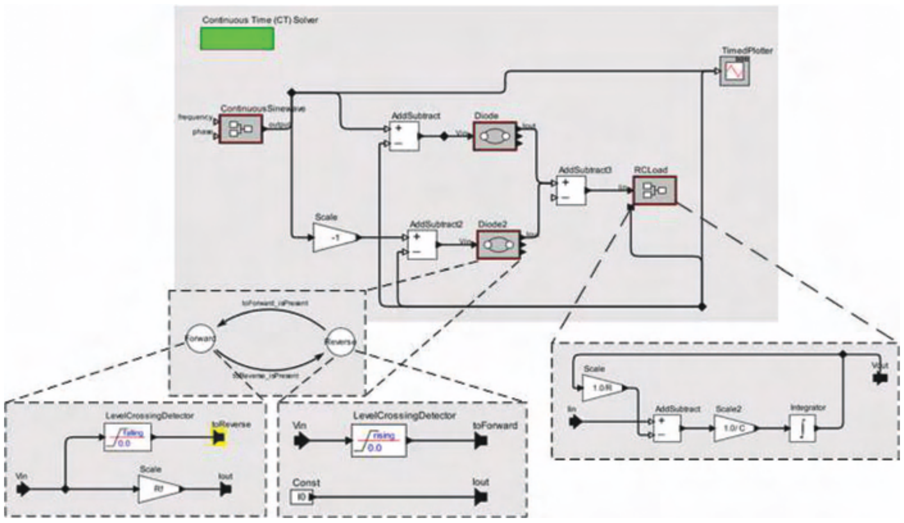

Fig. 3.11 HyVisual model of the full wave rectifier.

The Full Wave Rectifier Example. The HyVisual model of the full wave rectifier is shown in Figure 3.11. A diode is modeled as a hybrid system with two states: Forward and Reverse. The Forward state is refined into a linear continuous time system whose output current is proportional to the input voltage by a constant R_f. The Reverse state is refined into a system whose output current is constant and equal to I_0. The RC load model implements the two equations:

$$V_{out}(t) = \frac{1}{C}\int_{t_0}^{t} I_C(t)dt + V_{out}(t_0)$$

$$I_C(t) = I_{in}(t) - I_R(t) = I_{in}(t) - \frac{V_{out}}{R}$$

The simulation result is shown in Figure 3.12.

When the load is replaced by a simple resistor with $V_{out} = RI_{in}$, HyVisual reports an error for the presence of an algebraic loop.

3.3.4 Discussion

HyVisual is a graphical environment for modeling hybrid systems. Graphical representations have the advantage of being intuitive and

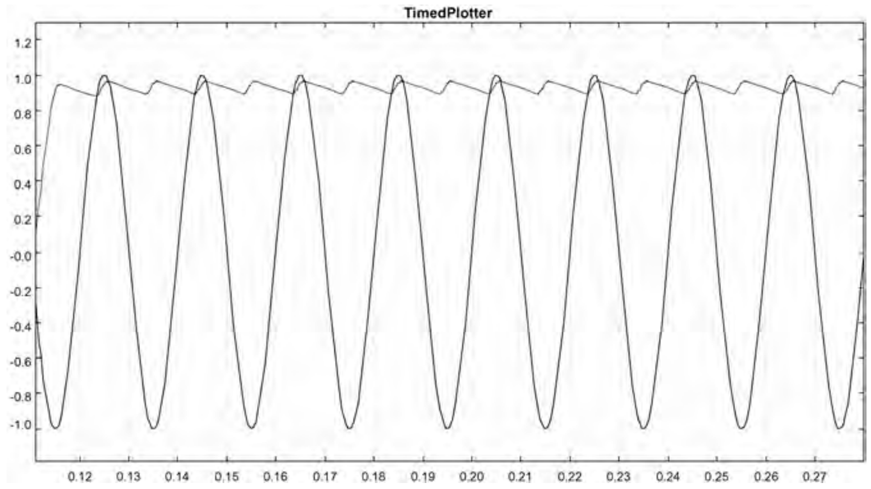

Fig. 3.12 HyVisual simulation result for the full wave rectifier.

easy to use. There is a rich library of components making the language expressive enough to model hybrid systems. Type checking and inference are desirable features in designing large systems, because they help the users focus on the structure of the system. The implementation of hierarchy in HyVisual is very clean and allows the users to encapsulate subsystems into larger blocks. Furthermore, state machines can be hierarchical in the sense that a state can be refined in other state machines. This feature of grouping states is very important when dealing with systems having a large state-space. It is important to stress that state and transition refinements can be arbitrary Ptolemy models. This is different from Simulink, where the states of Stateflow are atomic objects, and the control they exercise over a continuous-time model is via continuous-time signals with discontinuities rather than via mode transitions. Finally, HyVisual stores the entire design in an XML format, which can be easily converted into other XML-based formats using XSL transformations.

HyVisual is based on a solid operational semantics that is missing in Simulink/Stateflow or even in Modelica. HyVisual formally defines the trace that results from the execution of a model without assuming any particular solver. Modelica and Simulink/Stateflow both rely on the particular simulator that completes the definition of their operational semantics.

Compared to MODELICA, HYVISUAL can only express causal models and is based on a graphical syntax that is not always easy to manipulate. When the model becomes complicated, the number of connections can grow quadratically with the number of blocks making the diagrams difficult to edit.

3.4 Scicos

SCICOS (SCILAB Connected Object Simulator) is a SCILAB package for modeling and simulation of dynamical systems including both continuous and discrete time subsystems [143]. SCILAB (Scientific Laboratory) is a scientific software package for numerical computations that provides a powerful open computing environment for engineering and scientific applications [77]. Since 1990 SCILAB has been developed by researchers from INRIA and ENPC. In May 2003 the newly created SCILAB Consortium took over maintenance and development of SCILAB. Since 1994 SCILAB has been distributed freely via the Internet and used in educational and industrial environments around the world. SCICOS has been developed also at INRIA and is freely available for download at http://www.scicos.org. SCILAB can be seen as similar to MATLAB while SCICOS is similar to SIMULINK.

SCICOS users can build models of hybrid systems by composing functional blocks from a predefined library (as well as newly-defined blocks) and simulate them. This is done within a graphical editor. Additionally, users can generate executable C code implementing the functionality of some subsystem in the original hybrid system. This is limited to discrete time subsystems, i.e., subsystems that do not include continuous-time blocks. The main application of SCICOS is embedded control: continuous blocks can be used to model the physical environment while the discrete subsystems specify the functionality of the controller. After simulating and refining the design of the controller, the user can generate C code to be executed on the target hardware architecture. Finally, for the important case of distributed real-time applications, the users can rely on the SCICOS-SYNDEX interface [64] to generate and deploy executable code on multiprocessors architectures. SynDEx is a system-level CAD software for distributed real-time embedded systems designed and developed at INRIA that is freely available at "www.rocq.inria.fr/syndex".

3.4.1 Scicos **Syntax**

A system is modeled in Scicos by assembling functional components called *blocks* that interact by means of *signals*. Each signal, in turn, is characterized by an *activation time set*, which determines the intervals in which the signal can evolve and change its value. Each system operation in Scicos is associated to a block. The activation times of a signal correspond to the activation times of the block that generates it. Figure 3.13 illustrates a generic block. This can present ports associated to four different signal types: regular input, regular output, activation (event) input, activation (event) output. By convention these ports are placed respectively on the left, right, top, and bottom side of the block. The set of signals in Scicos is partitioned into two subsets: *regular* signals and *activation* signals. Regular signals are used to exchange data among blocks, while activation signals carry control information. Activation signals are also called *event signals* or *impulses*. Regular inputs are linked to regular outputs via *regular paths*, while activation inputs are linked to activation outputs via *activation paths*. Regular paths

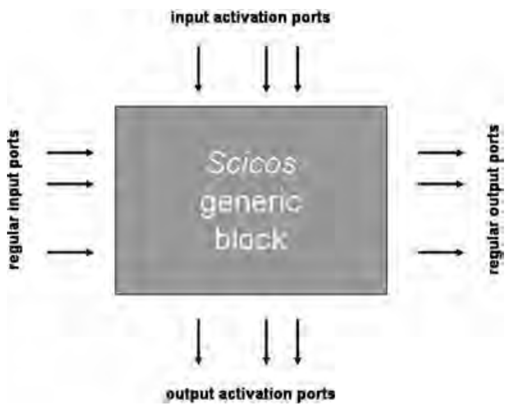

Fig. 3.13 A generic Scicos block and its I/O signals.

carry piece-wise right-continuous functions of time whereas event paths transmit timing information concerning discrete events (impulses). In particular, an event signal specifies the time when the blocks connected to the output event port generating the event signal are updated according to the internal relations of the block (see Section 3.4.2).

An activation signal causes the block to evaluate its outputs and new internal states as a function of its inputs and previous internal states. A block with no input activation port is permanently active (*time-dependent block*). The output signals inherit their activation times set from the union of the activation times of the input signals of the generating block. In turn, they can be used to drive other blocks. The signals leaving the output activation ports are activation signals generated by the block. For instance, a clock block may generate a periodic activation signal that can be connected to the input of a scope block to control the sampling of its inputs [143]. There are two general types of blocks: *basic blocks* and *super blocks*. Super blocks are obtained as the hierarchical composition of basic blocks and other super blocks. SCICOS comes with a library of more than 70 basic blocks [143]. Additionally, the users can build new basic blocks by defining an *interfacing function* and a *computational function* for each of them. The former is always a SCILAB function, while the latter can also be written in C or Fortran to achieve greater performance in the simulation. Besides defining the graphical aspect of the block, the interfacing function allows users to define the number and types of ports and to initialize the state and parameters of the block. The computational function specifies the dynamic behavior of the block through a set of tasks and is called by the SCICOS simulator that controls their execution.

3.4.2 SCICOS Semantics

A signal x in SCICOS is a pair $\{x(t),T\}$, where $x(t)$ is a function of time and T is the associated *activation time set* on which the signal x can potentially evolve and change its value [33]. The activation time set is the union of time intervals and isolated points called *events*. In fact, a generic signal in SCICOS can be the result of operating on both continuous (time intervals) and discrete (time events) signals. Outside

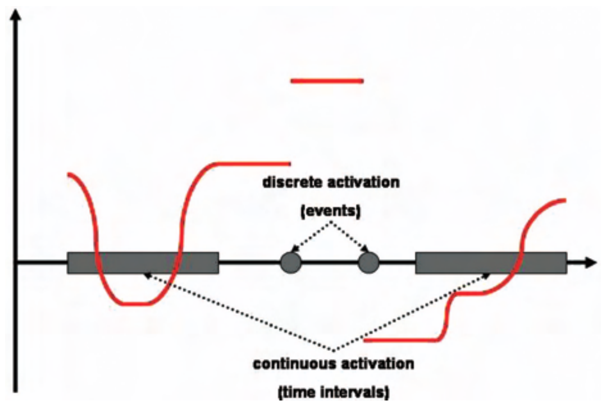

Fig. 3.14 A Scicos signal remains constant outside its activation time set.

its activation time set, a signal is constrained to remain constant as illustrated in Figure 3.14, which shows the evolution of a hybrid signal x. Activation time sets are used in Scicos in the same way as *clocks* are used in the synchronous programming language Signal [34, 82], namely as a *type checking* mechanism. For instance, two signals can be constrained to have identical time sets. In general, the various Scicos signal operators induce relations between the corresponding time sets. Given a generic binary operator f, the activation time set of the resulting signal is the union of the activation time sets of its operands, i.e.:

$$f(\ \{x_1(t), T_1\}, \{x_2(t), T_2\}\ ) = \{\ f(x_1(t), x_2(t)), (T_1 \cup T_2)\ \}$$

It is possible to reason formally on the time sets of Scicos signals as it is the case for the clocks of Signal variables.[7] Hence, Scicos users have a sound basis for tasks like design optimization and scheduling analysis.

Depending on the type of the block and the directive of the simulator, the invocation of a computational function may result in various actions like evaluation of new outputs, state update, or computation of the state derivative. There are four types of basic blocks in Scicos: *continuous*, *discrete*, *zero-crossing*, and *synchro*.

[7] Notice however that the Scicos model of computation is not the same as the Signal one.

A *continuous basic blocks (CBB)* can have both regular input (output) ports and event input (output) ports. CBBs can model more than just continuous dynamics systems. A CBB can have a continuous state x and a discrete state z. Let the vector function u denote the regular inputs and y the regular outputs. Then a CBB imposes the following relations:

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

where f and h are block specific functions, and p is a vector of constant parameters. The above relation represents two constraints that are imposed by the CBB as long as no events (impulses) arrive on its event input ports. An event input can cause a jump in the states of the CBB. Assume one or more events arrive on the CBB event ports at time t_e. Then the states jump according to the following equations:

$$\begin{aligned} x &= g_c(t_e, x(t_e^-), z(t_e^-), u(t_e^-), p, n_{evprt}) \\ y &= g_d(t_e, x(t_e^-), z(t_e^-), u(t_e^-), p, n_{evprt}) \end{aligned}$$

where g_c and g_d are block specific functions, n_{evprt} designates the ports through which the events have arrived, and $z(t_e^-)$ is the previous value of the discrete state z (which remains constant between any two successive events). Finally, CBBs can generate event signals on their event output ports. These events can only be scheduled at the arrival of an input event. If an event has arrived at time t_e, the time of each output event is generated according to

$$t_{evo} = k(t_e, z(t_e), u(t_e), p, n_{evprt})$$

for a block specific function k and where t_{evo} is a vector of time values, each entry of which corresponds to one event output port. Normally all the elements of t_{evo} are larger than t_e. If an element is less than t_e, it simply means the absence of an output event signal on the corresponding event output port. Notice that setting "$t_{evo} = t$" should be avoided because the resulting causality structure is ambiguous. Also, notice that setting "$t_{evo} = t$" does not mean that the output event is synchronized with the input event because two events can have the same time without being synchronized. The scheduled t_{evo} is recorded inside the CBB in a register that has size equal to the number of output event ports. The value in the register is used to "fire" the events at the specified time. This register can be pre-loaded at the beginning of the simulation by setting the corresponding *initial firing* in the CBB. Because the register can hold only one value per output event port, only one output event can be scheduled on each output event port at a time (both at the beginning and in the course of the simulation). In other words, by the time a new event is ready to be scheduled, the old one must have been already fired. Another interpretation is that as long as the previously scheduled event has not been fired yet, the corresponding output port is considered busy, meaning that it cannot accept a new event scheduling. If the simulator encounters such a conflict, it stops and returns the *event conflict* error message [143].

While a CBB permanently monitors its input ports and continuously updates its output ports and continuous state, a *discrete basic block (DBB)* only acts when it receives an input event, and its actions are instantaneous. DBBs can have both regular and event input and output ports, but they must have at least one event input port. DBBs can model discrete dynamical systems. A DBB can have a discrete state z but no continuous state. Upon the arrival of events at time t_e, the state and the outputs of a DBB change as follows

$$\begin{aligned} z &= f_d(t_e, z(t_e^-), u(t_e^-), p, n_{evprt}) \\ y &= g_d(t_e, z, u(t_e), p) \end{aligned}$$

where f_d and h_d are block specific functions. The regular output y remains constant between any two successive events. In fact, the output y and the state z are piece-wise constant, right-continuous functions of time. Like CBBs, DBBs can generate output events according to a specific function k and their events can be pre-scheduled via initial firing. The difference between a CBB and a DBB is that a DBB cannot have a continuous state and that its outputs remain constant between two events. Although in theory CBBs subsume DBBs, specifying a block as a DBB has performance advantages since the simulator can optimize its execution because it knows that the outputs of the block remain constant between events. Note that the regular output signal of a DBB is always piece-wise constant. Being piece-wise constant does not necessarily imply that a signal is discrete. For example, the output of an integrator (which is a CBB with a continuous state) can, in some special cases, be constant. However, signals that are piece-wise constant can be identified based solely on the basic properties of the blocks that generate them. In particular, in SCICOS, every regular output signal of a DBB is discrete and every regular output signal of a state-less time invariant CBB receiving only discrete signals on its inputs is also discrete. Thus, the discrete nature of signals in a model can be specified statically. Again, the SCICOS compiler relies on this information to optimize the performance of the SCICOS simulator.

A *zero crossing basic block (ZCBB)* has regular inputs and event outputs but no regular outputs, or event inputs. ZCBBs can generate event outputs only if at least one of their regular inputs *crosses zero* (i.e., it changes sign). In such a case, the generation of the event, and its timing, may depend on the combination of the inputs which have crossed zero and the signs of the inputs (just before the crossing occurs). The simplest example of a *Surface Crossing Basic Block* is the zcross [143]. This block generates an event if all the inputs cross zero simultaneously. Inputs of ZCBBs can start off at zero, but cannot remain equal to zero during the simulation. This is considered an ambiguous state and is declared as an error. Similarly the input of a ZCBB should not jump across zero. If it does, the crossing may or may not be detected. ZCBBs cannot be modeled as CBBs or DBBs because in these blocks, no output event can be generated unless an input event has arrived beforehand.

Synchro basic blocks (SBBs) are the only blocks able to generate output events that are synchronized with their input events. These blocks have a unique event input port, a unique (possibly vector) regular input, no state, no parameters, and two or more event output ports. Depending on the value of the regular input, the incoming event input is routed to one of the event output ports. SBBs are used for routing and under-sampling event signals. Typical examples are the event select block and the if-then-else block [143].

Synchronization. In Scicos if two event signals have the same time, they are not necessarily synchronized. In other words, one is fired just before or just after the other but not "at the same time". Two event signals can be synchronized *only when* they can be traced back to a common origin (a single output event port) through event paths, event additions, event splits, and SBBs alone. In particular, a basic block cannot have two synchronized output event ports. This is possible, however, for super blocks like the 2-freq clock block [143].

3.4.3 Examples

Scicos does not provide a direct way of describing the discrete dynamics of a hybrid automaton as a state machine. Guard conditions have to be implemented using threshold crossing detectors, and reset maps have to be implemented using switches that load different initial conditions to dynamical systems. Moreover, changing the continuous dynamics requires switching outputs and state variables through different integration paths.

Figure 3.15 shows a model of the three-mass system. This model is not complete and can only simulate correctly if $x_{20} < x_{30}$. Besides the fact that when $x_{20} = x_{30}$ the model described in Section 2.2 gives an incorrect answer, the model implemented in Scicos does not guarantee that vx_2 is reset before vx_3. An explicit serialization of the two events should be implemented. Also we assume that $m_i = 1$ and $\epsilon = 0.9$. The three coordinates x_i and y_i are computed by double integration of ax_i and ay_i respectively. Each integrator has three input ports: the input function to integrate, the initial condition and a reset event. When the reset is present, the integrator is reset with the current value on the initial condition input. The horizontal acceleration is always 0 but the initial velocities are determined by selectors whose selection inputs depend on discrete events. For instance, the horizontal velocity of m_2 is reset to $(1 - 0.9)vx_2$ if m_2 hits m_3 (i.e., $x_2 - x_3$ crosses zero), to $(1 + 0.9)vx_1$ if m_1 hits m_2 (i.e., $x_1 - x_2$ crosses zero) and to $0.9vx_2$ if m_2 hits the ground (i.e., y_2 crosses zero).

The reason why this model simulates correctly only for $x_{20} < x_{30}$ is that in the case of events that happen at the same time stamp, their

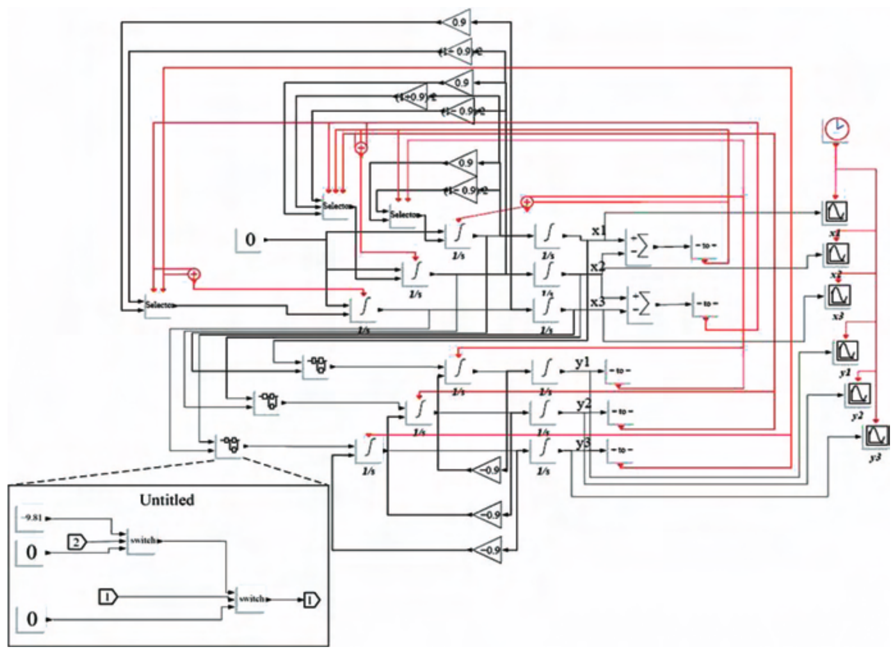

Fig. 3.15 Scicos model of the three-mass system.

order is not specified. If $x_{20} = x_{30}$ then the two events indicating the collision of m_1 with m_2 and m_2 with m_3 are not sequentially ordered and, therefore, the reset conditions are not guaranteed to be sequentially ordered, either. In order to have a correct simulation, it would be necessary to further complicate the model by implementing a priority scheme on the reset actions. The simulation results are shown in Figure 3.16.

The Full Wave Rectifier Example. The rectifier example is shown in Figure 3.17. Similarly to Simulink, Scicos users can organize designs hierarchically by grouping blocks into super-blocks. A diode is a super-block (shown by the sub-figure in Figure 3.17) composed of a switch that selects the output current between two inputs: one proportional to the input voltage and the other constant and equal to $-I_0$. The selection criteria is based on the value of the input voltage: the first input is selected if it is greater than zero, the second otherwise.

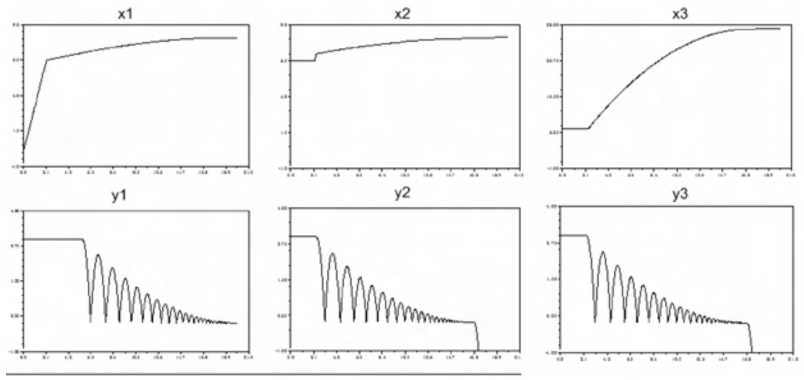

Fig. 3.16 Scicos simulation result of the three-mass system.

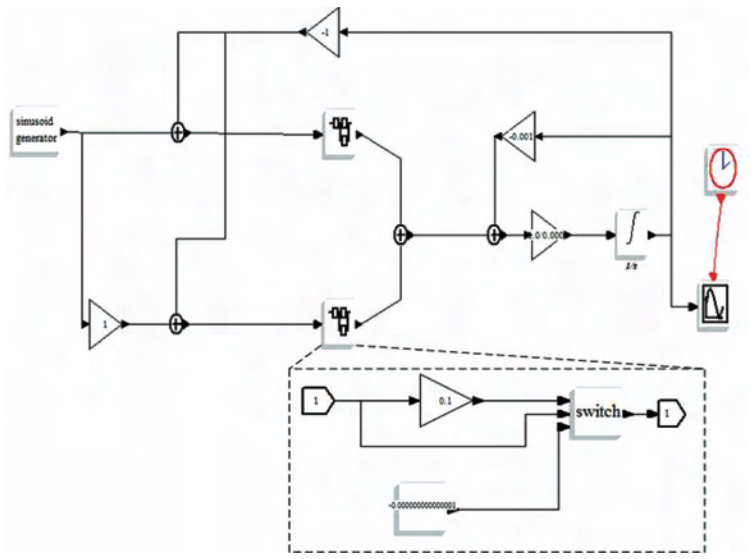

Fig. 3.17 Scicos model of the full wave rectifier.

The simulation results are shown in Figure 3.18. As in the case of Simulink and HyVisual, the circuit cannot be simulated for a pure resistive load due to an algebraic loop error reported by the simulator.

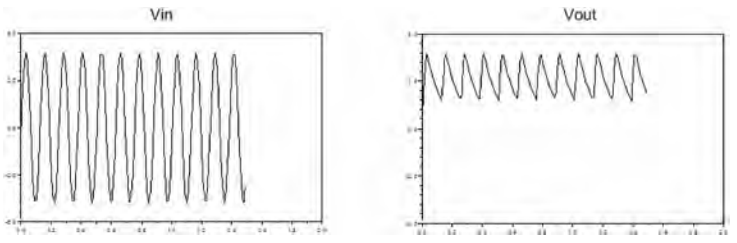

Fig. 3.18 Scicos simulation result of the full wave rectifier.

3.4.4 Discussion

Scicos provides a graphical environment for modeling hybrid systems. Differential equations are described using integrators and other math operators. Even if Scicos can model only causal systems, it is conceptually closer to Modelica than HyVisual. A system, in fact, is modeled by specifying constraints on continuous states and events acting on them. However, Scicos does not provide a graphical tool for the specification and refinement of hybrid automata like the finite state machine editor of HyVisual. Instead, the discrete dynamics must be described using threshold blocks and switches. Building hybrid system models becomes tedious for designers and reverse-engineering a model to its specification could be very difficult. Moreover, adding a state or changing an invariant condition could require major changes in the model netlist.

On the other hand, the Scicos-SynDEx Interface [64] allows users to pair up Scicos and SynDEx, thereby deriving a design flow for distributed real-time embedded control applications that leverages the hybrid systems approach. SynDEx is a system level CAD software for the rapid prototyping and the optimization of distributed real-time embedded applications onto “multi-component” architectures. It is based on the “algorithm-architecture *adequation*” (AAA) methodology [78, 156]. The AAA methodology aims at finding the best match between an algorithm and an architecture while satisfying real-time constraints. This is formalized in terms of graph transformations. The algorithm is specified with a data-flow graph while the architecture is capture via a multiprocessor hyper-graph. Then, an implementation is derived by distributing and scheduling the former on the latter. The result of the graphs transformations is an optimized *Synchronized Distributed Executive* (a SynDEx), which is automatically built from a library of architecture dependent executive primitives composing the executive kernel [78]. These primitives support boot-loading, memory allocation, interprocessor communication, sequencing of user supplied computation functions and of interprocessor communication, and inter-sequences synchronization. The users are provided with a library of executive kernels for various supported processors, while kernels for other processors can be ported from the existing ones. Based on this methodology, SynDEx enables rapid prototyping of complex distributed real-time embedded applications. This is centered on automatic code generation, which is performed in three steps: (1) implementation onto a single-processor workstation for simulation; (2) implementation onto a multi-processor system in order to study parallelism benefits and accelerate simulation; (3) real-time execution on the targeted multi-component architecture which may include programmable components (processors) as well as non-programmable components like application-specific integrated circuits (ASICs). The main feature of the SynDEx software is the seamless environment that guides the user from the specification level (functional specification, distributed hardware specifications, real-time and embedding constraints) to the distributed real-time embedded code level, through (multi-)processor simulations. In particular, it automatically generates, distributes and schedules real-time embedded code.

By relying on the Scicos-SynDEx Interface [64], users can model an embedded control application in Scicos as it is described in Figure 3.19: a model for the physical plant (the environment) is obtained using continuous-time blocks while the controller is designed by assembling discrete-time blocks. The users can perform the "high-level" simulation of the entire hybrid system to reach a first-cut design of the controller. Then, the discrete subsystem modeling the controller is transferred into SynDEx via the provided interface to generate the embedded code for the targeted distributed architecture. This step is simplified by the following facts: (1) Scicos and SynDEx share the same model of computation for the discrete subsystem

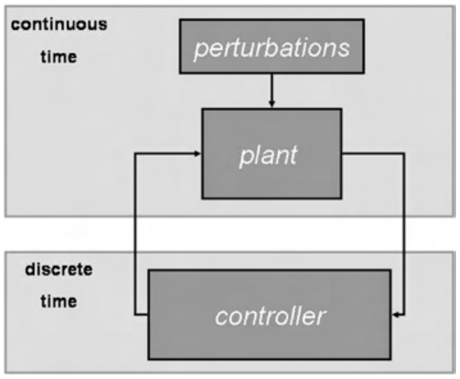

Fig. 3.19 Modeling embedded control as a hybrid system.

(a data flow graph) and (2) the I/O interface of the functional discrete blocks is the same.[8] Also, SynDEx tries to take advantage of the parallelism intrinsically captured by the data flow model to match the parallelism offered by the target architecture, thereby obtaining an implementation that satisfies the real-time constraints. Notice that the interface has been specifically developed for this kind of application and does not support the translation of *continuous-time basic blocks* and *zero-crossing basic blocks*.

3.5 Shift

Shift is a modeling language developed at U.C. Berkeley for the description of *networks* of hybrid automata [60, 61, 150]. The name Shift is a permutation of HSTIF: Hybrid Systems Tool Interchange Format.

The main difference between Shift and other modeling paradigms is that the overall hybrid system in Shift has a dynamically changing structure. More precisely, the entire system in Shift is called the *world*. The world consists of a certain number of hybrid components that can be destroyed or created in real-time as the system evolves. Therefore the Shift language is mainly used for the description and simulation of highly complex hybrid systems whose configuration varies

[8] In fact, it may be the case sometimes that a single Scicos block is translated into a group of SynDEx blocks. For further details see [64].

over time. The conception of Shift was motivated by the specification and analysis of designs for the automatic control of vehicles and highway systems (AHS) [16, 58, 59, 76, 169]. The research area involved in this approach is quite rich, going from the design and validation of communication protocols [74, 102] to the verification of safe design [62, 81, 113, 146], and including the development of suitable implementation methodologies [68, 75]. Hence the need of a modeling framework that is general enough to capture all these distinct issues, while staying at a low level of complexity to facilitate learning and formal analysis.

At the time Shift was developed, other modeling paradigms for the composition of multiple concurrent agents included extended FSMs [106], Communicating Sequential Processes [99], DEVS [101], SDL [105] and also the models of computation described in [104, 137, 175]. However none of them had the feature to model dynamic configurations of hybrid components. The characteristic of being able to describe dynamic networks of hybrid systems makes Shift quite unique as a modeling and simulation tool. Areas of application possibly include, together with the mentioned AHS, air traffic control systems, robotics shop-floors, and coordinated robotic agents with military applications, like Unmanned Aerial Vehicles (UAV) (see [108, 107, 158] and the references contained therein).

3.5.1 Shift Syntax

A **world** in Shift is a set:

$$W = \{h_1, \cdots, h_w\}$$

where h_i is called the i-th hybrid component in the world. A hybrid component can be viewed as a hybrid automaton A_H having Q as the set of discrete states. In each state $q \in Q$ the continuous state x follows a continuous evolution determined by the flow F_q, which can be of the form of a differential constraint or even a simple algebraic definition. An instantiation of a hybrid component is called a *type*. A type is a tuple: $H = (q, x, C, L, F, T)$, where:

- $q \in Q$ is the discrete state variable;

- $x \in \mathbb{R}^n$ is the continuous state variable;
- $C = (C_0, \cdots, C_m)$ with each $C_i \subset W$ is the configuration state variable;
- $L = \{l_1, \cdots, l_p\}$ are the event labels;
- $F = \{F_q \mid q \in Q\}$ are the flows;
- T are the transition prototypes.

Each component h is, at a specified time, in a particular configuration C_h. Hence the *configuration* (or discrete state) of the world is given by the tuple:

$$C_W = (C_{h_1}, \cdots, C_{h_w})$$

The continuous state x can be constrained in one of the following ways:

a) $\dot{x}_i = F^D_{i,q}(x, x_{C_0})$ for differential constraints;
b) $x_i = F^A_{i,q}(x, x_{C_0})$ for algebraic constraints;

where x_{C_0} is a vector containing the continuous state variables of all the elements of C_0. The set of transitions T is a set of tuples δ of the form:

$$\delta = (q, q', g, E, a)$$

where:

- $q, q' \in Q$ are respectively the *source* and *sink* (discrete) states of the transition.
- g is a *guard condition*: it takes the form of a (possibly quantified) Boolean expression. It can assume one of two forms:
  (1) $g(x, x_{C_0})$ (Boolean predicate);
  (2) $\exists c \in C_i : g(x, x_{C_0}), 1 \le i \le m$ (Boolean predicate).
- E is a set of event labels whose purpose is to synchronize the current component with the rest of the world. An "internal" transition, i.e., a transition which does not synchronize with transitions occurring in the other components of the world, is specified in Shift by leaving E empty.

- a is an action that modifies the state of the world. An action may also create or destroy new components.

In the rest of this section, we give the precise syntax definition of components and transitions and we will omit parts that are not essential for understanding the language semantics. In the following syntactic definitions, non-terminals are in *italics*. Keywords and other literal tokens are in sans-serif. Braces indicate repetition: $\{X\}^*$ means zero or more repetitions of X, $\{X\}^+$ means one or more repetitions. Brackets indicate optional parts, that is $[X]$ stands for zero or one instances of X. The vertical bar ('|') denotes alternation.

A Shift specification is a sequence of *definitions*:

$$\begin{array}{rcl}
\textit{specification} & \Rightarrow & \{\ \textit{definition}\ \}^+ \\
\textit{definition} & \Rightarrow & \textit{component-type-definition} \\
 & | & \textit{external-type-definition} \\
 & | & \textit{global-variable-decl} \\
 & | & \textit{external-function-decl} \\
 & | & \textit{global-setup-clause}
\end{array}$$

A component type definition describes a set of components with common behavior.

$$\begin{array}{rcl}
\textit{component-type-definition} & \Rightarrow & \textsf{type}\ \textit{type-name}\ [\texttt{:}\ \textit{parent}] \\
 & & \texttt{\{}\ \{\ \textit{type-clause}\ \texttt{;}\ \}^+\ \texttt{\}}\ [\texttt{;}] \\
\textit{type-clause} & \Rightarrow & \textsf{state}\ \textit{state-declarations} \\
 & | & \textsf{input}\ \textit{input-declarations} \\
 & | & \textsf{output}\ \textit{output-declarations} \\
 & | & \textsf{export}\ \textit{export-declaration-list} \\
 & | & \textsf{setup}\ \textit{setup-clause} \\
 & | & \textsf{flow}\ \textit{flow-list} \\
 & | & \textsf{discrete}\ \textit{discrete-state-list} \\
 & | & \textsf{transition}\ \textit{transition-list}
\end{array}$$

Each type can have inputs, outputs and states. Output variables can be written to and are visible outside the component and can be used by other components. Input variables are defined as external to the component while states are not visible outside. Exported events can be used to synchronize discrete state transitions among components. The keyword flow is used to define differential and algebraic constraints on variables. Each flow is a set of equations and is identified by a name. The keyword discrete is used to define a discrete state with an associated name, flow and list of synchronization labels:

$$\begin{array}{rcl}
\textit{discrete-state-list} & \Rightarrow & \textit{discrete-state-clause} \\
 & & \quad \{\ \texttt{,}\ \textit{discrete-state-clause}\ \}^{*} \\
\textit{discrete-state-clause} & \Rightarrow & \textit{state-name}\ [\ \texttt{\{}\ \textit{equation-list}\ \texttt{\}}\ ] \\
 & & \quad [\ \textsf{invariant}\ \textit{expression}\ ]
\end{array}$$

A transitions is defined as follows:

$$\begin{array}{rcl}
\textit{transition-list} & \Rightarrow & \textit{transition}\ \{\ \texttt{,}\ \textit{transition}\ \}^{*} \\
\textit{transition} & \Rightarrow & \textit{from-set}\ \texttt{->}\ \textit{to-state}\ \textit{event-list} \\
 & & \quad \textit{transition-clauses} \\
\textit{from-set} & \Rightarrow & \textit{set-of-states} \\
\textit{to-state} & \Rightarrow & \textit{state-name} \\
\textit{event-list} & \Rightarrow & \texttt{\{}\ [\ \textit{event}\ \{\texttt{,}\ \textit{event}\}^{*}\ ]\ \texttt{\}} \\
\textit{event} & \Rightarrow & \textit{local-event} \\
 & | & \textit{external-event} \\
\textit{external-event} & \Rightarrow & \textit{link-var}\ \texttt{:}\ \textit{exported-event}\ [\ \texttt{(}\ \textit{set-sync-rule}\ \texttt{)}\ ] \\
\textit{set-sync-rule} & \Rightarrow & \textsf{one}\ [\texttt{:}\ \textit{temporary-link}] \\
 & | & \textsf{all} \\
\textit{transition-clauses} & \Rightarrow & [\ \textsf{when}\ \textit{expression}\ ]\ [\ \textit{action-clause}\ ]
\end{array}$$

A transition specifies the source and target states, a list of synchronization events and a set of actions to be taken depending on some conditions. Events in the event list can be locally defined (i.e., *local-event*s) and exported or can be events defined and exported by other components (i.e., *external-event*s). Events can be of type open or closed. An action is a set of reset assignments, creation of components and connection of components.

3.5.2 Shift Semantics

A Shift system starts by executing all initializations of global variables, at time $t = 0$. Then, the system evolves by alternating *discrete* and *continuous* phases, starting with a discrete phase. In the discrete mode, all possible transitions are taken, in some serial order unless explicitly synchronized. Time does not pass in the discrete mode. The system switches to continuous mode when no more transitions are possible. The system evolves in continuous mode according to the flow associated to the discrete state of each component. As soon as it becomes possible for one or more components to execute a transition, time stops again. A component synchronizes its state machine with other state machines by labeling its own edges with *local-event*s and *external-event*s. Local events are exported; they can be used as external events by other components, and they can appear in *connection actions*. Each label of an edge E establishes conditions under which a transition may be taken along E. When all conditions are satisfied, and the guard, if present, evaluates to true, and the component is in a state that has E as an outgoing edge, then the transition along E is taken simultaneously with other transitions as required by the conditions. The conditions associated with each label are as follows. Let x and y be components, and Z a set of components. Let c be a single-valued link, and C a set-valued link. Let e_y be a local event for y, and e_z a local event for all components in Z.

- If c evaluates to nil, an edge labeled $c{:}e_y$ may not be taken.
- If c evaluates to y, an edge E labeled $c{:}e_y$ must be taken simultaneously with an edge E' labeled e_y in y.
- If e_y is of type open then an edge E' labeled e_y in y requires that there exists a component x with an edge E labeled $y{:}e_y$ and E' must be taken simultaneously with E. However, if e_y is of type closed then:

  – if there is no other component x with an edge E labeled $c{:}e_y$, where c evaluates to y, then E' may be taken alone.

  – if there is at least one other component x with an edge E labeled $c{:}e_y$, where c evaluates to y, then E' must be taken simultaneously with E.

- If C evaluates to the empty set, the edge labeled $C{:}e_z$ may not be taken if set-sync-rule is one. Otherwise it may be taken.
- If C evaluates to Z then an edge labeled e_z in any $z \in Z$ may only be taken simultaneously with an edge labeled $C{:}e_z$. The following also applies.
  - If the synchronization rule is one, then an edge labeled $C{:}e_z$ may only be taken simultaneously with an edge labeled e_z in a single component $z \in Z$. If a temporary link is specified, it is assigned the component z. The scope of the temporary link is the action list for the transition.
  - Otherwise, if the rule is all, an edge labeled $C{:}e_z$ must be taken simultaneously with an edge labeled e_z in every $z \in Z$.

Actions are executed in phases as follows.

(1) All components specified by *create-expression*s are created.
(2) The right-hand sides and the destinations of resets are evaluated, and so are the component initializers.
(3) The previously computed values for resets and link actions and component initial values are assigned to their destinations.
(4) Connection actions are executed.

3.5.3 Examples

The point masses m_1, m_2 and m_3 are modeled in SHIFT as instantiations of a type “pointmass”. A pointmass exposes many variables to the other components of the world and exports a collision event. It also has a “connection” with the mass to its right and the one to its left:

```
type pointmass {
  output continuous number hvelocity, hposition;
  output continuous number L, h;
  output continuous number x,vx,y,vy;
```

```
state continuous number ay := 0.0;
output pointmass rightmass := nil;
output pointmass leftmass := nil;
export collisiontoright;
flow default {
   x' = vx;
   vx' = 0.0;
   y' = vy;
   vy' = ay;
   hvelocity = vx;
   hposition = x;
} ;
discrete
   on ,
   off ;
transition
   on − > on {collisiontoright} /*This mass collides with another
   one*/
   when ( rightmass /= nil and x >= hposition(rightmass) and
   vx > hvelocity(rightmass))
   do {
      /*Reset my velocity*/
      vx := vx*(1-0.9)/2 + hvelocity(rightmass)*(1-0.9)/2;
   },

   on − > on {leftmass:collisiontoright} /*Another mass collides
   with this one*/
   do {
      /*Reset my velocity*/
      vx := hvelocity(leftmass)*(1+0.9)/2 + vx*(1-0.9)/2;
   },
   on − > off {} /*Falling*/
   when ( x >= L and vx > 0 and y >= h )
   do {
      ay := -9.81;
   },
    off − > off {} /*Bouncing*/
    when ( y <= 0 and vy < 0)
    do {
      vx := 0.9*vx;
      vy := -0.9*vy;
    };
}
```

A point mass has two states: on and off. In the on state, it can collide with the mass to its right or can be hit by the mass to its left. When the point mass m_i collides with the mass to its right m_j, the collision event collisiontoright is notified. Mass m_j has a transition that is synchronized with the event leftmass:collisiontoright. The two on− >on transitions in m_i and m_j are then taken together at precisely the same time. Note that discrete states do not specify a flow, i.e., the same flow, denoted by the keyword default, is assumed to define the dynamics in each discrete state. Instantiation, creation and interconnection of types is done by the following code:

```
global threemass t := create(threemass);
type threemass {
  output pointmass m1 := create( pointmass, L := 7.0, h := 3.0,
  x := 0.0, vx := 3.0, y := 3.0, vy :=0);
  output pointmass m2 := create( pointmass, L := 7.0, h := 3.0,
  x := 6.5, vx := 0.0, y := 3.0, vy := 0);
  output pointmass m3 := create( pointmass , L := 7.0, h := 3.0,
  x := 7.0, vx := 0.0, y := 3.0, vy := 0);
  discrete a;
  setup
    do{
      rightmass(m1) := m2;
      leftmass(m2) := m1;
      rightmass(m2) := m3;
      leftmass(m3) := m2;
    };
}
```

The three masses are created and initialized. The threemass type has only one discrete state. Before entering state a the setup clause is executed and connections among components are established.

The SHIFT source code is compiled into standard C code which is used, together with other libraries, to generate an executable simulation file. The user can choose between a command line and a graphical interface for debugging the code. Since we could not compile the graphical user interface, we had to rely on the textual printing ability of the SHIFT executable simulation in order to show the correctness of the model.

The Full Wave Rectifier Example. The full wave rectifier is modeled as a set of components:

```
type diode {
  output continuous number i;
  input source s;
  flow
    res {
      i = 10.0 * v(s) ;
    },
    zeroi {
      i = 0.0;
    };
  discrete
    forward {res},
    reverse {zeroi};
  transition
    forward − > reverse {}
    when ( v(s) < 0.0 ) ,
    reverse − > forward {}
    when ( v(s) >= 0.0 ) ;
}
type source {
  output continuous number v;
  output continuous number vsd;
```

```
    output continuous number vs;
    input load l;
    input continuous number w0;
    flow default {
        vsd' = -w0*w0*vs;
        vs' = vsd;
        v = vs - v(l);
    }
    discrete a;
}
type load {
    output continuous number v;
    input diode d1;
    input diode d2;
    flow default {
        v' = - v*10.0 + (i(d1) + i(d2))*10000.0;
    }
    discrete a;
}
```

A diode is a type with two states: forward and reverse. When in forward state, the flow that defines the output current is the Ohm's law. When in reverse state, the output current is set to zero. The input voltage to the diodes is generated by two sources. A source generates an output voltage equal to the difference of an internally generated sinusoidal waveform and the output voltage of a load component.

The creation and interconnection of all the components is carried out by the following SHIFT program:

```
global rectifierRC r := create(rectifierRC);
type rectifierRC {
    output source s1 := create( source, vs := 0.0, vsd := 4.0*314.0,
    w0 := 314.0 );
    output source s2 := create( source, vs := 0.0, vsd := -4.0*314.0,
    w0 := 314.0 );
    output diode d1 := create(diode, i := 0.0);
    output diode d2 := create(diode, i := 0.0);
    output load l := create(load, v := 0.0);
    discrete a;
    setup
      do {
        l(s1) := l;
        l(s2) := l;
        s(d1) := s1;
        s(d2) := s2;
        d1(l) := d1;
        d2(l) := d2;
      };
  }
```

Notice that the two sources are initialized with different values in order to generate a sine wave and its opposite respectively for $s1$ and $s2$.

3.5.4 Discussion

Shift is a modeling paradigm for the description of dynamic networks of hybrid components. The major distinction with respect to other modeling languages for hybrid systems (like Charon, or Masaccio) is that in Shift the configuration of the examined system (called *world* in the Shift jargon) is dynamic, meaning that it results from the continuous creation/destruction of objects, each modeling a distinct hybrid sub-system. This description of networks of hybrid automata, which is intrinsic to Shift, can in principle also be carried out using other modeling languages, but it would require additional effort because languages like Charon or Masaccio are oriented towards a static description of the modeled system.

A Shift component can export events. Components can label their transitions with events exported by other components. Since such events can be emitted on automata transitions, Shift allows composition of hybrid systems both in the continuous and discrete domains. The automata synchronization feature eases the composition of models and results in compact specifications as in the case of the three mass system.

SHIFT is both a programming language and a run-time environment for the simulation of dynamic networks of hybrid automata [153]. A compiler for translating a SHIFT program to a C program is also available. More recently a new language has been developed by the research group that created SHIFT. Its name is λ-SHIFT [154]. Like its predecessor, λ-SHIFT is a language for the specification of dynamic networks of hybrid components and it is designed to provide a tool to simulate, verify and generate real-time code for distributed control systems arising in applications like AHS and the other mentioned above. What really distinguishes λ-SHIFT from its predecessor is the syntax: λ-SHIFT is an extension of the Common Lisp Object System (CLOS) [37, 157]. In particular, in order to provide a better use of the CLOS capabilities, the Meta-Object Protocol (MOP) [36] has been extended to provide an *open* and *specializable* implementation of the λ-SHIFT specification language.

3.6 Charon

CHARON, an acronym for *coordinated control, hierarchical design, analysis and run-time monitoring of hybrid systems*, is a high-level language for modular specification of multiple, interacting hybrid systems developed at the University of Pennsylvania [3, 4, 5, 8]. CHARON is implemented and distributed in a toolkit that includes several tools for the specification, development, analysis and simulation of hybrid systems. The CHARON toolkit is entirely written in JAVA and features: a graphical user interface (GUI), a visual input language (similar to STATEFLOW), an embedded type-checker, and a complete simulator. The graphical input editor converts the specified model into CHARON source code, using an intermediate XML format. The plotter is based on a package from the modeling tool Ptolemy, developed at U.C. Berkeley. It supports the visualization of system traces as generated by the simulator. The CHARON toolkit is also fully compatible with external programs written in JAVA; the simulator itself is an executable Java program. The CHARON toolkit Version 1.0 is freely distributed and can be downloaded from http://www.cis.upenn.edu/mobies/charon.

3.6.1 Charon Syntax

The Charon language enables specification of *architectural* as well as *behavioral* hierarchies and discrete as well as continuous activities.

The architectural hierarchy reflects the composition of distinct processes working in parallel. In this framework, the basic building block is represented by an *agent*. Agents model distinct components of the system whose executions are all active at the same time. They can be of two types: primitive and composite. Primitive agents are the primitive types or basic building blocks of the architectural hierarchy. Composite agents are derived by *parallel composition* of primitive agents. Other main operations supported by agents are *variable hiding* and *variable renaming*. The hiding operator makes a specified set of variables private or local, that is other agents cannot access private variables for read/write operations. Variable hiding implements *encapsulation* for data abstraction. Variable renaming is for supporting instantiation of distinct components having the same structure. Agents communicate among themselves and with the external environment by means of shared variables, which represent input/output/state signals of the overall hybrid system.

The behavioral hierarchy is based on the sequential composition of system components acting sequentially in time. Such components are called *modes*. Modes represent the discrete and continuous behaviors of an agent. Each agent consists of one or more distinct modes that describe the flow of control inside an agent. Modes can contain the following elements: control points (entry points, exit points), variables (private, input, output), continuous dynamics, invariants, guards, and nested submodes. *Control points* are where the flow of control enters or exits the given mode. The execution of the mode starts as soon as the flow of control reaches an entry point and ends when it reaches an exit point. A *guard condition* can be associated to each control point (entry point or exit point). A guard condition is a rule or a set of rules enabling the control flow to actually go trough a given entry or exit point, i.e., enabling the hybrid systems to make a *jump* or discrete transition. As for agents, variables in a mode represent discrete or continuous signals. Input and output variables represent respectively input and output signals of the agent, while private variables either represent state signals, which are not visible externally, or *true* auxiliary variables such as those necessary to perform some functional computation. Modes can be *atomic* or *composite*; composite modes contain nested submodes which can themselves be composite or atomic. Modes can have three types of constraints:

- **invariants**: the flow of control can reside in a mode as long as an inequality condition, called the *invariant*, is satisfied (e.g., if x and y are two variables, an invariant can be of the form $|x - y| \leq \epsilon$). When invariants are violated the flow of control must exit the active mode from one of its exit points.
- **differential constraints**: these are used for modeling continuous dynamics evolving in the current mode (e.g., by differential equations, like: $\dot{x} = f(x, u)$).
- **algebraic constraints**: algebraic equations model resets of variables occurring during discrete transitions of the hybrid system. The values of the variables are reassigned using an algebraic expression, such as $y = g(x, u)$.

Agents and modes are represented as tuples. If $T = (t_1, \ldots, t_n)$ is a tuple then the element t_i of T is denoted as $T.t_i$. This notation can be extended to collection of tuples, so that if ST is a set of tuples, then:

$$ST.t_i = \bigcup_{T \in ST} \{T.t_i\}$$

Variables should be formally distinct from their valuations: given a set V of variables a *valuation* is a function mapping variables in V to their respective values. Q_V denotes the set of all possible valuations over V. If s is a valuation of variables in V and $W \subseteq V$, then $s[W]$ is the restriction of the valuation s to variables in W. Continuous-time behaviors of modes are modeled by flows. A flow is a differentiable function $f : [0, \delta] \rightarrow Q_V$, where δ is called the *duration* of the flow.

A mode is a tuple $(E, X, V, SM, Cons, T)$ where:

- E is a set of entry points and X is a set of exit points. There are two particular control points: a *default entry* $de \in E$ and a *default exit* $dx \in X$.
- V is a set of variables, which can be analog or discrete (characterizing signals for flows and jumps of the hybrid system, respectively). Variables can also be local, their scope being limited only to the active mode, or global, if they can be accessed externally.
- SM is a finite set of submodes.
- $Cons$ is a set of constraints, which can be of three types: *differential, algebraic* and *invariant*, as described above.
- T is a set of transitions of the kind (e, α, x), where: $e \in E \cup SM.X$ and $x \in X \cup SM.E$; α, called the *action* associated to the current transition, is a relation from Q_V to Q_V and it updates variables (analog or discrete and global or local) when the mode undergoes the transition T.

A mode with $SM = \emptyset$ is called *atomic*. *Top-level* modes are composite modes that are not contained in another mode (they can only be contained in agents); they have only one non-default entry point and have no default exit points.

The syntax of agents is simpler than that of modes. An agent is formally defined as a tuple (TM, V, I), where V is a set of variables, I is a set of initial states and TM is a set of top level modes. The set of variables V results from the disjoint union of the set of global variables V_g and local variables V_l; formally: $V = V_g \cup V_l$ with $V_g \cap V_l = \emptyset$. The set I of initial states can specify a particular initialization of the variables in the agent. The elements of an agent can be accessed through the "dot" operator: for example, $A.V_g$ is the set of global variables of the agent A.

Intuitively, top-level modes in TM describe the behavior (i.e., execution in time) of the agent. As for modes, variables in agents can be local or global. Primitive agents have only one top-level mode, while composite agents contain several top-level modes and can be obtained as the parallel composition of primitive agents.

The execution of an agent can be derived from those of its top-level modes. A primitive agent has a single top-level mode, while composite agents have several top-level modes (each possibly containing submodes) and results from the parallel composition of other agents. Execution trajectories start from the specified set of initial states and consist of a sequence of flows interleaved with jumps, defined by the modes associated to the agent. In particular, the jumps correspond to discrete transitions of the environment or of one of the modes of the agent, while flows are concurrent continuous executions of all the modes of the agent. Traces are obtained similarly by projecting onto the global variables.

Agents can be combined using the operators of variable hiding, variable renaming and parallel composition. The hiding operator makes a set of variables in an agent private. Given an agent $A = (TM, V, I)$, the restriction to V_h is the agent $A \setminus \{V_h\} = (TM, V', I)$ with $V'_l = V_l \cup V_h$ and $V'_g = V_g - V_h$. The renaming operator makes a replacement of a set of variables inside an agent with another set of variables. This is useful for interfacing the agent with its external environment (i.e., with other agents). Let $V_1 = \{x_1, \ldots, x_n\}$ and $V_2 = \{y_1, \ldots, y_n\}$ be indexed sets of variables with $V_1 \subseteq A.V$. Then $A[V_1 := V_2]$ is an agent with the set of global variables $(A.V_g - V_1) \cup V_2$. Parallel composition is used to combine agents to form a hierarchical structure. The parallel composition $A_1 \,||\, A_2$ of the two agents A_1 and A_2 is an agent A defined by the following relations:

- $A.TM = A_1.TM \cup A_2.TM$
- $A.V_g = A_1.V_g \cup A_2.V_g$ and $A.V_l = A_1.V_l \cup A_2.V_l$
- if $s \in A.I$ then $s[A_1.V] \in A_1.I$ and $s[A_2.V] \in A_2.I$

3.6.2 Charon **Semantics**

Modes can exhibit both a continuous and discrete behavior, but not at the same time: this implies that a mode undergoes a sequence of *jumps* (discrete transitions) and *flows* (continuous executions). During a flow the mode follows a continuous trajectory subject to the corresponding differential constraints. As soon as the trajectory no longer satisfies the invariant constraints, the mode is forced to make a discrete transition.

A jump is a finite sequence of discrete transitions of submodes and transitions of the mode itself that are enabled by the corresponding guards. Any discrete transition starts in the current active state of the mode and terminates as soon as either a *regular* exit point is reached or the mode yields control to its external environment via one of its default exit control point.

Formally, the semantics of a mode is represented by its set of *executions*. An execution is a path through the transition graph induced by the mode and its submodes of the form

$$(e_0, s_0) \xrightarrow{\lambda_1} (e_1, s_1) \xrightarrow{\lambda_2} \cdots \xrightarrow{\lambda_n} (e_n, s_n),$$

where e_i is a control point and s_i a state in the form of a variable evaluation. The transitions λ_i represent either discrete jumps or flows. Jumps can be taken by the mode, in which case they are denoted by a circle o, or by the environment (changes to the global variables of the mode by other components while the mode is inactive), denoted by ϵ. The initial and final state s_i and s_{i+1} of a jump, as well as the corresponding control points must be consistent with the transitions and the corresponding action labels of the mode. Otherwise, λ_i is a flow f_i of s_{i-1} defined over $[0, t]$ (the duration of the flow), and such that $f_i(t) = s_i$. Externally, the semantics of a mode is represented by its set of *traces*, which are obtained by projecting the executions onto the global variables of the mode. That is, a trace is obtained from each execution by replacing every s_i with $s_i[V_g]$, and every f in transition labels with $f[V_g]$.

Compositionality. The semantics of CHARON is *compositional* in the sense that the semantics of one of its components (possibly the entire hybrid system) is entirely specified in terms of the semantics of its subcomponents. Compositionality holds for both agents and modes. Indeed, the set of traces of a given mode is determined by the definition of the mode itself and by the semantics of its submodes. For a composite agent the set of traces can be reconstructed from the traces of its top-level modes.

Compositionality results can be extended to the operators on agents by introducing a refinement relation on modes and agents. A mode M *refines* a mode N, written $M \preceq N$, if it has the same global variables and control points, and every trace of M is a trace of N. The compositionality properties implies that if $M.SM \preceq N.SM$, then $M \preceq N$.

Similarly, an agent A *refines* an agent B if $A.V_g = B.V_g$, and every trace of A is a trace of B. Compositionality results holding for modes can be naturally extended to agents because an agent is basically a collection of modes with synchronized flows and interleaving jumps. In particular agent operators are *compositional with respect to refinement*. Formally, the result states that the operations on agents are monotonic relative to the refinement order. Thus, assume $A \preceq B$, $A_1 \preceq B_1$ and $A_2 \preceq B_2$ are agents, $V_1 = \{x_1, \ldots, x_n\}$ and $V_2 = \{y_1, \ldots, y_n\}$ are indexed sets of variables with $V_1 \subseteq A.V$, and let $V_h \subseteq A.V$. Then, $A \setminus \{V_h\} \preceq B \setminus \{V_h\}$, $A[V_1 := V_2] \preceq B[V_1 := V_2]$ and $A_1 \mid\mid A_2 \preceq B_1 \mid\mid B_2$. This result is particularly useful to help reduce the complexity of refinement verification by applying compositional techniques. In practice, refinement can be verified at the component level using predicate abstraction (to reduce the complexity to a finite state model), and can be extended to the entire system using the compositionality result [4].

3.6.3 Examples

The Charon distribution comes with a graphical user interface for the specification of agents, modes and their interconnection. CharonVisual is a Java front-end that can be used to input a hybrid system specification. In addition, CharonVisual can generate a Charon netlist that can be compiled and simulated. Figure 3.20 shows a model of the full wave rectifier circuit. The system is composed of four agents: two diodes, a load and a source block. A diode has two modes: forward and reverse. A project is stored in an XML file with all the model as well as graphical information. The diode agent contains a top mode defined by the following code snippet:

```
mode DiodeTop()
  read analog real vin;
```

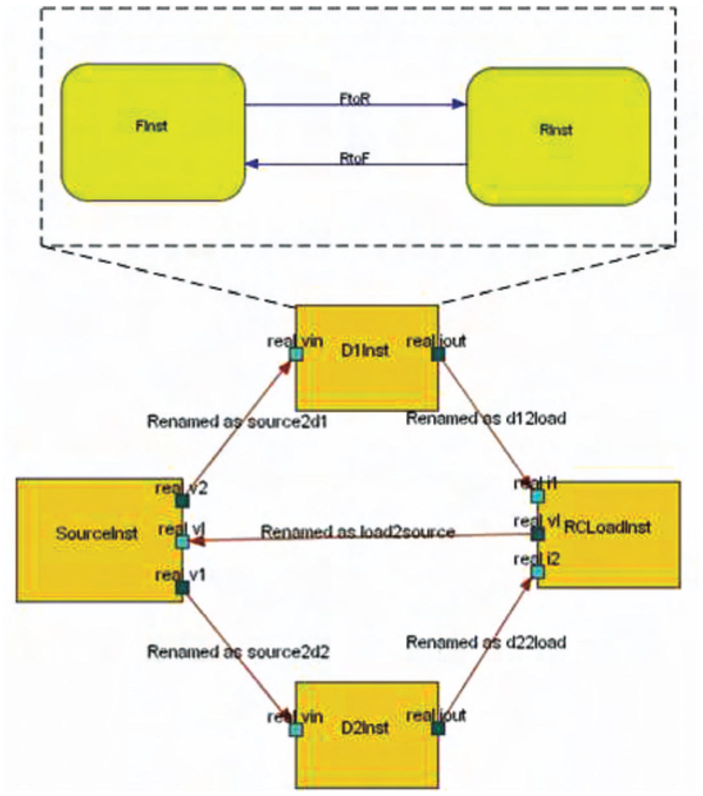

Fig. 3.20 CHARON model of the full-wave rectifier circuit.

readWrite analog real iout;
mode FInst = forward();
mode RInst = reverse();
trans start **from** default **to** FInst **when** true **do** { }
trans F2R **from** FInst **to** RInst **when** (vin < 0) **do** { }
trans R2F **from** RInst **to** FInst **when** (vin > 0) **do** { }

The forward mode is described as follows:

mode forward()
  readWrite analog real iout;
  read analog real vin;

```
    inv Finv { vin >= 0 }
    alge Outeq { iout == vin }
```

In forward mode the diode's output current is proportional to the input voltage by a constant that in this case we assume to be equal to one. The relation between input voltage and output current is declared in an algebraic constraint. The invariant constraint declares that a diode stays in forward mode as long as the input voltage is greater than or equal to zero. When the invariant is violated, the output transition F2R is enabled and the diode switches to the reverse mode whose output current is equal to $-I_0$.

The load is modeled as a dynamical system:

```
agent RCload()
    read analog real i2;
    read analog real i1;
    readWrite analog real vl;
    init { vl = 0 }
    mode top = RCloadTopMode(0.00001, 1000);

mode RCloadTopMode(real C, real R)
    read analog real i2;
    read analog real i1;
    readWrite analog real vl;
    diff Loadeq { d(vl) == -vl/( R * C) + (i1 + i2)/C }
```

It has one mode of operation that declares one differential constraint for the load voltage. Simulation results are shown in Figure 3.21.

The Three-Mass Example. We model the three-mass system with only one agent in order to show how modes can be hierarchically organized. The hybrid system model is very similar to the Simulink/Stateflow one. The minor differences concern the invariant specification. Figure 3.22 shows the complete model. Each mode is characterized by the same differential constraint that specifies the motion of the three masses:

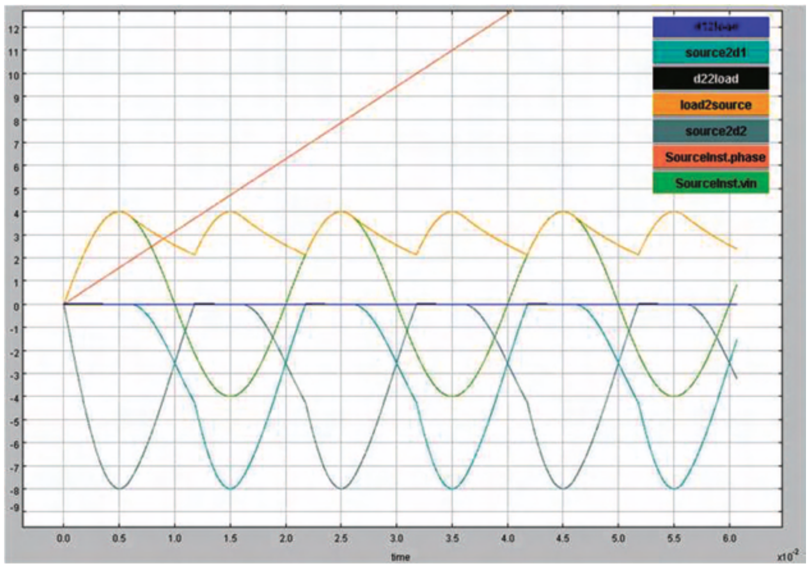

Fig. 3.21 CHARON simulation results of the full-wave rectifier with RC load.

```
diff motion {
   d(vx1) == 0.0; d(x1) == vx1; d(vy1)==ay1; d(y1)==vy1;
   d(vx2) == 0.0; d(x2) == vx2; d(vy2)==ay2; d(y2)==vy2;
   d(vx3) == 0.0; d(x3) == vx3; d(vy3)==ay3; d(y3)==vy3
}
```

Unlike SIMULINK/STATEFLOW and other tools like HYVISUAL that have triggering transition semantics, CHARON has enabling semantics meaning that a system is allowed to stay in a mode as long as the invariant constraint is satisfied (even if a guard on a transition is also satisfied). Therefore, we must declare in each mode an invariant constraint that is the conjunction of the complement of the guards on the output transitions. To this end, we have to distinguish, for instance, mode $m3bounce$ (where $m2$ and $m1$ are still on the table) from mode $m3purebounce$ (where all masses have fallen from the table). The reason is that in the first case $m1$ and $m2$ can still collide, thereby requiring a transition to mode $m1tm2$, while in the second case the collision cannot happen. Figure 3.23 shows the simulation result. First we note that, in the simulation, the balls keep moving to the right, despite

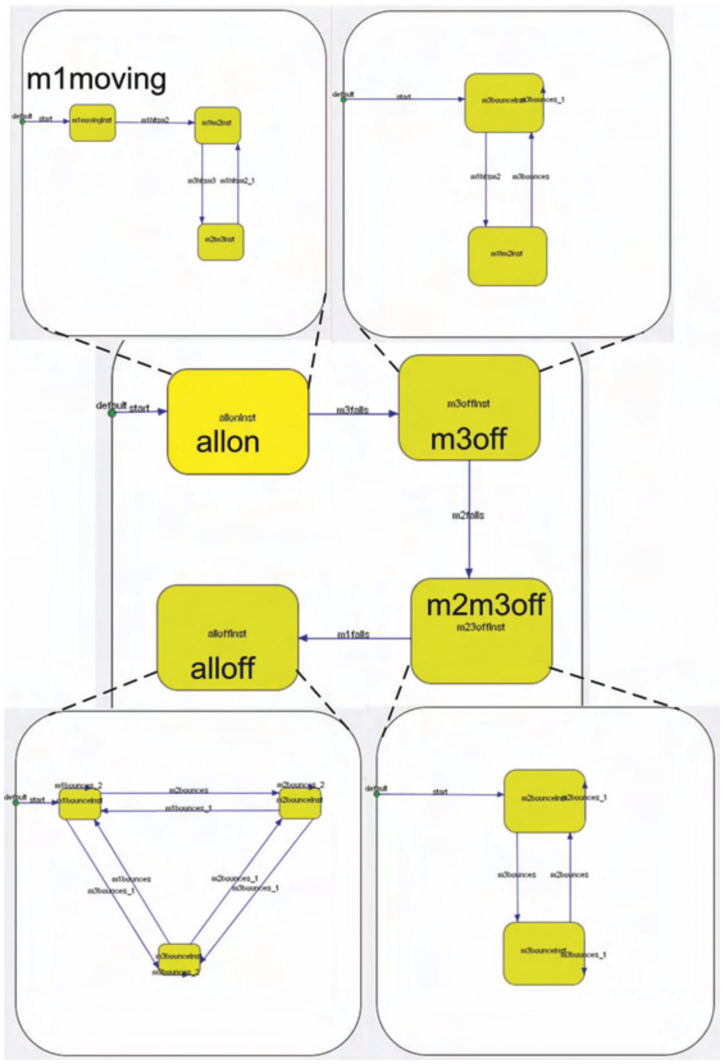

Fig. 3.22 Charon model of the three-mass system.

the fact that, because the system is Zeno, they shouldn't move past a certain point. This artifact is a consequence of the minimum time imposed by Charon in traversing each state, a condition that causes time to always progress. Second, the balls (correctly) do not fall below the floor level, contrary to the other tools that we have evaluated. This

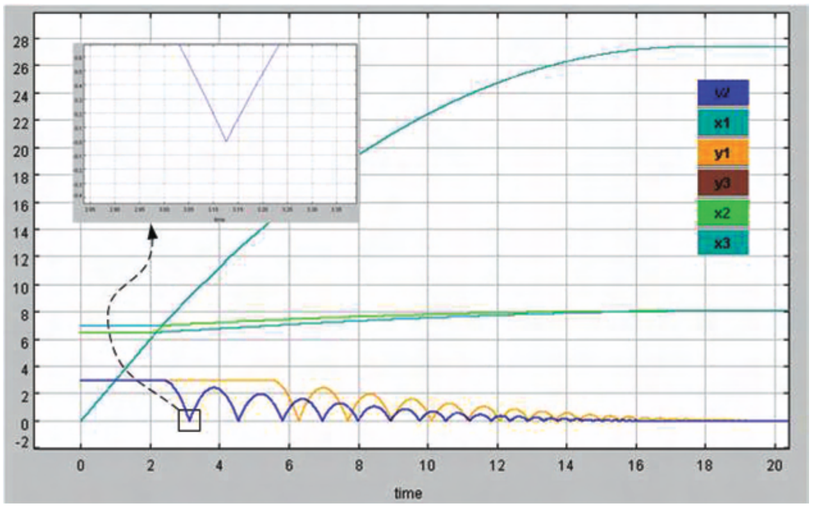

Fig. 3.23 CHARON simulation result of the three-mass system.

is because the transitions are not only sensitive to events, i.e., changes in the values of the variables that may go undetected because of the size of the integration step, but are also forced by the violation of the state invariants, which are static constraints evaluated on the present value of the variables.

3.6.4 Discussion

By combining the notions of *agent* and *mode* the language CHARON supports the hierarchical modeling of hybrid systems both at the architectural and behavioral level. For the hierarchical description of the system architecture, CHARON provides the operations of instantiation, hiding, and parallel composition on agents, which can be used to build a complex agent from other agents. Modes are used to describe the discrete and continuous behaviors of an agent. For the hierarchical description of the behavior of an agent, CHARON supports the operations of instantiation and nesting of modes. The description of complex discrete behaviors is facilitated by the availability of features such as weak preemption and history retention, as well as by the possibility of invoking externally defined JAVA functions. Continuous behaviors can be specified using differential as well as algebraic constraints, and invariants restricting the flow spaces, all of which can be declared at various levels of the hierarchy. Guard conditions in CHARON are enabling and not triggering like, for instance, in HYVISUAL. This means that an enabled guard condition may or may not necessarily be taken. This is an important point to keep in mind when one builds a CHARON model that uses triggering transitions as in the case of the three-mass example. Invariants are checked at run-time and an error is reported when an invariant is violated and no transition is enabled. Unfortunately, this is one of the few debugging features offered by the current implementation of CHARON.

The modular structure of the language is not merely syntactic, but it is exploited by analysis tools and it is supported by a formal semantics with an accompanying compositional theory of *modular refinement* [6, 7]. Compositionality is obtained by restricting the way in which a hybrid system is specified. In general, every tool that targets verification and synthesis imposes restrictions on the input specification, while more freedom is left to the designers by those tools that target simulation like SIMULINK/STATEFLOW and MODELICA.

4

Tools for Formal Verification

This section is dedicated to tools (all coming from academia) for the formal verification of hybrid systems. Formal verification is very appealing as a concept since it avoids the pitfalls of simulation that cannot guarantee design correctness. Formal verification is intended to *prove* that some properties hold for all admitted modes of operation of the system under analysis. Its power is limited by the complexity of the analysis that grows very large as the size of the system increases beyond fairly simple designs. The best way to use formal verification is by leveraging abstraction to build models that have few variables but do not lose the accuracy necessary to model the phenomena of interest.

Formal verification amounts to an intelligent exhaustive search in the input space of the designs. Intelligence lies in the exploration mechanisms and in avoiding searches in uninteresting parts of the input space. Formal verification allows one to identify errors by backtracking mechanisms in the search space that provide an example of faulty behavior and that can be used to debug the system.

For dynamical systems, safety properties [123] are the easiest to check. Safety is related to the impossibility of the system to enter a "bad" set of states. To check for this condition, all the formal verification tools reviewed here use some sort of *reachability* analysis, i.e., they identify the set of states that can be reached from a set of initial conditions under a set of allowable inputs.

We divide the tools into two bins:

- Formal verification approaches that give an answer to the question whether the system is safe:
  - HyTech, the first tool to be developed for the formal verification of a class of hybrid systems.
  - Masaccio, a language that was developed by the same investigators as HyTech, but that addresses a very important topic in formal methods: *compositionality.* Compositionality makes it possible to infer properties of an ensemble from the properties of its components, thus decreasing the complexity of the overall analysis.
  - CheckMate, developed at CMU, is likely to be the most used tool for formal verification of hybrid systems, albeit it is no longer supported. One of the most interesting features of CheckMate is its input language, a subset of the Simulink language, hence offering a nice environment where simulation, carried out with a popular tool, and formal verification can go hand-in-hand.
  - PHAVer, a tool for the safety verification of hybrid systems with piecewise-constant bounds on the derivatives. PHAVer uses exact arithmetic whose robustness is guaranteed by the use of the Parma Polyhedral Library [25].
  - HSolver, a tool for the safety verification of hybrid systems [147] developed at the Max-Planck-Institut für Informatik in Saarbrücken, Germany. HSolver uses the general idea of reducing the infinite state space of a hybrid system to a finite one by partitioning the continuous space into boxes.

  - Toolboxes that have been recently developed based on the use of the *ellipsoidal calculus* to compute approximations of continuous sets.

- the other deals with tools that can synthesize controllers that keep the system from entering the set of bad states:
  - HYSDEL: this tool is appealing since it is based on well-developed piecewise-linear techniques and mathematical programming. However, HYSDEL requires an initial discretization step left to the user that converts continuous dynamics into a discrete one. The discretization step requires choosing the sampling time that has to be selected depending on the *fastest* dynamics of the system even if in some region the system evolution is much slower.
  - d/dt even though it uses the most advanced techniques known today, still suffers from limitations in expressive power and high complexity.

An excellent review of the state-of-the-art tools for formal verification of hybrid systems was published in 2001 by Silva et al [152]. The tools reviewed included HYTECH, CHECKMATE, d/dt, UPPAAL (an integrated tool environment for modeling, validation and verification of real-time systems that are modeled as networks of timed automata, extended with data types such as bounded integers, arrays, etc.) [32, 55], and VERDICT (a framework for the modeling of hybrid systems described as Hybrid Condition/Event Systems (HCES) in the CELESTE language that provides translation from CELESTE to the input format of other tools such as HYTECH) [159]. These tools were compared and analyzed using a simple digital-control problem: a chemical batch reactor that became the workhorse example for formal verification. The comparison was made on the basis of expressive power and features such as the capability of running simulations, the possibility of specifying constraints in temporal logic and the presence of a graphical user interface. The paper contains also a section discussing the features that these tools must offer in order to reach industrial success for the design of embedded control systems. In particular the authors advocate that

- developers of formal verification tools enable the reuse of existing models of plant and controllers;
- tools for interactive model building and analysis interpretation be provided since, as we also argued, complexity can be beaten only by using appropriate abstractions of detailed models;
- aids be given to translate informal requirement specifications into formal specifications, since formal specifications are quite difficult to write for practicing engineers.

We agree with most of the conclusions of the authors and we chose not to repeat their analysis. Our review focuses on *bona fide* hybrid-system tools. Hence, we do not consider all timed-automaton-only tools such as UPPAAL, Kronos (a model checker for hybrid automata) [41, 57] and Taxys (an extension of Kronos with a compiler for programs written in the synchronous language Esterel) [35, 50]. Since we focus in this section on environments that offer verification algorithms, we also excluded Verdict from consideration.

Among the tools that do not address directly hybrid system, we would like to mention a very interesting Matlab toolbox for reachability analysis that may have an important impact on hybrid systems in the near future if the authors extend it accordingly: MATISSE [72, 73, 135]. Given a constrained linear system, MATISSE computes a lower dimensional approximation of the system, and, unique in this respect, it provides error bounds using an approximate bisimulation relation that captures the most significant characteristics of the system dynamics. The precision of the bisimulation provides a bound of the distance between the trajectories of the system and of its abstraction. MATISSE checks if the distance of the unsafe set from the reachable set of the abstraction of the system is greater than the precision of the approximate bisimulation. If that is the case, then the original system is safe. Our hope is that MATISSE will provide significant computational advantages for reachability analysis for hybrid systems using approximations while guaranteeing the accuracy of the final results, a sound approach indeed.

We tried several formal verification tools on realistic hybrid examples in the automotive domain. We have concluded that without significant effort in abstraction and modeling, the tools would simply not be adequate for industrial strength examples. Much research is needed to bring the tools and the frameworks to a degree of maturity that will make them usable by design engineers.

4.1 Introduction to verification methods

A simulator for hybrid systems solves the following problem: given an initial discrete location (or state) and an initial value for the continuous variables, compute a temporal sequence of hybrid states that complies with the specification of the system and its semantics. At each point in time, a simulator computes one location and one value for all the variables. In presence of non-determinism or uncertainty, a simulator has to make a choice in order to produce a unique value. For deterministic systems, and for a unique (or a limited set) of initial condition, simulation could be a good analysis tool. In many cases, the initial condition belongs to a set and simulating the system for all possible initial conditions is not possible. Moreover, due to abstraction and parameters that are not known in the early design stage, the system is non-deterministic. Simulation is not the right tool to use for analysis in these cases because the ability to discover corner cases is left to the experience of the designer. One would like to know if it is possible, for any of the system behavior, to reach a state in the system that leads to undesirable events. This requires to check whether a hybrid state is reachable for all initial conditions and all possible choices of non-deterministic values.

The *reachability problem* can be stated as follows (and its formulation is independent of the discrete, continuous or hybrid nature of the system): given two states σ and σ' of a system, is σ' reachable from σ?

For discrete time systems, the reachability problem has been extensively investigated. There is a conspicuous set of powerful tools for verification of discrete systems like SMV [42] and SPIN [100].

Verification for continuous and hybrid systems is particularly challenging because the reachable set of states is uncountable. Continuous variables, in fact, range over intervals of real numbers. As in the case of discrete systems, where reachable states are implicitly represented for example using binary decision diagrams, a suitable representation for sets of states has to be chosen. Such representation must be compact and have an efficient implementation. The choice depends on many factors, but the most important are the complexity of the operations to be performed on sets of states and the memory space needed to store the representation.

Consider affine hybrid systems. The dynamics, in each discrete location l, are equations of the form $\dot{x} = A_l x + B_l u$. Let l_0 be the initial location and $X_0 \subseteq \mathbb{R}^n$ be the set of initial states for the continuous vector of n variables x. Intuitively, one would let time elapse while in location l_0 and compute the set of reachable states until the invariant $Inv(l_0)$ is violated. In order to compute such a set, one has to be able to perform the following sequence of operations:

(1) rotate a set to compute $X' = \{A_{l_0} x | x \in X_0\}$;
(2) compute the geometric sum of two sets $X'' = X' + \{B_{l_0} u | u \in U\}$;
(3) perform the intersection $X''' = X'' \cap Inv(l_0)$;
(4) check if X''' is empty.

Once the set of reachable states has been computed in one location, it has to be intersected with the guards of the outgoing transitions to determine the reachable locations.

The complexity of the four operations on sets introduced above depends on how such sets are represented. While various representations based on different geometric primitive objects are possible, the two most important ones are based on polyhedra (e.g., [13] and [94]) and ellipsoids [39, 111]. Depending on the dynamics of a system, the reachable set can be represented exactly using unions of polyhedra (as in the case of constant rate systems) or it can just be over-approximated.

Consider the case where we want to check if a system can reach a state that belongs to a set of bad states S_{bad}. This problem can be solved by computing the reachable set R and checking if $R \cap S_{bad} \neq \emptyset$.

For general dynamics, however, we can only compute $R' \supseteq R$, an over-approximation of the reachable set. Consequently, if the verification result is that the over-approximated system is safe then we can also claim that the system is safe because $R' \cap S_{bad} \neq \emptyset \Rightarrow R \cap S_{bad} \neq \emptyset$. If, instead, we determine that the over-approximated system is not safe, then we cannot make any claim on the safety of the actual system and the over-approximation must be refined in order to improve its accuracy. Unfortunately, for general dynamics the reachability problem is undecidable [96], therefore a verification algorithm based on successive refinement is not guaranteed to terminate.

4.1.1 The Full-Wave Rectifier Revisited

In this section we revisit the full-wave rectifier example already introduced in Section 2.2. We want to verify that for a given input voltage $v_{in} = A \cdot \sin(2\pi f_0 t)$ with $A \approx 4V$ and $f_0 \approx 50Hz$, and an initial condition $v_{out}(0) \approx 4V$, at any time t the output voltage $v_{out}(t)$ does not drop below a certain threshold v_{min}.

Since most of the verification tools only allow linear dynamics, we use a second order differential equation to model the sinusoidal input. Also, we use two state variables x_0 and x_1 such that:

$$\begin{pmatrix} \dot{x}_0 \\ \dot{x}_1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -(2\pi f_0)^2 & 0 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix}$$

with initial conditions $x_0 = -A/(2\pi f_0)$ and $x_1 = 0$. The solution of this system gives $x_1 = A \cdot \sin(2\pi f_0 t)$. The uncertainty on the oscillation frequency translates into an uncertainty on the initial condition and the system matrix. The uncertainty on the amplitude translates into an uncertainty on the initial condition only.

If we model the sinusoidal input, the system becomes autonomous. Even if some of the tools also allow the specification of bounded inputs, we explicitly model the voltage source.

We also eliminate the *onon* discrete state. This choice is motivated by the fact that, in order to have both diodes on, we must have $v_{in} \geq v_{out} \wedge -v_{in} \geq v_{out}$ which implies $v_{out} \leq 0$ that, in our circuit, is never true. The hybrid automaton that models the full-wave rectifier is shown in Figure 4.1 where we renamed v_{in} to x_1 and v_{out} to x_2.

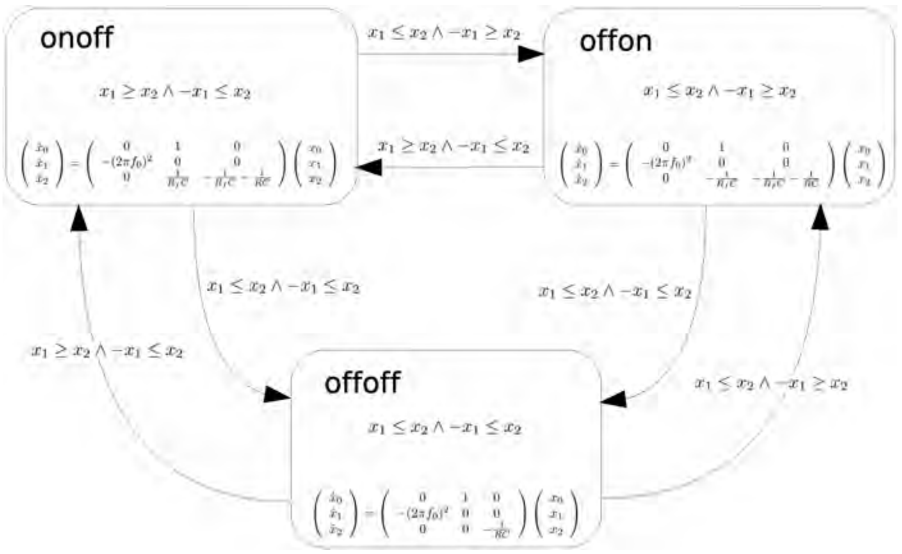
onoff
$x_1 \geq x_2 \wedge -x_1 \leq x_2$
$\begin{pmatrix} \dot{x}_0 \\ \dot{x}_1 \\ \dot{x}_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ -(2\pi f_0)^2 & 0 & 0 \\ 0 & \frac{1}{R_f C} & -\frac{1}{R_f C} - \frac{1}{RC} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix}$
$x_1 \leq x_2 \wedge -x_1 \geq x_2$
$x_1 \geq x_2 \wedge -x_1 \leq x_2$
offon
$x_1 \leq x_2 \wedge -x_1 \geq x_2$
$\begin{pmatrix} \dot{x}_0 \\ \dot{x}_1 \\ \dot{x}_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ -(2\pi f_0)^2 & 0 & 0 \\ 0 & -\frac{1}{R_f C} & -\frac{1}{R_f C} - \frac{1}{RC} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix}$
$x_1 \leq x_2 \wedge -x_1 \leq x_2$
$x_1 \leq x_2 \wedge -x_1 \leq x_2$
$x_1 \geq x_2 \wedge -x_1 \leq x_2$
$x_1 \leq x_2 \wedge -x_1 \geq x_2$
offoff
$x_1 \leq x_2 \wedge -x_1 \leq x_2$
$\begin{pmatrix} \dot{x}_0 \\ \dot{x}_1 \\ \dot{x}_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ -(2\pi f_0)^2 & 0 & 0 \\ 0 & 0 & -\frac{1}{RC} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix}$

Fig. 4.1 Hybrid automaton of the revisited full-wave rectifier.

4.2 Hytech

HyTech is a symbolic model checker for *linear hybrid automata*, a subclass of hybrid automata that can be analyzed automatically by computing with polyhedral state sets [10, 11, 87, 88, 95]. The development of HyTech, a joint effort by T. Henzinger, P. Ho and H. Wong-Toi, went through three phases [88]. The earliest version of HyTech, developed at Cornell University, was built on top of the commercial tool Mathematica [173] and linear predicates were represented and manipulated as symbolic formulas [10]. Based on the observation that a linear predicate over n variables defines a union of polyhedra in R^n, the second generation of HyTech [87] combined a Mathematica main program with calls to a library of efficient routines for polyhedral operations [83]. This change provided a speed-up of one order of magnitude with respect to the first prototype. The third generation of HyTech is a fully-rewritten C++ program that is two to three orders of magnitude faster than the previous one. This implementation of HyTech, however, uses exact arithmetic and suffers from overflow errors. It's successor, HyperTech, uses interval arithmetic and is able to model more complicated dynamics. A detailed guide to the last version as well as to HyTech-related papers is given in [88].

HyTech takes two inputs: a hybrid system description and a set of analysis commands. The hybrid system, which is modeled as a collection of linear hybrid automata, is specified textually using the HyTech system description language [88]. A linear hybrid automaton consists of a finite control graph whose nodes are called *control modes* and whose edges are called *control switches* together with a set X of continuous variables. The continuous dynamics within each control mode are subject to a constant polyhedral differential inclusion, while the discrete dynamics are modeled by the control switches each of which has a guard condition and a reset condition over X. A state is a pair consisting of a control mode and a vector of variable values. Before drafting the textual description, the users of HyTech must (1) identify the concurrent components of the system (and the communication mechanisms between the components), (2) model each component using a hybrid automaton, and (3) conservatively approximate each hybrid automaton with a linear hybrid automaton. While (1) and (2) are common to most of the tools that we take into account (if they support composition), the last step is required in order to model complex continuous dynamics using linear dynamics. HyTech processes the textual specification and derives a model for the whole system by performing a parallel composition as the product of these automata. The analysis commands are given using a simple command language that allows the specification of iterative programs for performing formal verification tasks such as reachability analysis and error-trace generation.

4.2.1 HyTech **Syntax**

HyTech models a hybrid systems as the parallel composition of linear hybrid automata (LHA). A LHA uses an ordered set $X = \{x_1, x_2, ..., x_n\}$ of real-valued variables to model continuous activities. All variables in the system are global and declared at the beginning of a hybrid system description and can be of type `discrete`, `clock`, `stopwatch`, `parameter`, or `analog`. A valuation $\mathcal{V}(X)$ is a function that associates a point in $\mathbb{R}^n$ to X. A *linear expression* over X is a linear combination of variables in X with rational coefficients. A *linear inequality* is an inequality between two linear expressions and a *convex predicate* is a finite conjunction of linear inequalities. A *predicate* is a finite disjunction of convex predicates, defining a set of valuations.

An linear hybrid automaton is defined by a set of discrete states or locations, initial conditions, invariant conditions, transitions and rate conditions where:

- *locations* are control modes that are used to define the discrete states of the automaton. Let $V = \{v_1, v_2, ..., v_l\}$ be the set of locations;
- the *initial condition* is a predicate over X;
- *invariant conditions* are convex predicates over X. For a location v, $inv(v)$ is the invariant associated with that location;
- *transitions* are labeled edges between locations. Let $E \subseteq V \times V$ be the set of edges. An edge is labeled with an *update set* and a *jump condition*. The update set Y is a subset of X and the jump condition is a convex predicate over $X \cup Y'$, where primed variables refers to the value of the variables after the transition. For a transition $e = (v_i, v_j)$ from location v_i to location v_j, label $act(e)$ denotes the condition associated to the transition;
- *rate conditions* are convex predicate over $\dot{X}$ where for a variable $x \in X$, $\dot{x} \in \dot{X}$ denotes the rate of change of x. For a location v, $dif(v)$ is the rate condition associated to that location;
- *synchronization labels* is a finite set L of labels. A labeling function *syn* assigns a subset of labels from L to each edge. Synchronization labels are used to compose automata.

Commands are built using objects of two basic types: *region expressions* for describing regions of interest, and *boolean expressions* that are used in the control of command statements. Regions may be stored in variables, provided the region variables are declared via a statement such as

```
var  init_reg,  final_reg:  region;
```

which declares two region variables called init_reg and final_reg. HyTech provides a number of operations for manipulating regions, including computing the reachable set, successor operations, existential quantification, convex hull, and basic boolean operations. For added convenience, there are built-in macros for reachability analysis, parametric analysis, the conservative approximation of state assertions [86], and the generation of error trajectories.

Parametric Analysis. An important feature of HyTech is the ability to perform parametric analysis, i.e., to determine the values of design parameters for which a linear hybrid automaton satisfies a temporal-logic requirement. With parametric analysis, model checking can be used to go beyond the mere confirmation that a system is correct with respect to certain requirements. While completing the specification of a system, the users can decide to introduce some parameters as symbolic constants with unknown, fixed values. These values will be defined only later at the design implementation stage. Meanwhile, parametric analysis makes it possible to determine necessary and sufficient constraints on the parameters under which safety violations cannot occur. Common uses for parametric analysis include determining minimum and maximum bounds on variables, and finding cutoff values for timers and cutoff points for the placement of sensors.

4.2.2 HyTech Semantics

At any time instant the state of a hybrid automaton is defined by a control location and a valuation of all variables in X. The state can change because of a location change or because time elapses. A *data trajectory* (δ, ρ) of a linear hybrid automaton consists of a non-negative duration $\delta \in \mathbb{R}_{\geq 0}$ and a differentiable function $\rho : [0, \delta] \rightarrow \mathbb{R}^n$. A data trajectory (δ, ρ) is a v-trajectory for a location v, if for all reals $t \in [0, \delta]$, $\rho(t)$ satisfies $inv(v)$ and $\dot{\rho}(t)$ satisfies $dif(v)$. A trajectory of a hybrid automaton is an infinite sequence:

$$(v_0, \delta_0, \rho_0) \rightarrow (v_1, \delta_1, \rho_1) \rightarrow (v_2, \delta_2, \rho_2) \rightarrow ...$$

of locations v_i and v-trajectories (δ_i, ρ_i) such that $\forall i \geq 0$, there is a transition $e_i = (v_i, v_{i+1})$ and $(\rho_i(\delta_i), \rho_{i+1}(0))$ satisfies $act(e_i)$.

A hybrid system is modeled in HyTech as a composition of linear hybrid automata that coordinate through variables and synchronization labels. Let $A_1 = (X_1, V_1, inv_1, dif_1, E_1, act_1, L_1, syn_1)$ and $A_2 = (X_2, V_2, inv_2, dif_2, E_2, act_2, L_2, syn_2)$ be two linear hybrid automata of dimension n_1 and n_2, respectively. The product $A_1 \times A_2$ is a linear hybrid automaton $A = (X_1 \cup X_2, V_1 \times V_2, inv, dif, E, act, L_1 \cup L_2, syn)$ such that:

- for each location $(v_1, v_2) \in V_1 \times V_2$, $inv(v_1, v_2) = inv_1(v_1) \wedge inv_2(v_2)$ and $dif(v_1, v_2) = dif_1(v_1) \wedge dif_2(v_2)$;
- E contains the transition $e = ((v_1, v_1'), (v_2, v_2'))$ if and only if
  (1) $v_1 = v_1'$ and there is a transition $e_2 = (v_2, v_2') \in E_2$ with $L_1 \cap syn(e_2) = \emptyset$; or
  (2) there is a transition $e_1 = (v_1, v_1') \in E_1$ with $syn(e_1) \cap L_2 = \emptyset$, and $v_2 = v_2'$; or
  (3) there are transitions $e_1 = (v_1, v_1') \in E_1$ and $e_2 = (v_2, v_2') \in E_2$ such that $syn(e_1) \cap L_2 = syn(e_2) \cap L_1$.

  In case (1), $act(e) = act_2(e_2)$ and $syn(e) = syn_2(e_2)$. In case (2), $act(e) = act_1(e_1)$ and $syn(e) = syn_1(e_1)$. In case (3), $act(e)$ has the update set equal to $Y_1 \cup Y_2$, the jump condition that is the conjunction of the jump conditions, and $syn(e) = syn(e_1) \cup syn(e_2)$.

Symbolic Model Checking. Model checking-based formal verification is performed by considering the state space of the system model and automatically checking it for correctness with respect to a requirement expressed in temporal logic [48]. In particular, *symbolic model checking* makes it possible to do so more efficiently by using constraints that represent state sets, thereby avoiding the full enumeration of the entire state space [42, 49, 136]. Whenever a system fails to satisfy a temporal-logic requirement, a model checking tool generates an error trajectory, i.e., a time-stamped sequence of events that leads to the requirement violation. This is an important feature because designers can use error trajectories for debugging the system. The model-checking approach has been extended to several classes of infinite state-transition systems, including *timed automata* [1, 90]. The symbolic representation of state sets is necessary for timed automata due to the presence of real variables that have infinite domains.

Timed Automata and Linear Timed Automata. With symbolic model checking, timed automata can be effectively analyzed by manipulating sets of linear constraints. For timed automata, these linear constraints are typically disjunctions of inequalities whose components are bounded, e.g., $x - y \leq b$ where x, y are real vectors and b is a constant integer vector. Timed automata have a finite bisimilar quotient meaning that it is possible to partition the state space in a finite number of regions and obtain a finite transition system where transitions are in bijection with transitions of the original system. Therefore, the quotient system is safe if and only if the original system is safe. This property allows one to perform verification on a finite automaton. *Linear hybrid automata* [11] are an extension of timed automata where the linear constraints can be disjunctions of inequalities of the form $A\dot{x} \leq c$ where A is a constant matrix and c a constant vector. The consequence of this extension, however, is that the bisimilar quotient transition system could have an infinite number of states. Therefore, model checking is no longer guaranteed to terminate. Still termination occurs often in practice and, when it does not, it can be enforced by considering the system behavior over a bounded interval of time [95].

Linear hybrid automata are more expressive compared to other formalisms for which model checking is possible, such as finite automata and timed automata. That notwithstanding, there are still many embedded applications that do not satisfy the linearity requirement. In these cases, it is possible to derive a *conservative approximation* of the system in terms of linear hybrid automata, so that the satisfaction of the correctness requirement by the approximated model guarantees the correctness of the original system as well [89]. On the other hand, when the approximate system violates the requirement it is necessary to (1) check if the generated error trajectory belongs to the original system and (2) refine the approximation whenever this is not the case.

4.2.3 Examples

In order to model the full-wave rectifier in HyTech we have to approximate its behavior. The approximation is required because the circuit dynamics cannot be written as convex predicates on $\dot{X}$. For instance, when diode D_1 is in the on state and D_2 is in the off state, the dynamics describing the continuous evolution of the output voltage is $\dot{v}_{out} = (v_{in} - v_{out})/(R_f C) - v_{out}/(RC)$ that is a linear expression over both the variables and their first derivatives.

We approximate the circuit as follows. The sinusoidal voltage source is approximated by a triangular voltage source as shown in Figure 4.2. Between the two bounds, we select the one that is indicated as `lower` in the figure.

The state variables are the input voltage v_{in}, the output voltage v_{out} and a clock variable p that is used to switch the voltage source between positive and negative first derivative.

```
var
  x, – vin
  v : analog; – vout
  p : clock;
```

The voltage source is described by the following automaton:

```
automaton voltagesource
synclabs:;
initially up & x = -4 & p = 0;
```

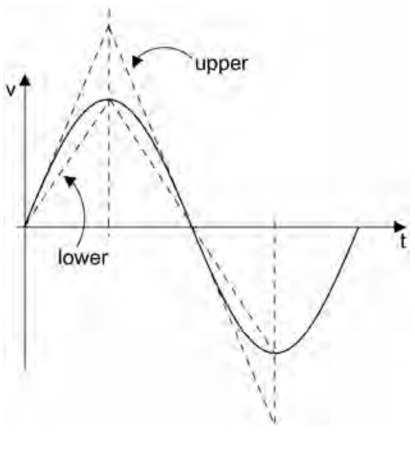

Fig. 4.2 Upper and lower bound lines for the approximation of the sinusoidal voltage source.

```
loc up: while p < 1/100 wait { dx = 800 }
  when p= 1/100 do { p'= 0} goto down;
loc down: while p< 1/100 wait { dx = -800 }
  when p= 1/100 do { p'= 0} goto up;
end –voltagesource
```

the clock variable p switches the sign of the derivative every half period.

The rest of the circuit is modeled by the following automaton:

```
automaton circuit
synclabs:;
initially offoff & v = 4;
loc onoff: while x >= v & v + x >= 0 wait { dv= 800 }
  when x <= v & v + x <= 0 do { v' = v } goto offon ;
  when x <= v & v + x >= 0 do { v' = v } goto offoff ;
loc offon: while x <= v & x + v <= 0 wait { dv= 800}
  when x >= v & v + x >= 0 do { v' = v } goto onoff ;
  when x <= v & v + x >= 0 do { v' = v } goto offoff ;
loc offoff: while x <= v & v + x >= 0 wait { dv= −40}
  when x >= v & v + x >= 0 do { v' = v } goto onoff ;
  when x <= v & v + x <= 0 do { v' = v } goto offon ;
end –circuit
```

We have done two approximations:

- we have considered the diodes to be ideal, i.e., with $R_f = 0$, therefore the capacitor charges at the same rate as the input;
- in the *offoff* state we consider a discharge current equal to the maximum current of $4/(R * C)$ (where $4V$ is the peak voltage, $R = 1K\Omega$ and $C = 100\mu F$).

The hybrid system description is followed by the analysis description:

```
var init_reg, final_reg, reached: region;
init_reg := loc[voltagesource]=up & x=-4 & p=0 & loc[circuit]=offoff & v=4;
final_reg := loc[circuit] = offoff & v¡=4-1/2;
reached := reach forward from init_reg endreach;
print reached;
if empty(final_reg & reached)
  then prints "Rectifier is SAFE";
  else prints "Rectifier is UNSAFE";
endif;
```

The analysis section declares the initial set as a region init_reg defined by the initial discrete locations for the automaton and the values for the variables. The initial set is the conjunction of discrete locations and polyhedral regions in the state variables space. In the analysis commands, the symbol & refers to set intersection $\cap$. The verification checks whether the output voltage drops below $3.5V$ and also prints the set of reachable states. The output looks as follows:

```
Location: down.offoff
     x + 800p = 4   & 20v = x + 76   & x <= 4   & 21x + 76 >= 0
|
     x + 800p = 4   & x + v = 0   & x + 4 >= 0   & 0 >= 21x + 76
Location: down.offon
     x + 800p = 4   & x + v = 0   & x + 4 >= 0   & 0 >= 21x + 76
Location: down.onoff
     x = 4   & v = 4   & p = 0
Location: up.offoff
     800p = x + 4   & x + 20v = 76   & x + 4 >= 0   & 21x <= 76
|
     800p = x + 4   & v = x   & x <= 4   & 21x >= 76
Location: up.offon
     x + 4 = 0   & v = 4   & p = 0
Location: up.onoff
     800p = x + 4   & v = x   & x <= 4   & 21x >= 76
Rectifier is SAFE
```

The system satisfies the property. For each location, the set of reachable states is reported as a disjunction of convex polyhedra (described as a conjunction of inequalities).

HyTech also supports parametric analysis. Parameters cannot be used in the definition of the dynamics. In our case, this means for example that it is not possible to directly parameterize the load resistor. It is possible, though, to define different locations for the $offoff$ state each with a different discharge rate and check the safety property for a discrete number of possible loads.

HyTech supports differential inclusions. It would be possible, for instance, to define the input voltage rate condition to be an inclusion like dx in [800,900]. Unfortunately the exact arithmetic used by HyTech leads to an overflow error.

4.2.4 Discussion

HyTech can efficiently analyze systems modeled with linear hybrid automata, either directly or through conservative approximations.

HyTech uses exact arithmetic that gives exact answers to the reachability question. On the other hand, it is difficult to find accurate enough polyhedral abstractions for many systems without computational bottlenecks. A detailed discussion of some of the lessons learned from developing HyTech is provided by its authors in [91].

HyTech users must minimize the number of continuous variables in their models and avoid models whose neighbouring control modes present very different rate conditions. In other words, HyTech is better suited to high-level system descriptions where the continuous variables have either simple dynamics or can be adequately abstracted to ones with simple dynamics, e.g., rate-bounded systems. Parametric analysis with a limited number of parameters is reported to be often successful, but the analysis of systems with complex relationships between multiple parameters and timing constants generally leads to arithmetic overflow, due to the implementation of the solution algorithms. In practice, users must use HyTech iteratively to refine their model by further abstracting each system component or merging multiple components into a single one. As recognized by its authors, "it is a fine art to choose a level of abstraction that is simple enough for HyTech to complete and yet accurate enough for properties to be proven" [91]. We found that this statement applies not only to HyTech but also to all formal verification frameworks we have worked with.

4.3 Masaccio

As we have already seen in the previous sections, the concept of hierarchy for the specification of complex systems is quite consolidated. We now point our attention to the possible ways of nesting components in hierarchical systems, since -as we will see below- Masaccio offers the greatest flexibility in this sense.

We already mentioned the concurrent and sequential hierarchies of some modeling tools like Statecharts [84], UML [38] and Ptolemy [56]. Other languages, like Charon, also address specifically the issue of hierarchical modeling for hybrid systems. However, all these modeling formalisms focus on simulation rather than formal analysis.[1]

[1] Actually, as discussed in Section 3.6, some latest results allow some kind of formal analysis also in Charon.

Tools that support compositional verification are some variants of Statecharts, hierarchical modules and hybrid I/O automata. Statecharts has been extended in [167] with variants that allow compositional verification, but still suffers from some major limitations, most notably the absence of support for *assume-guarantee reasoning.* Hierarchical modules [6] provide both serial and parallel composition and support assume-guarantee, but components can be only discrete, thus there is no way of characterizing continuous-time behavior. On the other hand, hybrid I/O automata [130] can also model continuous-time components but serial composition is not supported.

Masaccio is a modeling formalism for compositional verification of hybrid systems that goes a step further. Hybrid systems described in Masaccio result from a hierarchical specification made of *components* [93, 97]. Masaccio supports both discrete and continuous time components that can be arbitrarily nested and composed via both parallel and serial operators. Moreover, Masaccio offers support for assume-guarantee reasoning, a compelling example of which is provided in [97].

4.3.1 Masaccio **Syntax**

Hybrid systems in Masaccio are built out of *components* that are defined in terms of interfaces (describing the syntactic structure) and executions (defining the semantics). The *interface* of a component A consists of:

- A finite set V_A^i of input variables.
- A finite set V_A^o of output variables.
- A dependency relation $\prec \subseteq V_A^i \times V_A^o$ between input/output variables.
- A finite set L_A of interface locations. Locations are points through which the control flow enters/exits the component.

For variables, the following condition must hold: $V_A^i \cap V_A^o = \emptyset$. The state of component A is an assignment of values to the set of variables

$V_A = V_A^i \cup V_A^o$. All variables in MASACCIO are *typed*, so assignment must be consistent with variable types. The set of all possible state assignments to the variables in V_A is denoted by $[V_A]$.

The meaning of the dependency relation is the following: assume $x \prec y$, then the value of y depends without delay on the value of x. Specifically, for jumps, the value of y after the discrete transition takes place depends on the value of x also after the jump. For flows, the value of the derivative $\dot{y}$ depends instantaneously on the value of $\dot{x}$. MASACCIO requires that the dependency relation be acyclic in order to guarantee the existence of input/output values (for jumps) or curves (for flows). This condition may seem too restrictive, since input/output values or curves can exist also if some cyclic dependency exists, but has the obvious advantage of avoiding expensive fixed-point calculations. This eliminates some of the potential sources of non-determinism in the behavior of the hybrid systems.

For each location $a \in L_A$, the interface specifies a *jump entry condition* $\Psi_A^{jump}(a)$ and a *flow entry condition* $\Psi_A^{flow}(a)$. The component can be entered through a given (jump or flow) location if the corresponding entry condition is satisfied by the current I/O state. Control can exit the component at any location. Typically, exit points are locations with unsatisfiable entry conditions. Therefore, we see that, unlike CHARON, which separates entry from exit locations, in MASACCIO there is no syntactical distinction between entry and exit points of the component.

4.3.2 MASACCIO Semantics

The semantics of MASACCIO is specified in terms of behaviors of single components. Given a generic component A its behavior is defined by a set E_A of finite executions. Infinite executions in finite time, i.e., Zeno behaviors, are not allowed in MASACCIO. Zeno behaviors have been thoroughly addressed in [92], and conditions are available for hybrid systems that prevent Zeno behavior. The user should verify whether one of these conditions apply for the description that he/she describes in MASACCIO.

An *execution* is a tuple of the form:

- (a, w, b)
- (a, w)

where $a \in L_A$ is an entry location, $b \in L_A$ is an exit location and w is a sequence of execution steps, i.e., either flows or jumps, as described below. The location a is called the *origin* of the execution, b (if present) is the *destination*, while w is called the *trace*.

A *jump* is a pair $(p, q) \in [V_A] \times [V_A]$ of I/O states; state p is called the *source* of the jump, while q is called the *sink*. A *flow* is a pair (δ, f), where δ is a non-negative number and $f : \mathbb{R} \to [V_A]$ is a function differentiable on the closed interval $[0, \delta]$. The quantity δ is called the *duration* of the flow, while $f(0)$ is the source and $f(\delta)$ is the sink. Intuitively, $f(t)$ describes the state trajectory for the whole duration of the flow. For consistency, the sink state of a step must be the same as the source state of the following step in the sequence.

Atomic components (discrete or continuous) contain only one origin and destination. Traces can only be single jumps for discrete components or single flows for continuous components. For discrete components, the allowed jumps are defined in terms of a *jump predicate*, which constrains the values of I/O states before and after a jump. Usually, such constraints are expressed in terms of difference equations. For continuous components, the allowed flows are determined by a *flow predicate*, usually defined by differential equations on I/O signals; clearly the causality property must hold: if u is the vector of input signals and y is the vector of outputs, then $u \prec y$.

The semantics of MASACCIO is made complete with the interpretation of jump or flow *entry conditions*, as explained in the previous section. We recall that executions can start only if the corresponding entry condition is satisfied, and they terminate when there is no entry condition which can be satisfied.

Generic components are defined by nested compositions of atomic components. MASACCIO supports two basic composition operators: parallel composition and serial composition.

Parallel Composition. Given two components A and B their parallel composition is denoted by $A \mid\mid B$. The corresponding execution starts at a common location in $L_A \cap L_B$, and it is synchronous for both components: every jump in A takes place at exactly the same time of a corresponding jump in B, and similarly flows in A are matched by flows in B having the same duration. MASACCIO supports preemption: when one of the two components reaches an exit location, the execution of the other component is halted and the control flow exits from $A \mid\mid B$.

Serial Composition. Serial composition represents sequencing of behaviors. Given A and B, their serial composition is denoted by $A + B$. Executions of $A + B$ are either executions of A or B. The set of control flow locations is the union of those of the two individual components, i.e., $L_{A+B} = L_A \cup L_B$. Also the set of variables is the union of the sub-components' variables: $V_{A+B} = V_A \cup V_B$. The triple (a, w, b) is an execution of $A + B$ if and only if either $(a, w[A], b)$ is an execution of A or $(a, w[B], b)$ is an execution of B.

In addition to the above operations, in MASACCIO it is possible to re-assign variables names in order to enable the sharing of information among the different components. Variables having the same names refer to the same signal. MASACCIO also supports variable hiding and location hiding to provide the language with the property of *encapsulation*. However, in order to prevent deadlocks, locations can be hidden only if their corresponding entry condition is satisfied so that the control flow can never halt at those locations. Hidden variables have local scope, meaning that their values are re-initialized each time the control flow enters into the corresponding component.

Assume-Guarantee. MASACCIO supports the technique of assume-guarantee reasoning. For this, we need to discuss the *refinement relationship* and the *compositionality properties* of the model.

Intuitively, if component A refines component B, we can think of A as being "more specific" than B; from the point of view of observational semantics, all the traces of A are also traces of B (the converse is in general not true). From an operational point of view component A may result from B by adding some constraints on it, e.g., $A = B \mid\mid C$ for some other component C. Formally, component A refines component B if the following conditions are satisfied:

(1) every input (output) variable of A is an input (output) variable of B and the dependency relation of B is a subset of the dependency relation of A. In symbols: $\prec_B \subseteq \prec_A$.
(2) every execution of $(a, w, b) \in E_A$ is such that $(a, w[B], b) \in E_B$, that is every execution of A is an execution of B provided traces are restricted to only variables belonging to B.

Compositionality means that the operators are monotonic relative to the refinement relationship. In other words, if A refines B then $A \mid\mid C$ refines $B \mid\mid C$ (for a generic component C), $A + C$ refines $B + C$ and the application of the variable renaming and variable/location hiding operators does not alter the refinement relation between components A and B.

Under these and other assumptions on the scope of the variables, Masaccio supports the assume-guarantee principle. Intuitively, one can separately verify the correctness of each component A (i.e., that A refines its specification), assuming that the rest of the components of the system behave according to their specification. Then, the correctness of the components implies the correctness of the whole system (for details see [97]). By using this technique, a large verification problem can be decomposed into many smaller verification problems, which are typically much easier to solve, as the complexity of verification grows more than linearly (often exponentially) in the size of the system. The approach, however, can only be applied under certain conditions. We refer the reader to the literature for more details [97].

4.3.3 Discussion

Masaccio is a formalism that is intended to study the theoretical implications of certain verification techniques, and therefore does not provide any practical support for the implementation of the models and for their verification. For this reason, we were unable to implement the examples in this case. The strength of MASACCIO lies in its formal definition of the semantic domain, which makes it an ideal denotational framework to develop techniques for the analysis of hybrid systems. In particular, the assume-guarantee and other compositional techniques are required, together with abstraction, to address the complexity of verification in hybrid systems.

4.4 CheckMate

CHECKMATE is a *hybrid system verification toolbox for* MATLAB that has been developed at Carnegie Mellon University. This section reviews how modeling and verification of hybrid systems is performed in this environment and is based on [151].

CHECKMATE supports simulation and verification of a particular class of hybrid dynamic systems called *threshold event-driven hybrid systems* (TEDHS) [92]. A verification procedure for these systems was proposed in [47]. In a TEDHS, the changes in the discrete state can occur only when continuous state variables encounter specified thresholds. Thresholds in the TEDHS model are hyperplanes. In the language of the general hybrid system model presented in Section 2.1, guards and invariants are linear function of states and are complementary, i.e., when invariants are not satisfied, an appropriate guard must be satisfied. This guarantees that when the system has to jump because the invariant is not satisfied at a given state, there is a transition that it can take and, therefore, the behavior is non blocking.

Hybrid system models in CHECKMATE have continuous dynamics described by standard differential state equations (possibly nonlinear), planar switching surfaces, and discrete dynamics modeled by finite state machines. The key theoretical concepts used in CHECKMATE are described in [46].

4.4.1 CHECKMATE **Syntax**

A very interesting feature of CHECKMATE is the use of standard industrial tools to enter the description of hybrid systems. CHECKMATE models are constructed using custom and standard SIMULINK and STATEFLOW blocks. The continuous state equations, parameters and specifications (the properties to be verified) are entered using the SIMULINK GUI and user-defined MATLAB *m*-files. Specifications express properties of trajectories of the CHECKMATE model. The CHECKMATE verification function determines if the given specifications are true for all trajectories starting from a polyhedral set of initial continuous states. Note that the semantics of the design must be the one understood by CHECKMATE. For this reason, the tool uses the *syntax* of the SIMULINK environment but restricts its semantics so that a formal approach can be used.

CHECKMATE models are built with the SIMULINK GUI using two customized SIMULINK blocks along with several of SIMULINK standard blocks. To build the model from scratch, the user must enter the command *cmnew* at the MATLAB command prompt. This will open the CHECKMATE library from which the user can construct the system model. Currently, the set of blocks used in CHECKMATE are:

(1) *Switched Continuous System Block (SCSB).* The custom SCSB represents a continuous dynamic system with state equation $\dot{x} = f(x,\sigma)$, where σ is a discrete-valued input vector to the SCSB and the continuous state vector x is the block's output. Currently, three types of dynamics can be specified in an SCSB for each value of the input vector σ: clock dynamics $\dot{x} = c$, where c is a constant vector, linear dynamics $\dot{x} = Ax + b$, where A is a constant matrix and b is a constant vector, and nonlinear dynamics $\dot{x} = f(x)$. The switching function is a *m*-file that provides the information about the dynamics of the system. The variable σ selects which dynamics should be used.

(2) *Polyhedral Threshold Block (PTHB).* The other custom block in CHECKMATE is the PTHB, which represents a polyhedral region $Cx \leq d$ in the continuous space of the continuous-valued input vector x. The PTHB output is a binary signal indicating whether x is inside the region or not, i.e., whether or not the condition $Cx \leq d$ is true. The initial condition, the analysis region, and the internal region hyperplane are defined as *linearcon* object.

(3) *Finite State Machine Block (FSMB).* Discrete dynamics are modeled using a FSMB. FSMBs are regular STATEFLOW blocks that conform to the following restrictions:

- no hierarchy is allowed in the STATEFLOW diagram;
- data inputs must be Boolean functions of PTHB and FSMB outputs only;
- event inputs must be Boolean functions of PTHB outputs only, i.e., events can only be generated by the continuous trajectory leaving or entering the polyhedral regions;
- only one data output is allowed;
- every state in the STATEFLOW diagram is required to have an entry action that sets the data output to a unique value for that state;
- no action other than the entry action discussed above is allowed in the STATEFLOW diagram.

Some of these restrictions are rather severe from an ease-of-use point of view. For example, hierarchy is a much used feature of STATEFLOW. Barring its use may force the designer to enter an unwieldy number of states. Event inputs are in general used to represent disturbances as well as control. Restricting events to represent jumps due to the evolution of the continuous state may again create inconveniences to the user. The other restrictions are made to guarantee deterministic execution of the hybrid automaton.

There are some parameters the user must enter in order to give CHECKMATE all the necessary details about the verification process. These parameters, as well as any variables used in the SIMULINK/STATEFLOW front-end model, are defined and stored in the MATLAB workspace. Parameters and variables can be defined manually or through the use of MATLAB *m*-files.

4.4.2 CheckMate Semantics

A threshold-event-driven hybrid system is a combination of a *switched continuous system (SCS)*, a *threshold event generator (TEG)*, and a *finite state machine (FSM)*. The SCS takes the discrete-valued input σ and produces its continuous state vector x as the output. The continuous dynamics for x evolve according to the differential equations or differential inclusions selected by the discrete input σ. The output of the SCS is fed into the TEG, which produces an event when a component of the vector x crosses a corresponding threshold from the specified direction (rising, falling, or both). The event signals from the TEG drive the discrete transitions in the FSM whose output, in turn, drives the continuous dynamics of the SCS.

CheckMate converts the TEDHS into a *polyhedral invariant hybrid automaton (PIHA)*. PIHA are a subclass of hybrid automata as presented in [92]. Recalling the definitions in Section 2.1, each discrete state in the hybrid automaton is called a location. Associated with each location is an invariant, the condition which the continuous state must satisfy while the hybrid automaton resides in that location, and the flow equation representing the continuous dynamics in that location. Transitions between locations are called edges. Each edge is labeled with guard and reset conditions on the continuous state. The edge is enabled when the guard condition is satisfied. Upon the location transition, the values of the continuous state before and after the transition must satisfy the reset condition. In general, the analysis of hybrid automata can be very difficult. In CheckMate, the attention is restricted to PIHA. A PIHA is a hybrid automaton with the following restrictions:

- the continuous dynamics for each location is governed by an ordinary differential equation (ODE);
- each guard condition is a linear inequality (a hyperplane guard);
- each reset condition is an identity;
- for the hybrid automaton to remain in any location, all guard conditions must be false. This restriction implies that the

invariant condition for any location is the convex polyhedron defined by the conjunction of the complements of the guards. This is the origin of the name polyhedral-invariant hybrid automaton.

These restrictions are needed to simplify the formal verification task and to allow the simulation of the hybrid system in SIMULINK/ STATEFLOW, but they certainly reduce the application range.

Formal Verification. In CHECKMATE, formal verification is performed by computing the set of states that are reachable given the initial conditions. Deriving the set of reachable states is computationally very hard even for linear time-invariant continuous-time systems. Hence there is a strong incentive for approximating the problem in a way that makes it computationally feasible. In CHECKMATE, formal verification is performed using finite-state approximations known in the literature as *quotient transition systems* [122]. A quotient transition system (QTS) is a finite state transition system that is a conservative approximation of the hybrid system. The states of a QTS correspond to the elements of a partition of the state space of the hybrid system. There is a transition between two states P_1 and P_2 of the QTS if and only if there is a transition between two states $p_1 \in P_1$ and $p_2 \in P_2$ in the original hybrid system. Thus, for every trajectory in the original hybrid system there is a corresponding trajectory in the QTS. However, the converse is not true, i.e., there may be trajectories in the QTS that do not correspond to any trajectory in the original hybrid system. The approximation is conservative in the sense that it captures all possible behaviors of the hybrid system, and possibly more. Therefore, if all trajectories in the QTS satisfy some property, then we can conclude that all trajectories in the hybrid system also satisfy the same property. If a negative result is found (the property is not verified), the verification of the original hybrid system is inconclusive and the user is given the option to refine the current approximation and attempt the verification again.

CHECKMATE only pays attention to the behavior of the hybrid system at the switching instants. Thus, CHECKMATE approximates the

QTS for the hybrid system from the partition of the switching surfaces, which are the boundaries of the location invariants in the PIHA, and the set of initial continuous states.

The verification method in the QTS is based on reachability analysis and, therefore, requires a very expensive computation for continuous-time dynamical systems. To reduce the computational complexity, reachability analysis is not performed on the original system, but using an approximation method called *flow-pipe approximation* [47]. The flow-pipe approximation is used to define transitions in the quotient transition system for the PIHA as follows. A state in the quotient transition system is a triple (π, p, q) where π is a polytope in location (p, q) of the PIHA. For each state in the quotient transition system, the flow pipe is computed for the associated polytope under the associated continuous dynamics. The mapping set, i.e., the set of states on the invariant boundary that can be reached from π, is computed. A transition is then defined from (π, p, q) to any other state whose polytope overlaps with the mapping from π. CheckMate then performs model checking on this transition system to obtain a verification result for the desired specification. If the verification returns a positive result, then the program informs the user and terminates. If a negative result is returned, then the user is given the option of quitting or allowing CheckMate to refine the approximation and repeating the verification. This process continues until a positive verification result is obtained, or the user decides to quit.

4.4.3 Examples

The three-mass system has no inputs and is characterized by twelve state variables: vertical positions, vertical velocities, horizontal positions and horizontal velocities. The block diagram for the three-mass system is shown in Figure 4.3. The switched continuous system, the PTHB blocks and the finite state machine are provided as a CheckMate block-set in Simulink. A PTHB block has a Polyhedron parameter that must be set to a variable defined in the Matlab workspace. Such a variable is defined by calling the function linearcon(CE,dE,CI,dI) provided with the CheckMate package. The linearcon function generates a data

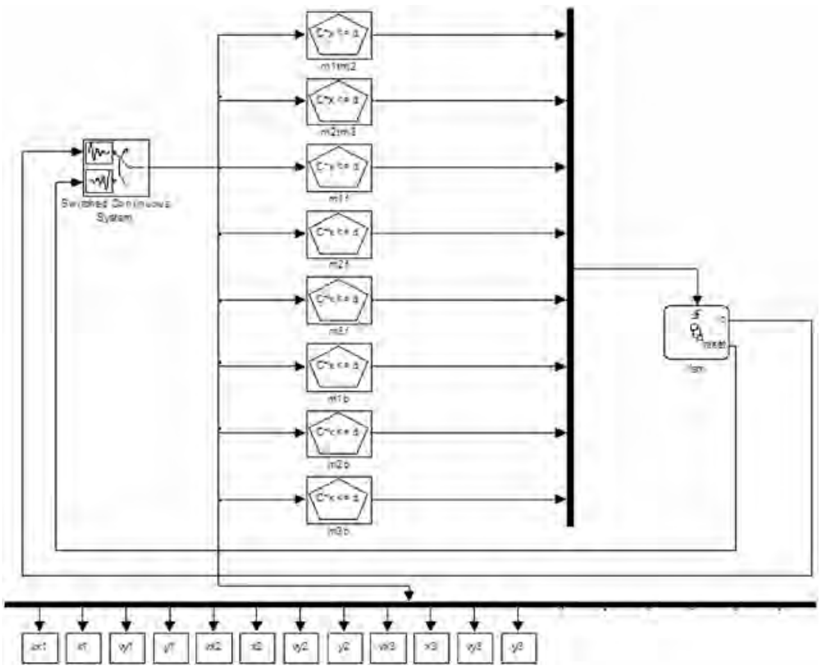

Fig. 4.3 Checkmate model of the three-mass system (block diagram).

structure that represents the set of linear constraints $CE = dE$ and $CI \leq dI$. The output of a PTHB block is equal to zero when such constraints are violated and equal to one when they are satisfied.

Figure 4.4 shows the state machine that represents the discrete part of the model. Arcs are labeled by events that are generated on the raising edge of the output of PHTB blocks. Each state is encoded with an integer. When the discrete automaton enters a state, it outputs the integer that corresponds to that state. The state number is used by the SCSB that is linked to a MATLAB function. Depending on the state, the MATLAB function selects a corresponding dynamical system and a set of reset maps. The simulation results are shown in Figure 4.5. The simulation engine in CHECKMATE based on SIMULINK/STATEFLOW cannot correctly simulate the critical case when $x_{3,0} = L$. This gives even more strength to the argument that model checkers are indispensible to verify correctness of hybrid systems. The figure also shows

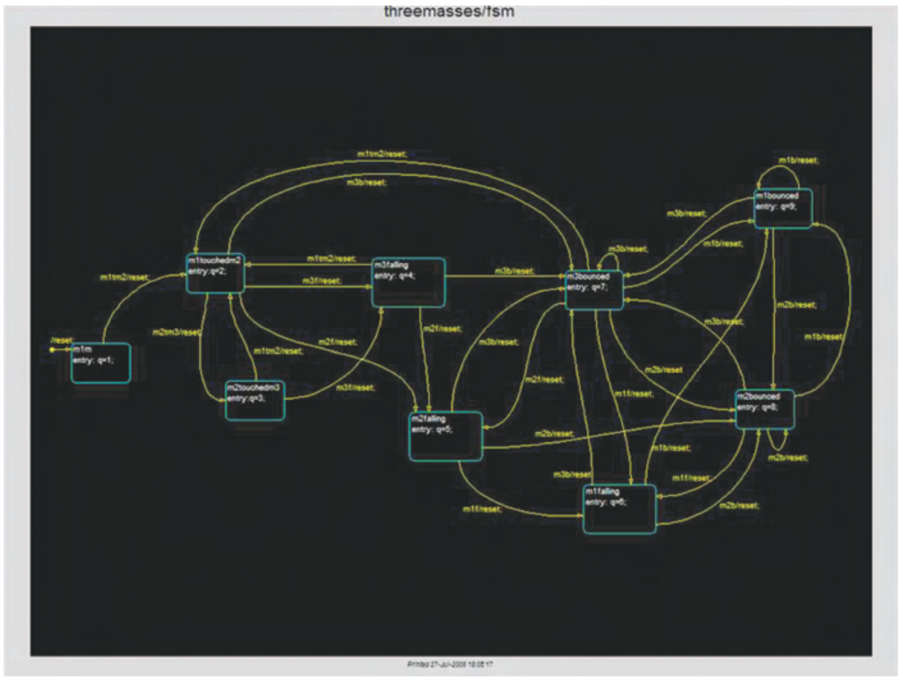

Fig. 4.4 Checkmate model of the three-mass system (finite state machine).

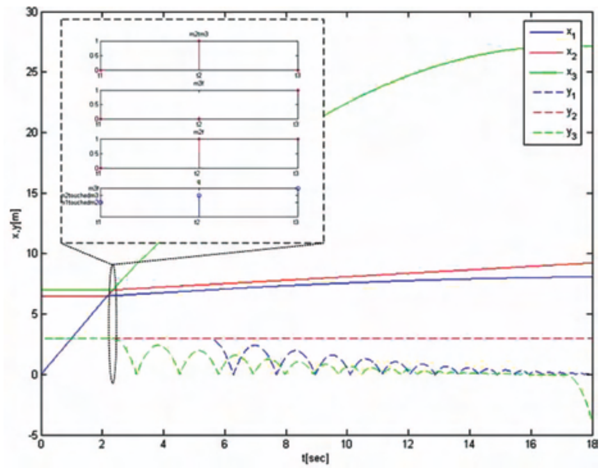

Fig. 4.5 CHECKMATE simulation results of the three-mass system.

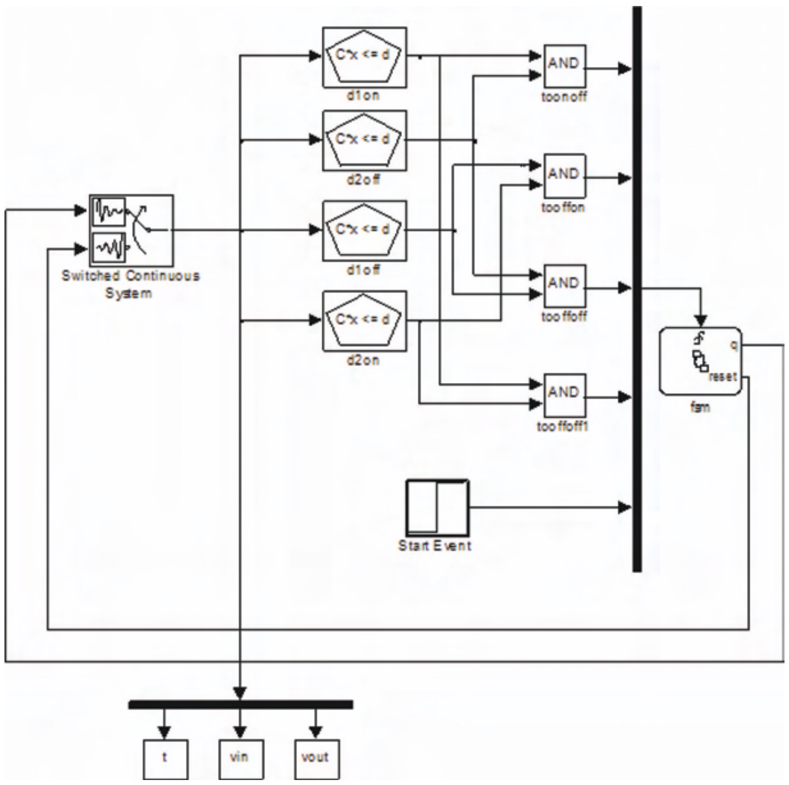

Fig. 4.6 Checkmate model of the full-wave rectifier.

the sequence of events that happen during the simulation. Event $m2f$, which indicates that m_2 starts falling, becomes enabled before event $m3f$ which indicates that m_3 starts falling. The reason is that in order for vx_3 to be greater than zero, one integration step is required which delays the enabling of $m3f$.

The Full Wave Rectifier Example. Figure 4.6 and Figure 4.7 illustrate a CheckMate model of the full-wave rectifier. The model structure is obviously identical to the three-mass system. CheckMate does not allow a dynamical system to have an external input (only the discrete input from the state machine is allowed). The sinusoidal voltage source has to be internally generated by the SCSB. If

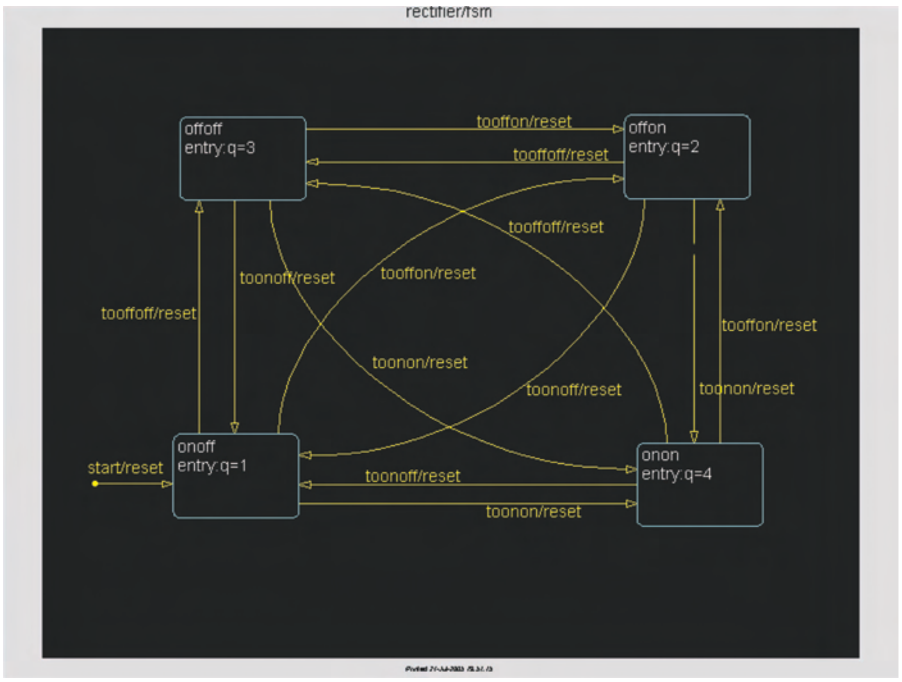

Fig. 4.7 Checkmate model of the full-wave rectifier.

we want to keep the system linear we can use a second order equation $d^2 v_{in}/dt = -\omega_0 v_{in}$ to generate the input voltage. Simulation results are shown in Figure 4.8. Algebraic loops are avoided by construction in CheckMate. In fact, only systems of the form $\dot{x} = f(x, u)$ can be described.

4.4.4 Discussion

CheckMate has several interesting aspects. First of all, it uses a very popular tool suite to capture the design specifications and to simulate the system. Second, it uses a particular restriction of the general hybrid system model presented in Section 2.1 that allows carrying out formal verification. Third, it uses conservative approximations to reduce the computation costs of formal verification for hybrid systems. From a practical viewpoint, the approximation scheme yields computational

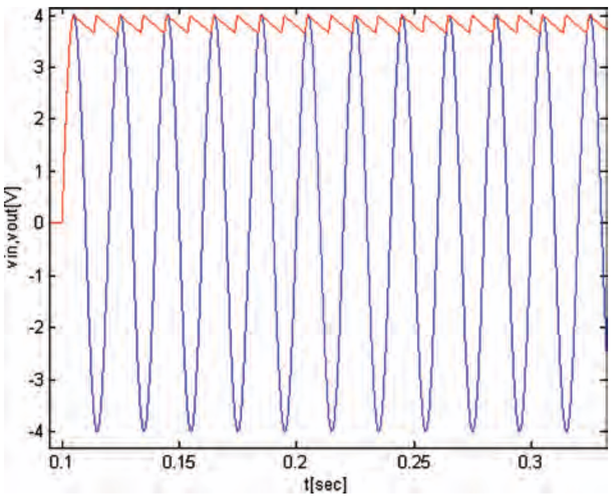

Fig. 4.8 CheckMate simulation results of the full wave rectifier.

problems that remain prohibitive when the number of variables is more than five (due to the high cost of reachability analysis). A CheckMate model implements a hybrid system as a differential equation $\dot{x} = f(x, u)$ with one discrete input u coming from a Moore-type state machine (implemented by a Stateflow chart). With CheckMate it is possible to specify dynamics that are more complex than the ones allowed by HyTech. On the other hand, HyTech provides a set of language features for the composition of hybrid automata, an operation which is not possible in CheckMate.

It was not possible to verify our models using CheckMate due to execution errors. We think that the errors arise from an incompatibility between the current version of Matlab and the version upon which CheckMate was originally developed. Unfortunately CheckMate is no longer supported and, therefore, it hasn't been possible to ask for an updated version.

4.5 PHAVer

The Polyhedral Hybrid Automaton Verifier, PHAVer [69], is a tool for the safety verification of hybrid systems with piecewise-constant bounds on the derivatives. PHAVER uses exact arithmetic whose robustness is guaranteed by the use of the Parma Polyhedral Library [25]. Safety verification reduces again to the reachability problem, which is decidable only for a subclass of hybrid systems called initialized rectangular hybrid automata. PHAVER uses an on-the-fly over-approximating algorithm to approximate piecewise affine dynamics with linear hybrid automata (LHA), which are not decidable [96]. A set of algorithms has been developed to reduce the number of bits and number of constraints that are needed to represent polyhedral regions, improving the overall efficiency of the verification algorithm. PHAVER has also the capability of computing simulation relations and of deciding equivalence and refinement between hybrid automata.

4.5.1 PHAVER Syntax

PHAVER syntax is similar to the one of HYTECH. PHAVER uses hybrid I/O automata. Given a set of variables V, a valuation is a function $v : V \to \mathbb{R}$ and $\mathcal{V}(V)$ is the set of all possible valuations of V. An activity is a function $f : \mathbb{R}_+ \to \mathcal{V}(V)$ and $act(V)$ is the set of activities on V. Also, a set of activities S is time-invariant if forall activities $f \in S$ and for all $d \in \mathbb{R}_+$, the function defined as $f_d(t) = f(t + d)$ is also in S (i.e., S is closed under time shift).

Definition 4.1. A Hybrid Input/Output Automaton is a tuple $H = (L, V_S, V_I, V_O, \mathcal{L}, \to, Act, Inv, Init)$ where:

- L is a finite set of locations;
- V_S and V_I are finite and disjoint sets of controlled and input variables, respectively. $V_O \subseteq V_S$ is the set of output variables. Let $V = V_I \cup V_S$;
- $\mathcal{L}$ is a finite set of synchronization labels;
- $\to \subseteq L \times \mathcal{L} \times 2^{\mathcal{V}(V) \times \mathcal{V}(V)} \times L$ is a finite set of discrete transitions;
- $Act : L \to 2^{act(V)}$ is a mapping that associates to each location a set of time-invariant activities;
- $Inv : L \to 2^{\mathcal{V}(V)}$ is a mapping that associates to each location a domain;

- $Init \subseteq L \times \mathcal{V}(V)$ is a set of initial states such that $(l,v) \in Init \Rightarrow v \in Inv(l)$.

The concrete syntax used to specify a hybrid automaton is also very similar to the one used by HyTech. The general structure of a hybrid automaton is specified as follows:

```
automaton aut
   state_var: var_ident, var_ident,... ;
   input_var: var_ident,var_ident,... ;
   parameter: var_ident,var_ident,... ;
   synclabs: lab_ident,lab_ident,... ;
   loc loc_ident: while invariant wait { derivative };
      when guard sync label_ident do { trans_rel } goto loc_ident;
      when ...
   loc loc_ident: while ...
end
```

The main difference is that PHAVer distinguishes between input and controlled variables, whereas in HyTech all variables are global. This distinction is important for equivalence checking. The *derivative*, *invariant*, and *guard* can be linear formulæ over the controlled and input variables, which increases the expressive power of PHAVer with respect to HyTech.

As in HyTech, PHAVer defines a set of analysis commands for the verification of a hybrid system, which are described in the next section.

4.5.2 PHAVer Semantics and Verification Strategy

The semantics of hybrid automata is described in [2]. At any time instant the state of a hybrid automaton is a pair (l,v) of a location and a valuation of the controlled variables. The state can change because of a discrete transition or because time elapses. A run of a hybrid system is then an infinite or finite sequence of states:

$$\sigma_0 \mapsto^{t_0}_{f_0} \sigma_1 \mapsto^{t_1}_{f_1} \sigma_2 \mapsto^{t_2}_{f_2} \mapsto \ldots$$

where $\sigma_i = (l_i, v_i)$, $t_i \in \mathbb{R}_+$, $f_i \in Act(l_i)$, such that:

- $f_i(0) = v_i$
- for all $0 \leq t \leq t_i$, $f_i(t) \in Inv(l_i)$
- $\sigma_i \xrightarrow{\alpha_i} \sigma_{i+1}$, i.e., $\exists \mu : (v_i, v_{i+1}) \in \mu \wedge (l_i, \alpha_i, \mu, l_{i+1}) \in\rightarrow$

Notice that a system may stay in a location only if the location invariant is true. Composition of hybrid automata is defined as in Section 4.2.2.

PHAVer implements an on-the-fly over-approximation algorithm that is based on the following principle. Consider a location l with invariant $Inv(l)$ and activity specified by the conjunction of linear expressions:

$$\bigwedge_{i=1}^{m} a_i^T \dot{x} + b_i^T x \bowtie_i c_i, \quad \bowtie_i \in \{<, \leq\}$$

Then it is possible to approximate each linear expression with the following simple rule:

$$\forall i = 1, ..., m \quad a_i^T \dot{x} \bowtie_i c_i - d_i \quad d_i = \inf_{x \in Inv(l)} b_i^T x$$

If the approximation is too coarse, a location is split in order to improve accuracy. We illustrate this algorithm with a simple example, which is graphically rendered in Figure 4.9. Consider the equation

$$\dot{v} = -\delta v$$

that can be written as $\dot{v} \leq -\delta v \wedge -\dot{v} \leq \delta v$. Let the invariant be $\beta \leq v \leq \alpha$. For the two linear equations we can compute the bounds as prescribed by the algorithm:

$$\inf_{v \in [\beta, \alpha]} \delta v = -\delta\alpha \qquad \inf_{v \in [\beta, \alpha]} -\delta v = -\delta\beta$$

We obtain the approximation $-\delta\alpha \leq \dot{v} \leq -\delta\beta$. Starting from a single point as initial condition, we can compute the reachable set, shown in Figure 4.9, as the area enclosed by the two lines $v = -\delta\alpha t$ and $v = -\delta\beta t$. If this approximation is too coarse, we can split the location into two locations along the hyperplane $v = \gamma$. The new locations have invariants $\beta \leq v \leq \gamma$ and $\gamma \leq v \leq \alpha$, respectively. The reachable set is the area enclosed by the dotted lines that refines the previous approximation. The hyperplanes along which a location is split can be

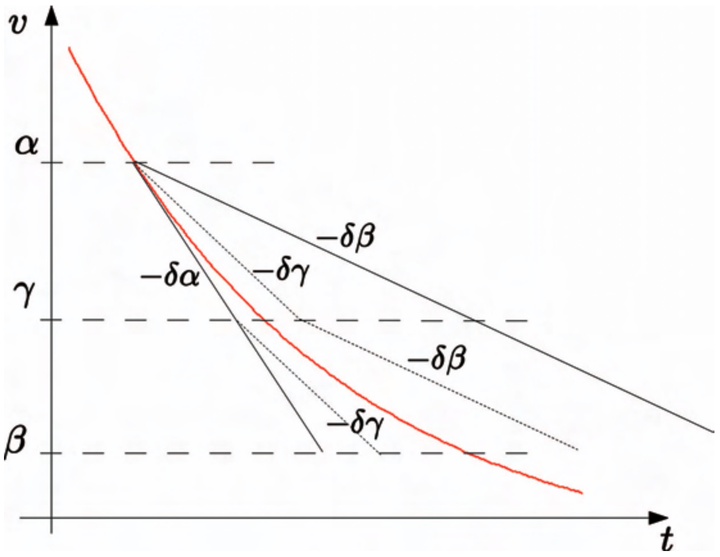

Fig. 4.9 PHAVer approximation example.

specified by the users who can guide the refinement process by relying on their knowledge of the system. For a more detailed explanation we refer the reader to [69].

4.5.3 Examples

The full-wave rectifier model consists of two hybrid automata: a voltage source and a circuit. The system of differential equations that generates the sinusoidal waveform is marginally stable, therefore any over-approximation would accumulate. In order to avoid this problem, an invariant can be added to confine the input voltage in an octagon as in Figure 4.10. The code that implements the voltage source is the following:

```
al := 0.01272; // lower bound on intersection with x0-axis
au := 0.01274; // upper bound on intersection with x0-axis
bl := 4; // lower bound on intersection with x1-axis
bu := 4; // upper bound on intersection with x1-axis
cu := 1.4143; // upper bound on sqrt(2)
al := 0.0127; // lower bound on intersection with x0-axis
au := 0.0128; // upper bound on intersection with x0-axis
bl := 4; // lower bound on intersection with x1-axis
bu := 4; // upper bound on intersection with x1-axis
```

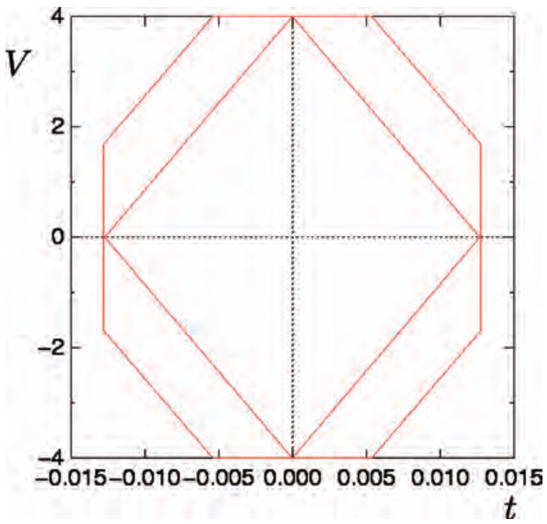

Fig. 4.10 Invariant region that confine the sinusoidal waveform.

```
cu := 1.42; // upper bound on sqrt(2)
x1min := -bu;
x1max := bu;
x0min := -au;
x0max := au;
automaton voltagesource
state_var: x0, x1;
synclabs: B;
loc l0x0: while
  x0min <= x0 & x0 <= x0max &
  x1min <= x1 & x1 <= x1max &
  x1 >= bl-bl/al*x0 &
  x1 <= cu*bu-bl/au*x0 &
  0 <= x0 & x0 <= x0max &
  0 <= x1 & x1 <= x1max
  wait { x0' == x1 & x1' == -98596*x0 };
  when true sync B goto l0x1;
  when true sync B goto l1x0;
loc l0x1: while
  x0min <= x0 & x0 <= x0max &
  x1min <= x1 & x1 <= x1max &
  x1 >= bl-bl/al*(-x0) &
  x1 <= cu*bu-bl/au*(-x0) &
  x0min <= x0 & x0 <= 0 &
  0 <= x1 & x1 <= x1max
```

```
    wait { x0' == x1 & x1' == -98596*x0 }
    when true sync B goto l0x0;
    when true sync B goto l1x1;
  loc l1x1: while
    x0min <= x0 & x0 <= x0max &
    x1min <= x1 & x1 <= x1max &
    -x1 >= bl-bl/al*(-x0) &
    -x1 <= cu*bu-bl/au*(-x0) &
    x0min <= x0 & x0 <= 0 &
    x1min <= x1 & x1 <= 0
    wait { x0' == x1 & x1' == -98596*x0 }
    when true sync B goto l0x1;
    when true sync B goto l1x0;
  loc l1x0: while
    x0min <= x0 & x0 <= x0max &
    x1min <= x1 & x1 <= x1max &
    -x1 >= bl-bl/al*x0 &
    -x1 <= cu*bu-bl/au*x0 &
    0 <= x0 & x0 <= x0max &
    x1min <= x1 & x1 <= 0
    wait { x0' == x1 & x1' == -98596*x0 }
    when true sync B goto l1x1;
    when true sync B goto l0x0;
  initially: $ & x0 == -0.01273 & x1 == 0;
  end
```

The rest of the rectifier is described by the following automaton:

```
  automaton circuit
  state_var: x2;
  input_var: x1;
  synclabs: A ;
  loc onoff: while
    x2min <= x2 & x2 <= x2max &
    x1 - x2 >= 0 & -x1 -x2 <= 0
    wait { x2' == 100000*x1- 100000*x2-10*x2 }
    when x1 -x2 <= 0 & -x1-x2 >= 0 sync A do {x2'==x2} goto offon;
    when x1 - x2 <= 0 & -x1 -x2 <= 0 sync A do {x2'==x2} goto offoff;
  loc offon: while
    x2min <= x2 & x2 <= x2max &
    -x1 - x2 >= 0 & x1 - x2 <= 0 wait { x2' == -100000*x1 - 100000*x2-10*x2 }
    when x1 - x2 >= 0 & -x1 - x2 <= 0 sync A do {x2'==x2} goto onoff;
    when x1 - x2 <= 0 & -x1 - x2 <= 0 sync A do {x2'==x2} goto offoff;
  loc offoff: while
    x2min <= x2 & x2 <= x2max &
    x1 -x2 <= 0 & -x1 - x2 <= 0 wait { x2' == -10*x2}
    when x1 - x2 >= 0 & -x1 - x2 <= 0 sync A do {x2'==x2} goto onoff;
    when x1 - x2 <= 0 & -x1 - x2 >= 0 sync A do {x2'==x2} goto offon;
  initially: offoff & x2 == 4;
  end
```

The circuit model is described as an affine hybrid system. The synchronization labels are not really needed. We included them only because they are presently required by the parser, although they have no effect on this model.

PHAVer provides a set of analysis commands to compute an over-approximation of the reachable set of states. In the full-wave rectifier case, we use the following commands:

```
sys=voltagesource&circuit;
sys.add_label(tau);
sys.set_refine_constraints((x0,au/8),(x1,bu/8),(x2,1/32),tau);
reg=sys.reachable;
reg2=reg;
reg.remove(x0); // project to x1 and x2
reg.print("out_reach",2); // save for plots
reg=reg2;
reg.remove(x2); // project to x0 and x1
reg.print("out_reach_x0x1",2); // save for plots
reg2.print("out_x0x1x2",1); // save for 3D plot
```

The first line defines a system as the composition of the voltage source and the circuit automata. The following two lines are used to guide the location splitting. The set_refine_constraints command declares a list of elements of the form (linear_expr,min). A location is split by a hyperplane of the form linear_expr $\leq c$ where c is the center of the location. The parameter min is the minimum extent of a location.

The command reg.reachable computes the set of reachable states of the system reg. It is then possible to project away some variables and generate the results. The reachable set (where x0 has been projected away) is shown in Figure 4.11.

4.5.4 Discussion

PHAVer is a very promising verification tool. It has some unique features: among others, the ability of computing simulation relations and deciding equivalence and refinement between hybrid automata. The

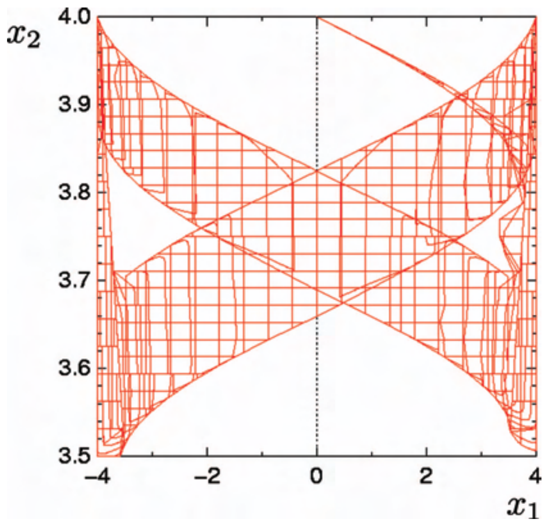

Fig. 4.11 Reachable set generated by PHAVer.

verification algorithm is very efficient: the full-wave rectifier is verified in 1748s on a Pentium 4 processor running at 2.8GHz.

PHAVer allows one to compose hybrid systems preventing the number of discrete states to grow exponentially. The class of models that can be described is the class of affine hybrid systems. Thanks to the rich command language, the user can guide PHAVer in splitting locations by defining splitting hyperplanes. This is extremely important in order to tune the verification process and reach a useful answer in a short time.

4.6 HSolver

HSolver is a tool for the safety verification of hybrid systems [147] developed at the Max-Planck-Institut für Informatik in Saarbrücken, Germany. HSolver uses the general idea of reducing the infinite state space of a hybrid system to a finite one by partitioning the continuous space into boxes. The basic reachability analysis is hence approximate. If the algorithm returns a negative answer, then the verification tool should refine the partition to make sure that indeed the set of "bad" states is not reachable.

HSolver approaches this problem by using the information of the continuous evolution inside the boxes to prune the search space from unreachable regions. Therefore, the refinement process does not always have to split boxes but can also rely on the efficient pruning algorithm.

4.6.1 HSolver **Syntax**

The syntax of the HSolver input language can be understood on the basis of the following model [147]. Let $s \in \{s_1, ..., s_n\}$ be a variable that takes values from a finite set of discrete modes, and $x_1, ..., x_k$ be variables ranging over closed intervals $I_1, ..., I_k$, respectively. Let Φ denote the resulting state space $\{s_1, ..., s_n\} \times I_1 \times ... \times I_k$. The derivative of a variable x_i is denoted by $\dot{x_i}$ and the targets of the jumps are denoted by primed variables. A constraint is an arbitrary Boolean combination of equalities and inequalities over terms that may contain function symbols like $+$, $\times$, exp, sin and cos. A *state space constraint* is a constraint on the variables $x_1, ..., x_k$. A *flow constraint* is a constraint on the variables $s, x_1, ..., x_k, \dot{x_1}, ..., \dot{x_k}$. A *jump constraint* is a constraint over the variables $s, x_1, ..., x_k, s', x'_1, ..., x'_k$. The description of a hybrid system consists of a flow constraint, a jump constraint, a state space constraint describing the set of initial states, and a state space constraint describing the set of unsafe states. For instance, consider the full wave rectifier. The set of discrete states is $\{onoff, offon, offoff\}$ and there are two continuous variables v_{in} and v_{out}. A flow constraints for the entire circuit can be described as follows:

$$\Big(s = \mathit{onoff} \rightarrow \dot{v}_{out} = \frac{v_{in} - v_{out}}{R_f C} - \frac{v_{out}}{RC} \wedge v_{in} \geq v_{out} \wedge -v_{in} \leq v_{out}\Big) \bigwedge$$
$$\Big(s = \mathit{offon} \rightarrow \dot{v}_{out} = \frac{-v_{in} - v_{out}}{R_f C} - \frac{v_{out}}{RC} \wedge v_{in} \leq v_{out} \wedge -v_{in} \geq v_{out}\Big) \bigwedge$$
$$\Big(s = \mathit{offoff} \rightarrow \dot{v}_{out} = -\frac{v_{out}}{RC} \wedge v_{in} \leq v_{out} \wedge -v_{in} \leq v_{out}\Big)$$

Notice that in this formulation jumps are forced by not allowing a flow in a certain region. In our example, we have included the invariants in the definition of the flow constraint.

Each constraint (flow, jump, initial states and unsafe states) is satisfied by a set of values, drawn from the corresponding domain of the constraint. Thus, the hybrid system can be equivalently described by the tuple $(Flow, Jump, Init, Unsafe)$ where

$$\begin{aligned} Flow &\subseteq \Phi \times \mathbb{R}^k \\ Jump &\subseteq \Phi \times \Phi \\ Init &\subseteq \Phi \\ Unsafe &\subseteq \Phi \end{aligned}$$

The input language of HSolver allows the description of hybrid systems by specifying modes, continuous variables, and the tuple $(Flow, Jump, Init, Unsafe)$. A hybrid system description has the following declarations:

- a list of the names of the variables spanning the continuous state space: VARIABLES
  [x1,...,xn] ,
- a list of the names of the discrete modes:
  MODES
  [m1,...,ms] .
- For each mode, the hyper-rectangle spanning the corresponding continuous state space:
  STATESPACE
  m1[[l1,u1],..,[ln,un]]
  ...
  ms[[l1,u1],..,[ln,un]]
  where [li,ui] denotes the lower and upper bound for variable xi.
- For each mode, a constraint describing the set of initial states in this mode:
  INITIAL
  m1 { *constraint* }
  ...
  ms { *constraint* } .
- For each mode, a constraint describing the continuous evolution in this mode:
  FLOW
  m1 { *constraint* }
  ...
  ms { *constraint* }

The constraint may contain the variables as specified using the keyword VARIABLES, and these variables followed by _d to represent the corresponding derivatives.

- For each pair of modes, a constraint describing discontinuous jumps of trajectories:

  ```
  JUMP
  mi -> mj { jump constraint }
  ...
  mk -> ml { jump constraint }
  ```

  The constraint may contain the variables as specified using the keyword VARIABLES and their primed versions. The unprimed versions describe the jump source and the primed versions the jump target.
- For each mode, a constraint describing the set of unsafe states in this mode:

  ```
  UNSAFE
  m1 { constraint }
  ...
  ms { constraint } .
  ```

4.6.2 HSolver Semantics

The semantics of a hybrid system in HSolver is defined by the set of admissible trajectories. For a function $r : \mathbb{R}_{\geq 0} \to \Phi$, let $\lim_{t' \to t_-} r(t') = (\phi, f)$ denote the left limit of r at t, where f is the left limit of the real-valued component of r and ϕ is the discrete state when, for taking the limit, the state variable is considered as a piecewise constant and left-continuous function.

Definition 4.2. A continuous time trajectory is a function in $\mathbb{R}_{\geq 0} \to \Phi$. A trajectory of a hybrid system $(Flow, Jump, Init, Unsafe)$ is a continuous time trajectory r such that:

- if the real-valued component f of r is differentiable at t, and $\lim_{t' \to t_-} r(t')$ and $r(t)$ have an equal mode s, then $((s, f(t)), \dot{f}(t)) \in Flow$,
- otherwise $(\lim_{t' \to t_-} r(t'), r(t)) \in Jump$.

A trajectory from a state x to a state y is a trajectory r such that $r(0) = x$ and $\exists t \in \mathbb{R}_{\geq 0}$ such that $r(t) = y$.

The semantics of a hybrid system is the collection of its trajectories. Note how these definitions represent essentially a simplification of the general definition that was given in Section 2.1, Definition 2.3. In particular, one could derive an equivalent hybrid time basis (Definition 2.2) by partitioning the real line into intervals over which the mode remains constant.

4.6.3 HSolver **Safety Verification**

HSolver builds an abstraction of a given hybrid system by decomposing the state space into boxes. Then it uses the observation that a point in a box B can be reachable only if it fulfills one of the three following conditions:

- it is reachable from an initial point in B via a continuous flow;
- it is reachable from a jump to B via a continuous flow;
- it is reachable from the boundary of B via a continuous flow;

The approach formalizes these three conditions in the first-order predicate language (i.e., as constraints that do not contain any differentiation symbols), and then uses the interval constraint propagation-based solver RSolver to remove points from boxes that do not fulfill any of these conditions.[2] If this is not sufficient to verify the safety of the input system, then the abstraction is refined by splitting boxes into pieces, and further pruning the state space using the approach above. Due to the fact that the constraint solver allows non-linear constraints as input, and that it is completely rigorous (i.e., the correctness of the results is not affected by rounding errors), HSolver inherits these properties.

[2] RSolver is available at `http://rsolver.sourceforge.net`

4.6.4 Examples

In this section we show how the full-wave rectifier can be described and verified. The set of modes of the rectifier is $\{m1, m2, m3, m4\}$ denoting the states *onoff*, *offon*, *offoff* and *onon*, respectively. The HSolver hybrid system description of the full-wave rectifier is as follows:

```
VARIABLES [ x0,x1,x2]
MODES [ m1,m2,m3 ]
STATESPACE
  m1[[-5,5],[-5,5],[0,4]]
  m2[[-5,5],[-5,5],[0,4]]
  m3[[-5,5],[-5,5],[0,4]]
INITIAL
  m1{x0<=-0.0121/\x0>=-0.0134/\x1=0/\x2=4}
FLOW
  m1{x0_d=x1}{x1_d=-98596*x0}{ x2_d=100000*(x1-x2) - 10*x2}
  m2{x0_d=x1}{x1_d=-98596*x0}{x2_d=-100000*(x1-x2) - 10*x2}
  m3{x0_d=x1}{x1_d=-98596*x0}{x2_d=-10*x2}
JUMP
  m1- >m2{x1<x2/\-x1>=x2}
  m2- >m1{x1>=x2/\-x1<x2}
  m1- >m3{x1<x2/\-x1<x2}
  m3- >m1{x1>=x2/\-x1<x2}
  m2- >m3{x1<x2/\-x1<x2}
  m3- >m2{x1<x2/\-x1>=x2}
UNSAFE
  m3{x2<3.5}
```

The input voltage peak amplitude A ranges between $3.8V$ and $4.2V$, $R_f = 0.1\Omega$, $R = 1K\Omega$, $C = 100\mu F$ and $f_0 = 50Hz$. The threshold voltage is set to $v_{min} = 3.5$.

The verification could not terminate. In this example, extremely fast movement happens near the mode switches in a very small area. HSolver needs a huge number of abstract states to achieve the necessary separation in this small area. Moreover, HSolver cannot exploit the fact that the problem is linear and deterministic.

4.6.5 Discussion

HSolver uses a traditional interval method for the verification of hybrid systems in an abstraction-refinement framework. When the verification algorithm finds that a system is unsafe, the over-approximation is refined by reducing the grid size. HSolver improves this traditional method by implementing a pruning algorithm that removes uninteresting parts of the state space before reducing the grid size. Consequently, the refinement of the over-approximation can be obtained even without increasing the number of grid locations, one of the causes of exponential blowout in the verification algorithms for hybrid systems.

The language for describing hybrid systems is very easy to understand. There are no limitations in describing a single automaton and the limited number of statements in the language makes it simple to use. HSolver does not support hierarchy and composition of hybrid automata.

4.7 Ellipsoidal calculus for reachability

In recent years various researchers in the control community have investigated ellipsoids as a tool to compute approximations of continuous sets. S. Veres has developed the Geometric Bounding Toolbox - currently available in the release GBT 7.3 [171] - as a Matlab toolbox that supports numerical computations with polytopes and ellipsoids in the n-dimensional Euclidean space for $n \geq 1$. The toolbox includes procedures for convex hull determination (both vertex enumeration and facet enumeration), polytope addition and difference in the Minkowski sense[3], intersections, hyper-volumes, surface-areas, orthogonal projections. affine transformations. The operations available for ellipsoids include: smallest volume ellipsoid covering a polytope, interior and exterior approximations, difference and intersection of ellipsoids.

The most systematic contributions to ellipsoidal calculus for representing reached sets are due to the research group of A.B. Kurzhanskiy, active both in Moscow and at UC Berkeley (with P. Varaiya). In a long sequence of papers [110, 113, 112, 116, 114, 115, 117], A.B. Kurzhanskiy and P. Varaiya developed techniques for approximating the reached sets of dynamical systems. They

[3] Given the convex and compact sets X and Y in R^n, the Minkowski sum is the set $X + Y = \cup_{x \in X} \cup_{y \in Y} \{x + y\}$, where $x + y$ is the vector sum of points x and y; similarly, the Minkowski difference is the set $X - Y = \cap_{y \in Y} \cup_{x \in X} \{x - y\}$, where $x - y$ is the vector sum of points x and $-y$.

addressed the general problem: Given the differential equation $\dot{x}(t) = f(x(t), u(t), v(t))$, $x(0) \in X_0$, where $x(t) \in R^n$ is the state, $u(t) \in U$ is the control, $v(t) \in V$ is the disturbance, and X_0 is the set of initial states, calculate (an approximation of) the set of states $X(t, X_0)$ that can be reached at time t, by choosing an appropriate control, whatever is the disturbance. In particular they studied how to approximate the reached sets externally and internally by ellipsoids and developed an ellipsoidal calculus.

A collection of MATLAB procedures to support the ellipsoidal calculus has been made available recently by A.A. Kurzhanski as the ELLIPSOIDAL TOOLBOX [120]. It implements the core procedures of ellipsoidal calculus and its application to the reachability analysis of continuous-time and discrete-time linear systems, and linear systems with disturbances. The main advantages of ellipsoidal representations are:

- their complexity grows quadratically with the dimension of the state space and remains constant with the number of time steps;
- it is possible to converge exactly to the reached set of a linear system through external and internal ellipsoids.

A couple of recent papers [119, 118] extended the analysis to hybrid systems under piecewise open-loop controls restricted by hard bounds, where the system equations may be reset when crossing some guards in the state space, and so there is an interplay between continuous dynamics governing the motion between the guards and discrete transitions determining the resets. They address the verification problem of intersecting or avoiding a target set at a given time or at some time within a given time interval, and propose computational strategies based on the ellipsoidal calculus.

Ellipsoidal calculus was applied in VERISHIFT, a package for safety verification of systems modeled by hybrid automata, developed by O. Botchkarev and S. Tripakis [39]. The authors worked out a reachability procedure for systems of hybrid automata with linear dynamics, expressed as differential inclusions of the form $\dot{x} \in Ax + U$; reachability analysis is performed for a bounded time Δ supplied as a parameter by the user. The algorithm over-approximates: (1) intersections, unions, linear transformations and geometric sums of convex sets; (2) the reachable set of a linear differential inclusion over time. It deploys new methods for over-approximating the unions of ellipsoids and intersections of ellipsoids and polyhedra. VeriSHIFT accepts systems of hybrid automata communicating by input/output variables and synchronous message passing and supports dynamic creation and reconfiguration of automata.

An improved version of Botchkarev's algorithm has been presented in [44], by avoiding in the reachability computation the approximations caused by the union operation in the discretized flow tube estimation. Therefore, the new algorithm may classify correctly as unreachable states that are reachable according to the original version of Botchkarev's algorithm, due to the loose over-approximations introduced by the union operation. The revised reachability algorithm was implemented inside VeriSHIFT and tested successfully on a real-life case study modeling a hybrid model of a controlled car engine. Some new theoretical results on termination of restricted classes of automata were also provided.[4]

An open research problem is how to integrate representations based on ellipsoids with those based on polyhedra to achieve the tighter approximation of a given set. A step in this direction has been recently taken with the Ariadne project [28]. Ariadne provides an environment in which algorithms for computing with hybrid automata can be developed based on representations of sets as unions of ellipsoids as well as unions of cuboids, zonotopes, simplices and polyhedra. Ariadne differs from other tools in that it uses a rigorous theory of computable analysis [172, 51] to specify a sound semantics for representations and computations involving points, sets, maps and vector fields. Using this semantics, optimal provably correct error bounds can be obtained. Currently, the geometry module, providing various representations of sets, has been completed, and work is in progress on the evaluation module, providing algorithms for evaluating functions

[4] The modified version of VeriSHIFT and the used test cases are available at `http://fsv.dimi.uniud.it/papers/improving_EC2004`.

on sets and integrating vector fields. Interfaces to these kernel modules are available through PYTHON and MATLAB, allowing scripts for safety verification by reachability analysis to be written. The ARIADNE package will soon be released as an open source distribution, so that different research groups may contribute new data structures, algorithms and heuristics.

4.8 d/dt

d/dt is a tool for the reachability analysis of continuous and hybrid systems with linear differential inclusions developed at Verimag [23, 24, 53]. Designers can use d/dt to solve the following problems:

- **reachability**: given an initial set F of states, compute an over approximation of the set of all the states reachable by the system from F.
- **safety verification**: given a set Q of bad states, check whether the system can reach Q.
- **safety switching controller synthesis**: given a safety property specified as a set S of safe states, synthesize a switching controller so that the controlled system always remains inside the safe set S by computing an under approximation of the *maximal invariant set.*

The algorithms implemented in d/dt are discussed in detail in [21, 22, 53].

4.8.1 d/dt Syntax

The input to d/dt is a hybrid automaton where:

- continuous dynamics are *linear with uncertain, bounded input* defined by a differential equation of the form $f(\mathbf{x}) = A\mathbf{x} + B\mathbf{u}$, where $\mathbf{u}$ is an input taking values in a bounded convex polyhedron U.
- all the invariants and transition guards are defined by convex polyhedra which are specified as conjunctions of linear inequalities.

- the *resets* associated with discrete transitions are *affine, set-valued maps* of the form $R(\mathbf{x}) = D\mathbf{x} + J$ where D is a matrix and J is a convex polyhedron.

Besides the hybrid automaton, the users of `d/dt` provide as input a safety specification and, optionally, some approximation parameters such as the time step or the granularity of the orthogonal approximations. Then, `d/dt` can process the input data in one of the three different modes mentioned above: reachability, safety verification, and controller synthesis. The safety specification is typically expressed as the set Q of bad states that should not be reached by the system under any possible evolution. The safety verification algorithm relies on forward reachability analysis to compute the over-approximation C of the reachable set. After checking whether C intersects with Q, `d/dt` outputs either the confirmation that the system is safe or a set of bad states that the system has reached.

4.8.2 `d/dt` Semantics

Under the continuous dynamics of the form $f(\mathbf{x}) = A\mathbf{x} + B\mathbf{u}$, the time successors of a reachable set usually form *curved objects* that in general cannot be computed exactly [24, 122]. `d/dt` relies on a conservative approximation based on polyhedral approximation and an extension of numerical integration from point-to-polyhedral sets:

(1) given a time step r and an initial polyhedron F, the tool computes another polyhedron C that approximates the set F_r of states reachable from F during the time interval $[kr, (k+1)r]$;

(2) reachable sets are represented by non-convex orthogonal polyhedra [40] because the accumulation of reachable states typically forms a highly non-convex set.

Although the same research group has presented a method for computing these approximations for an arbitrary differential function $f(\mathbf{x})$ in [54], `d/dt` only handles linear continuous dynamics. For systems with continuous dynamics of the form $f(\mathbf{x}) = A\mathbf{x}$, i.e., without input disturbances, the set of reachable states F_r is approximated by the convex hull $C = conv(F \cup F_r)$, which is first enlarged by an appropriate amount to ensure over-approximation and then approximated by a non-convex orthogonal polyhedron [24]. For systems with continuous dynamics of the form $f(\mathbf{x}) = A\mathbf{x} + B\mathbf{u}$, i.e., with uncertain bounded input disturbances, F_r is computed by simulating the evolution of the faces of F. This is done by relying on the *maximum principle* from optimal control to find the inputs that cover all possible reachable states [22, 170].

Switching Controller Synthesis Algorithm. d/dt can also be used to synthesize a controller that switches the system between continuous modes to avoid some states that belong to a set Q of bad states specified as an input by the users. The synthesis process is based on the derivation of the *maximal invariant set*, i.e., the set of states from which the controller by switching properly can avoid to enter into any element of Q. In fact, d/dt relies on the computation of an under-approximation of the maximal invariant set which is obtained through the application of the reachability techniques for hybrid automata and the use of the *one-step predecessor* operator π: given a set $\mathcal{F}$ of safe states, the set of states $\pi\mathcal{F}$ is derived by iteratively removing from $\mathcal{F}$ all those states that will leave $\mathcal{F}$ after no more than one switching, until convergence [21, 22, 53]. Then, from the maximal invariant set, d/dt derives the switching control laws that restrict the invariants and transition guards of the original hybrid automaton so that the resulting automaton meets the desired safety specification.

4.8.3 Examples

The full-wave rectifier example is modeled as a dynamical system with three states:

```
dimension: 3;
constants:
R = 1000,
C = 0.0001,
Rf = 0.1,
w0 = 314.16,
epsilon = 0.01;
```

```
initloc: 2;
initset:
type rectangle
   -0.01272 -0.01274,
   -0.00001 0.00001,
   3.99999 4.00001;
badset:
   loc_id: 2 /* offoff */
   type convex_constr
      0.0 0.0 1.0 3.5;
location: 0; /*onoff*/
   matrixA:
      0.0 1.0 0.0,
      [-w0*w0] 0.0 0.0,
      0.0 [ 1.0 / ( Rf * C ) ] [ - ( 1.0 / ( R * C ) + 1.0 / ( Rf * C ) ) ];
   scalB: 0.0;
   inputset:;
   stayset:
      type convex_constr
         0.0 -1.0 1.0 0.0, /* vin - vl >= 0*/
         0.0 -1.0 -1.0 0.0; /* -vin - vl <= 0*/
   transition:
   label nfff: /* onoff − > offoff */
      if in guard:
         type convex_constr
            0.0 1.0 -1.0 [ epsilon ]; /* vin - vl <= 0*/
         goto 2;
   label nffn: /* onoff − > offon*/
      if in guard :
         type convex_constr
            0.0 1.0 -1.0 [ epsilon ], /* vin - vl <= 0*/
            0.0 1.0 1.0 [ epsilon ]; /*-vin - vl >= 0*/
         goto 1;
   location: 1; /*offon*/
      matrixA:
         0.0 1.0 0,
         [-w0*w0] 0.0 0.0,
         0.0 [ - 1.0 / ( Rf * C ) ] [ - ( 1.0 / ( R * C ) + 1.0 / ( Rf * C ) ) ];
      scalB: 0.0;
      inputset:;
      stayset:
         type convex_constr
            0.0 1.0 -1.0 0.0, /*vin - vl <= 0*/
            0.0 1.0 1.0 0.0; /*-vin - vl >= 0 */
      transition :
      label fnff: /*offon − > offoff*/
      if in guard:
        type convex_constr
          0.0 1.0 -1.0 0.0, /*vin - vl <= 0*/
          0.0 -1.0 -1.0 0.0;/*-vin - vl <= 0*/
      goto 2;
    label fnnf: /*offon − > onoff*/
      if in guard:
        type convex_constr
          0.0 -1.0 1.0 0.0, /*vin -vl >= 0*/
          0.0 -1.0 -1.0 0.0; /*-vin -vl <= 0*/
      goto 0;
  location: 2; /*offoff*/
    matrixA:
      0.0 1.0 0,
      [-w0*w0] 0.0 0.0,
      0.0 0.0 [-1.0/(R*C)];
    scalB: 0.0;
    inputset:;
    stayset:
      type convex_constr
        0.0 1.0 -1.0 0.0, /*vin - vl <= 0*/
        0.0 -1.0 -1.0 0.0; /*-vin - vl <= 0*/
    transition :
    label fffn: /*offoff − > offon*/
      if in guard:
        type convex_constr
          0.0 1.0 1.0 0.0; /*-vin -vl >= 0*/
      goto 1;
    label ffnf: /*offoff to onoff*/
      if in guard:
        type convex_constr
          0.0 -1.0 1.0 0.0; /*vin - vl >= 0*/
      goto 0;
  ;
  limits:
    x[0] <= 10.0 and
    x[0] >= -10.0 and
    x[1] <= 10.0 and
    x[1] >= -10.0 and
    x[2] <= 10.0 and
    x[2] >= -10.0
```

We wish to verify that the output voltage does not drop below $3.5V$. This condition is described by the badset, which lists a set of regions that are considered unsafe. In our case, the region is characterized by location 2 where both diodes are off and the output voltage is less than $3.5V$. The rest of the code describes the hybrid automaton with three states. In each location the invariant is declared as a set of convex constraints on the state variables while the dynamics is specified as $\dot{x} = Ax + Bu$ where u is an external disturbance. Each location includes a list of its output transitions whose guards conditions are specified as convex regions. A parameter file is associated to the hybrid system model in order to tune the verification algorithm to the specific model and improve the verification efficiency.

Unfortunately, `d/dt` has problems in computing the over-approximation because of the marginally stable set of equations. As in the case of PHAVer, using octagonal restrictions may in principle solve the problem. However, since `d/dt` does not handle composition, the size of the resulting automaton would make this approach impractical.

4.8.4 Discussion

The features of `d/dt` are certainly very interesting. In particular, the capability of using the results of formal verification to synthesize a controller is quite appealing in embedded system design. Its limitations are similar to those of other formal verification tools: limited expressiveness, complex ways of specifying dynamics and properties, and high computational costs.

4.9 Hysdel

Hysdel is a hybrid systems description language publicly distributed by the Automatic Control Laboratory of the Swiss Federal Institute of Technology Zurich [164, 165]. Hysdel can be used to describe *discrete hybrid automata* (DHA). DHA result from the connection of a finite state machine, which provides the discrete part of the hybrid system, with a switched affine system (SAS), which provides the continuous part of the hybrid dynamics. DHAs are formulated in discrete-time and, therefore, Zeno behaviors cannot appear. The Multi-Parametric Toolbox that is based on Hysdel allows users to describe the hybrid dynamics in a textual form, perform reachability analysis and, ultimately, synthesize an optimal *piecewise affine (PWA) controller* [155].

The HYSDEL compiler is available at http://control.ee.ethz.ch/hybrid/hysdel. Additional related software in MATLAB is available at http://www.dii.unisi.it/∼hybrid/tools.html

4.9.1 HYSDEL **Syntax**

A HYSDEL netlist has the following structure:

```
SYSTEM <name> {
  /* C-style comments */
  INTERFACE {
  }
  IMPLEMENTATION {
  }
}
```

The interface section describes the following properties of a system:

- STATE, INPUT, OUTPUT: these denote the state variables, inputs and outputs subsections, respectively. State, input and output variables are declared by the type specifier (REAL for real-valued variables, or BOOL for Boolean-valued variables) that is followed by the variable name.[5] For real variables an optional interval can be specified by using the suffix [min,max] to denote the minimum and maximum value that the variable can assume, respectively.
- PARAMETER: In the parameter subsection, a parameter can be specified in one of the following ways:
  - BOOL name=value; where value is either TRUE or FALSE.
  - REAL name=value; where value is a real number.
  - REAL name; where the parameter is treated symbolically.

In the IMPLEMENTATION section, the user describes the behavior of the hybrid system using mainly the following subsections:

[5] In the sequel we shall indicate Boolean signals with a b subscript and real signals with an r subscript.

- CONTINUOUS : it contains the description of the dynamics of an affine discrete time dynamical system through equations of type var = affine-expression where var is a discrete time variable.
- AD : it is used to define Boolean variables from continuous ones using statements of type var = affine-expression <= real-number or var = affine-expression >= real-number. This section can be seen as an analog to a digital converter.
- DA : it is used to generate continuous variables from Boolean ones using the following statements: var = IF boolean-expr THEN affine-expr ; or var = IF boolean-expr THEN affine-expr ELSE affine-expr. A variable is assigned to an affine expression depending on the value of the Boolean expression. Sampling could be one example of DA section where the Boolean expression is a clock signal.
- AUTOMATA : it specifies the state transition equations of the discrete automata of the hybrid system through Boolean expressions of the form var = boolean-expression. A Boolean expression can use logic operators like & (AND), | (OR) and $\sim$ (NOT).
- OUTPUT : it defines the output functions of the hybrid system through static linear and logic relations.
- LOGIC : it is used to define internal Boolean variables.
- LINEAR : it is used to define real valued variables and algebraic expressions over them.
- MUST: it describes constraints on continuous and Boolean variables through expressions of the form: boolean-expression, affine-expression >= affine-expression, or
  affine-expression <= affine-expression

4.9.2 HYSDEL **Semantics**

HYSDEL systems semantics is defined in terms of discrete hybrid automata (DHA) (see Figure 4.12). The SAS block contains a set of discrete affine systems characterized by the following set of equations:

$$x_r'(k) = A_{i(k)}x_r(k) + B_{i(k)}u_r(k) + f_{i(k)} \quad (4.1)$$

$$y_r(k) = C_{i(k)}x_r(k) + D_{i(k)}u_r(k) + g_{i(k)} \quad (4.2)$$

where $x_r'(k) = x_r(k+1)$, $x_r \subseteq \mathbb{R}^{n_r}$ is the continuous state vector, $u_r \subseteq \mathbb{R}^{m_r}$ is the external input, $y_r \subseteq \mathbb{R}^{p_r}$ is the continuous output vector and, finally, $\{A_{i(k)}, B_{i(k)}, C_{i(k)}, D_{i(k)}\}_{i\in\mathcal{I}}$ are matrices of appropriate dimension. Depending on the value of k, index $i(k)$ selects a different set of matrices, and hence a different affine system. This means that $i(k)$ represents a mode of operation characterized by different discrete dynamics. The mode is computed by a logic function of the Boolean state and input variables, as described below. The finite state machine, in turn, represents the hybrid automata whose state transitions depend on the external Boolean input u_b, the previous state and the Boolean

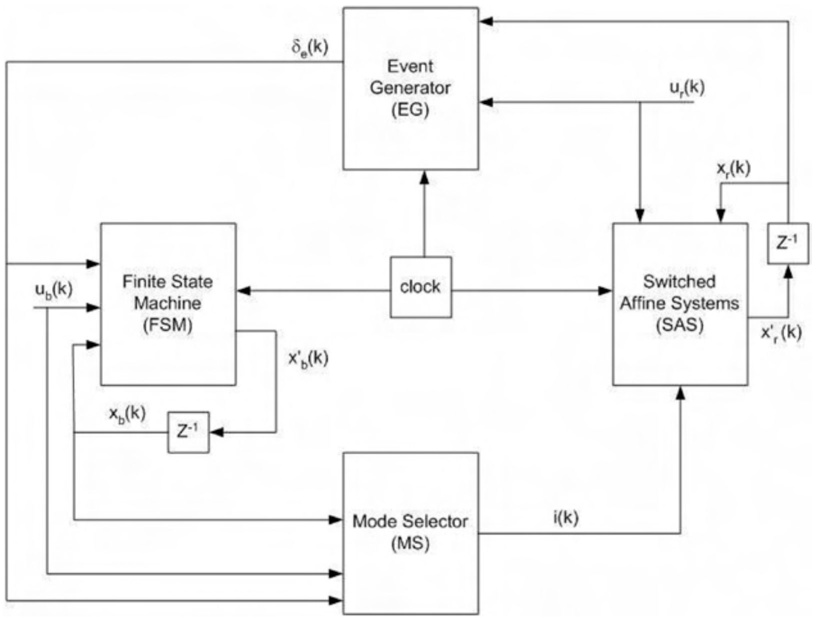

Fig. 4.12 Block diagram of a discrete hybrid automata.

variable $\delta_e(k)$,

$$x'_b(k) = f_B(x_b(k), u_b(k), \delta_e(k)) \tag{4.3}$$

$$y'_b(k) = g_B(x_b(k), u_b(k), \delta_e(k)) \tag{4.4}$$

$\delta_e(k)$ is true when some particular conditions on the continuous variables are satisfied. In particular:

$$\delta_e(k) = f_H(x_r(k), u_r(k), k)$$

$\delta_e(k)$ is a vector of boolean variables and we use the superscript i to denote the i-th component of the vector. In particular, time events are modeled as $\delta_e^i(k) = 1 \iff kT_s \geq t_0$ (where T_s is the sampling time), and threshold events are modeled as $\delta_e^i(k) = 1 \iff a^T x_r(k) + b^T u_r(k) \leq c$.

The mode selector is a logic function $i(k) = f_M(x_b(k), u_b(k), \delta_e(k))$. In this setting, reset maps can be considered as special dynamics acting for one sampling step. During this step, variables are set to a specific value.

A HYSDEL program has a natural interpretation as a DHA. The CONTINUOUS sections are used to describe affine systems in the SAS block. The AD sections are used to generate $\delta_e(k)$ while the DA sections are used to switch among several affine systems depending on the value of some Boolean variables. Finally, the AUTOMATA section is used to describe the finite state machine. The use of LINEAR sections could lead to the presence of algebraic loops. Algebraic loops are statically detected and reported by the HYSDEL compiler. Notice that the discrete nature of a HYSDEL program makes it impossible to describe Zeno automata.

4.9.3 Examples

HYSDEL only models discrete time dynamics with fixed sampling time. Hence, testing event detection and exploiting Zeno executions is not possible, and we therefore do not present the example of the three-mass system.

We model the full-wave rectifier and synthesize a controller that selects a value of the capacitance to limit the output ripple.

```
SYSTEM RectifierRC {
  INTERFACE {
    STATE {
      REAL vc[−10.0, 10.0];
      BOOL onon, onoff, offon, offoff; }
    INPUT {
      REAL vin[−10.0, 10.0]; }
    PARAMETER {
      REAL T = 0.000001;
      REAL Rf =0.1;
      REAL R = 1000;
      REAL C = 0.0001; }
  } /* end interface */
  IMPLEMENTATION {
    AUX {
      BOOL d1on,d2on;
      REAL i1,i2; }
    CONTINUOUS {
      vc = vc − vc * T/(R * C) + (i1 + i2) * T/C;
    }
    AUTOMATA {
      onon = (onon & d1on & d2on) | (onoff & d1on & d2on) |
      (offon & d1on & d2on) | (offoff & d1on & d2on);
      onoff = (onoff & d1on & ~d2on) | (onon & d1on & ~d2on) |
      (offoff & d1on & ~d2on);
      offon = (offon & ~d1on & d2on) | (onon & ~d1on & d2on) |
      (offoff & ~d1on & d2on);
      offoff = (offoff & ~d1on & ~d2on) | (onoff & ~d1on &
      ~d2on) | (offon & ~d1on & ~d2on) | ( onon & ~d1on &
      ~d2on );
    }
    AD{
      d1on = vin − vc >= 0.0;
      d2on = −vin − vc >= 0.0;
    }
    DA{
      i1 = { IF (onon | onoff) THEN (vin-vc)/Rf ELSE 0.0 };
      i2 = { IF (onon | offon) THEN (-vin-vc)/Rf ELSE 0.0 };
    }
  } /* end implementation */
}
```

The HYSDEL model has several states: vc is a continuous state representing the output voltage while onon, onoff, offon, offoff are discrete states representing a one-hot encoding of the four states in Figure 2.4. There is one input vin that represents the external voltage source. The CONTINUOUS section implements the time discretized version of the differential equation $\dot{vc} = -vc/(RC) + (i_1 + i_2)/C$, where i_i is the current flowing through diode d_i. Such current depends on the voltage difference across the diode.

The AUTOMATA section implements the logic of the state machine in Figure 2.4 and uses two auxiliary boolean variables indicating the region of operation of each diodes. Those variables are defined in the AD section. A diode is on when the voltage across its pins is positive which translates into a linear inequality in the variables of the model. The DA section computes the two currents i_1 and i_2 depending on the current state of the automaton.

After a model is described using the HYSDEL language it can be compiled with the HYSDEL compiler in order to generate an input file for a MATLAB simulation (it is also possible to generate a mixed logical dynamical description of the same system). The MATLAB simulation file has the following interface:

```
function [xn, d, z, y] = circuit(x, u, params)
```

It simulates one step starting from the initial conditions x, with input u and parameters $params$. It returns the new state xn, the output y, and some auxiliary variables used in the internal representation of a DHA. The HYSDEL toolkit provides also a wrapper function with the following interface:

```
function [XX,DD,ZZ,YY] = hybsim(x0,UU,sys,params,Options)
```

where UU is an input vector, $x0$ is the initial condition, sys is the MATLAB simulation file. The hybsim function simulates the system sys for all samples in UU. This is the reason why even if a system has

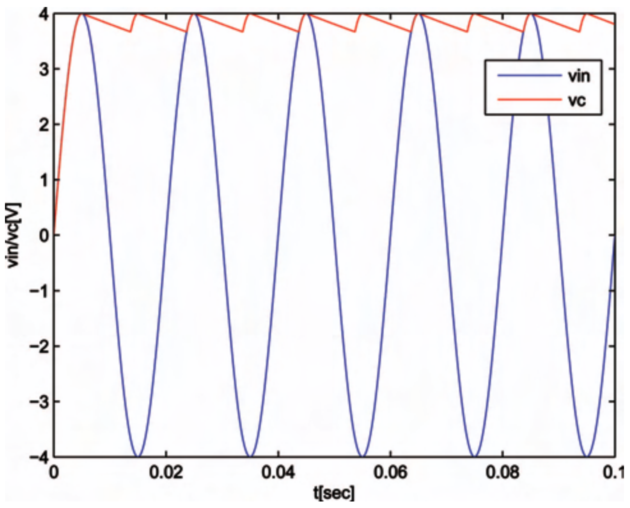

Fig. 4.13 HYSDEL simulation result of the rectifier example.

no inputs it is still necessary to have at least the time-line as input. Simulation results are shown in Figure 4.13.

4.9.4 Controller Synthesis: the Multi-Parametric Toolbox

Optimal controller synthesis. In [31] the authors propose a framework for modeling systems where physical laws, logic rules and constraints are interdependent. Models in the proposed formalism are denoted *mixed logical dynamical* (MLD) systems. An MLD description consists of a set of linear dynamic equations subject to linear inequalities involving real and integer variables. Using the MLD formulation, the authors give an algorithm for the synthesis of optimal control laws of a given discrete time hybrid system.

The proposed method to control hybrid systems is called *Model Predictive Control* and it is based on the *receding horizon* philosophy. At each given time when a measurement of the system's state is available, a sequence of input actions is computed based on the prediction of the future evolution of the system. Such a sequence is applied to the plant until a new measurement is available. At that time, a new sequence is computed. Computing the sequence of control actions is equivalent to solving a mixed-integer quadratic (MIQP) or linear (MILP) problem (depending on the norm used in the cost function). This technique requires the on-line solution of mixed-integer programs, a task that is typically very computationally intensive. In [29] the same authors propose a new method based on multi-parametric programming that moves all the computation off-line. Since the MLD representation has been proved to be equivalent to the piecewise affine (PWA) representation [85], we describe the optimal control problem formulation and the main results for PWA systems as it is done in [30, 121].

Consider a PWA system of the form:

$$x(k+1) \;=\; A_i x(k) + B_i u(k) + f_i \tag{4.5}$$

$$s.t. \; L_i x(k) + E_i u(k) \le W_i \tag{4.6}$$

$$if \;\; [x'(k)u'(k)]' \in \mathcal{D}_i, \quad i \in I \tag{4.7}$$

where $x \in \mathbb{R}^n$, $u \in \mathbb{R}^m$, $\mathcal{D}_i$ is a polyhedral set, I is an index set and the matrices are of suitable dimensions. Let's denote Equation 4.5 to Equation 4.7 with $x(k+1) = f_{PWA}(x(k), u(k))$. The constrained finite-time optimal control problem can be formulated as follows:

$$J_N^*(x(0)) = \min_{u_0,\ldots,u_{N-1}} ||Q_f x(N)||_l + \sum_{k=0}^{N-1} ||Ru(k)||_l + ||Qx(k)||_l \tag{4.8}$$

subject to

$$\begin{aligned}
&x(k+1) = f_{PWA}(x(k), u(k))\\
&x(N) \in \mathcal{X}_{set}\\
&\text{if } l = 2, \text{ then} \quad Q = Q' \succeq 0, \quad Q_f = Q_f' \succeq 0, \quad R = R' \succeq 0
\end{aligned}$$

Let $\mathcal{X}_f^N$ be the N-step feasible set, i.e., the set of initial states $x(0)$ for which the constrained finite-time optimal control problem is feasible. Then the following theorem holds:

Theorem 4.3. Consider the constrained finite-time optimal control problem. Then, the set of feasible parameters $\mathcal{X}_f^N$ is convex and the optimizer $U_N^*: \mathcal{X}_f^N \rightarrow \mathbb{R}^{Nm}$ is continuous and piecewise affine, i.e.,

$$U_N^*(x(0)) = F_r x(0) + G_r \quad \textit{if} \quad x(0) \in \mathcal{P}_r = \{x \in \mathbb{R}^n | H_r x \leq K_r\}, \\ r = 1, ..., R$$

The theorem says that the optimal controller generates a sequence of input actions as an affine function of the plant's state. The controller is indeed PWA. This problem can be solved as a multi-parametric program where the partition $\mathcal{X}_f^N = \{\mathcal{P}_r\}_{r=1}^R$ is computed and for each partition the optimal F_r and G_r is given. The algorithms are implemented in a MATLAB toolbox called Multi-Parametric toolbox [121].

The Multi-Parametric Toolbox. The multi-parametric toolbox (MPT) is available for download at http://control.ee.ethz.ch/∼mpt/downloads/. It is shipped together with a set of additional packages like *CDD* for polytope manipulation and a HYSDEL interface that reads a HYSDEL specification and generates a MATLAB structure that is used as internal representation by the MPT.

For the purpose of illustrating how the MPT works, we show how a PWA model is described directly in MATLAB. The system that we want to control (the plant) is illustrated in Figure 4.14. The voltage v_{in} is a triangular waveform. Depending on the value of the input u it is possible to decide whether the load is connected to R_1, R_2 or disconnected from the sources (we assume that R_1 and R_2 are equal, and denote their value with R_f). The system has three state variables: v_c is the voltage across the load which is the parallel connection of a resistor and a capacitor; v_{in} is the input voltage; s is a Boolean variable indicating if the input voltage has a positive or a negative slope. The plant is specified as a MATLAB structure sysStruct that lists the matrices A_i, B_i, C_i, D_i, the vectors f_i and g_i, and the bounds on the state and input variables. The polyhedral set $\mathcal{D}_i$ must be specified for each dynamic i and is described in additional fields of the same data structure by the matrices $guardX_i$, $guardU_i$ and $guardC_i$ such that $guardX_i x + guardU_i u \leq guardC_i$. For instance, if $u = 0$ then the

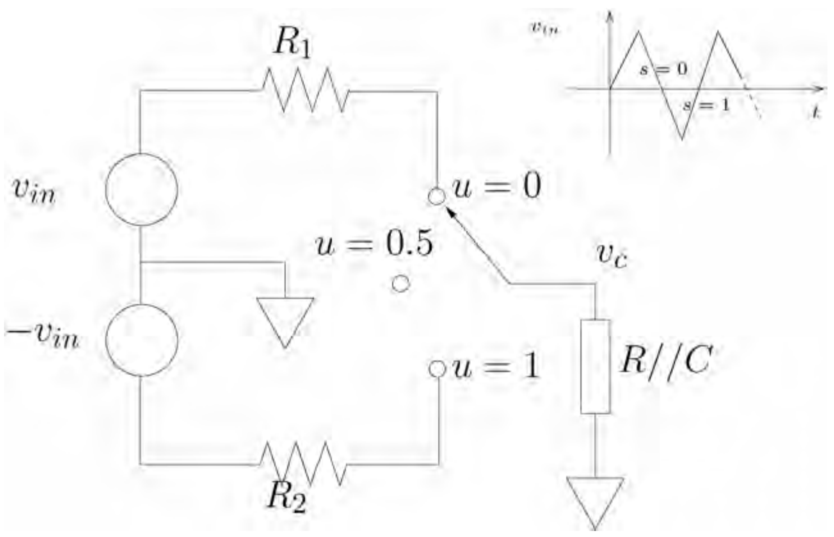

Fig. 4.14 The system under control (here $R||C$ denotes the parallel connection of the load resistor and capacitor).

state update equations are:

$$\begin{aligned} v_c(k+1) &= v_c(k)(1 - T\frac{(R_f + R)}{R_f RC}) + v_{in}(k)\frac{T}{R_f C} \\ v_{in}(k+1) &= v_{in}(k) + d * T \\ s(k+1) &= s(k) \end{aligned}$$

that are active in a polyhedral region defined respectively

$$\begin{aligned} s = 1 \wedge v_{in} \leq 1 \wedge u \leq 0 \quad &\textit{if } d > 0, \\ s = 0 \wedge -v_{in} \leq 1 \wedge u \leq 0 \quad &\textit{if } d < 0, \end{aligned}$$

where d is the derivative of the input voltage.

The MATLAB structure probStruct is used for setting up the synthesis problem aim at deriving controller automatically. Table 4.1 describes the most important parameters that are stored in probStruct. There are three possible levels of optimality that can be specified:

- 0 seeks the cost-optimal solution that minimizes the cost function in Equation 4.8;

Table 4.1 Parameters of the controller synthesis algorithm.

Parameter	Meaning
probStruct.N	Prediction horizon
probStruct.Q	Weights on the states
probStruct.R	Weights on the inputs
probStruct.norm	1 or Inf for linear problem, 2 for quadratic problem
probStruct.subopt_lev	Level of optimality, either 0, 1 or 2
probStruct.Tset	A polytope describing the terminal set $\mathcal{X}_{set}$

- 1 seeks a time-optimal solution where the controller pushes a given state to an invariant set around the origin as fast as possible;
- 2 is used for a low-complexity control scheme.

After the two MATLAB structures sysStruct and probStruct have been defined, a controller can be synthesized with the command

```
ctrl = mpt_control(sysStruct,probStruct),
```

where ctrl is a MATLAB structure representing the synthesized controller.

The MPT offers a rich set of features for debugging and optimizing the final result. It is possible to visualize the regions of the synthesized controller with the command plot(ctrl). Furthermore, a SIMULINK library is provided to instantiate and connect a plant and a controller in closed loop. The SIMULINK blocks read the plant and controller structures from the MATLAB workspace and a simulation can be run to check if the controller performances are as expected. In our case we want to synthesize a controller that selects u in such a way that the state v_c is close to the input peak voltage, which is equal to one. For this purpose, we set the plant output y equal to the state v_c and set the parameter probStruct.yref = 1 which means that the controller has to minimize the distance of the plant output from the reference output. We also choose probStruct.subopt_lev = 0, probStruct.N = 2. The resulting controller, which has 27 regions, is shown in Figure 4.15 together with the SIMULINK model. The simulation trace was obtained using the command mpt_plotTimeTrajectory(ctrl,x0,horizon,Options) that simulates the closed loop system for a number of steps specified by horizon starting from x_o. The resulting controller behaves as expected, i.e., it rectifies the input voltage in order to minimize the error with respect to the given y_{ref}.

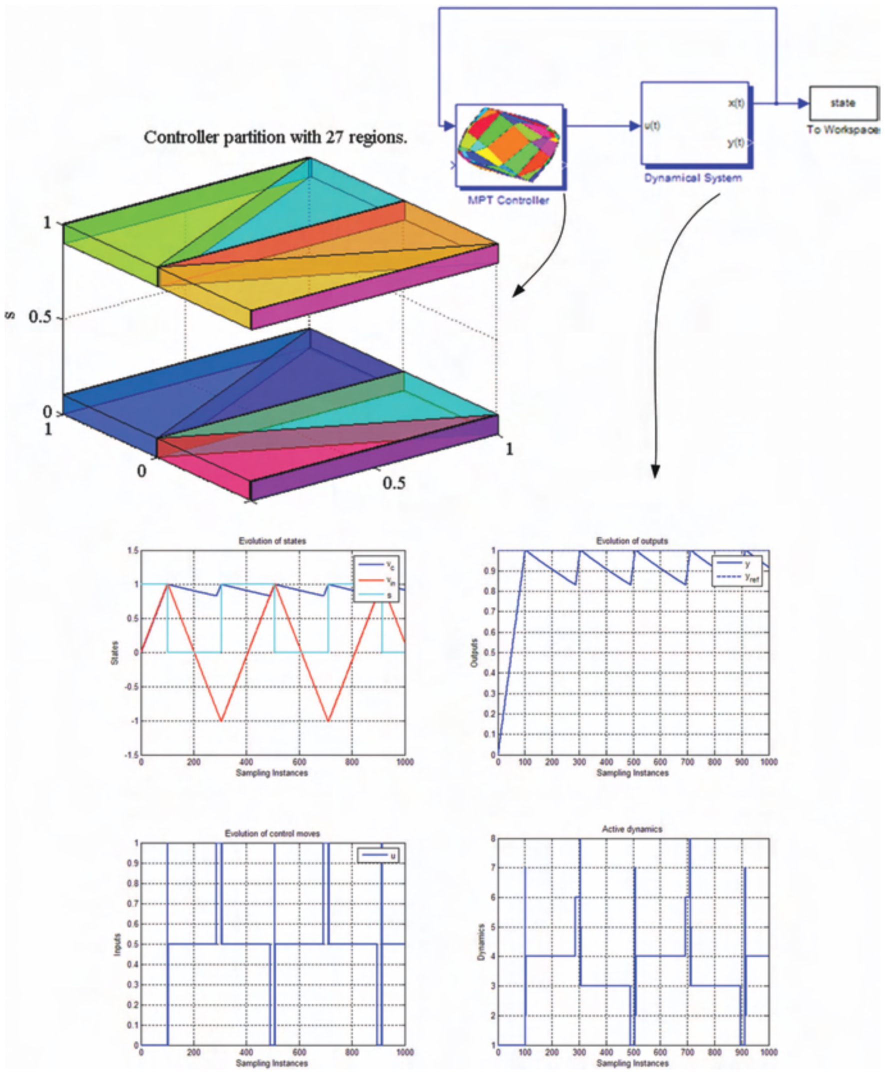

Fig. 4.15 Simulation of the closed loop system.

4.9.5 Discussion

HYSDEL is a language for the description of discrete hybrid automata. The language was developed targeting the modeling of discrete-time, affine dynamical systems. There are important features that are missing from the language. First of all hierarchy: HYSDEL programs are flat, i.e., it is not possible to instantiate subsystems and compose them (not even the syntax supports it). Features like declaration, instantiation, hiding, and object-orientation are also missing. In fact, it is not possible to declare objects of any sort and then instantiate them to compose a system of more complex objects.

The possibility of linking a HYSDEL description to a synthesis flow is a unique feature. The MPT is in a very advanced stage of development and has been used in industrial applications for synthesizing controller and generating code. It suffers from the intrinsic complexity of the synthesis algorithm but it provides a very powerful infrastructure for debugging and post-processing synthesis results. Being embedded in the MATLAB environment it provides a user friendly and familiar interface.

5 Comparative Summary

In this section we give a comparative summary of the design approaches, languages, and tools presented in this survey.

An important, and expected, conclusion of our analysis is that no single tool covers all the needs of designers that use hybrid system as models to solve their problems. While being able to capture the behavior of the system under study in an intuitive and compact way and simulating it is an important feature for any design framework, formal analysis and synthesis tools have a much higher potential in delivering a substantial productivity gain and error-free designs. These tools rely upon abstraction and hierarchy to solve industrial-strength problems. The choice of abstraction levels and of decompositions into parts is not unique and it is rare that a designer can find the right solution at the first try. Hence, interactive environments where simulation is used to guide the selection of the appropriate abstractions and decompositions are indispensable to advance the state of the art in the design and verification of hybrid systems.

To build this kind of environment, it is essential to provide a common ground for the different tools to integrate. When models are as complex as hybrid systems, defining this common ground is by no means trivial.

Table 5.1 Main purpose of the various languages, modeling approaches, and toolsets.

Name	Main Purpose
CHARON	formal semantics for hierarchy, concurrency, refinement
CHECKMATE	formal semantics (TEDHS) for simulation and verification
d/dt	safety verification of hybrid systems with linear continous dynamics
HSOLVER	safety verification of hybrid systems
HYSDEL	modeling of discrete-time affine dynamical systems
HYTECH	modeling and verification of linear hybrid automata
HYVISUAL	modeling and simulation of hybrid systems, hierarchy support
MASACCIO	support for concurrent, sequential, and timed compositionality
MODELICA	object-oriented modeling of heterogeneous physical systems
PHAVER	safety verification of affine hybrid systems
SCICOS	modeling and simulation of hybrid systems
SHIFT	modeling of *dynamic* networks of hybrid automata
SIMULINK	analysis and simulation, hierarchy support, model discretizer
STATEFLOW	FSM, statechart formalism, hierarchy support.
SYNDEX	real-time code generation, distribution and scheduling

Table 5.2 Nature and features of the various languages, modeling approaches, and toolsets.

Name	Nature	Additional Features
CHARON	modeling language	simulator, type checker, interface to JAVA
CHECKMATE	verification toolbox	integrated with MATLAB SIMULINK/STATEFLOW
d/dt	verification tool	synthesis of safe switching controllers
HSOLVER	verification tool	accepts non-linear input constraints
HYSDEL	modeling language	generation of input for MATLAB simulation
HYTECH	symbolic model checker	support for parametric analysis
HYVISUAL	visual modeler	PTOLEMY II-based block-diagram editor
MASACCIO	formal model	enables *assume-guarantee reasoning*
MODELICA	modeling language	MODELICA standard library, commercial tools
PHAVER	verification tool	support for equivalence/refinement between hybrid automata
SCICOS	hybrid system toolbox	C code generation, interface to SYNDEX
SHIFT	programming language	C code generation, λ-SHIFT for real-time control
SIMULINK	interactive tool	MATLAB-based, library of predefined blocks
STATEFLOW	interactive tool	chart animation, debugger
SYNDEX	system-level CAD	HW/SW codesign support, formal verification

Table 5.1 and Table 5.2 summarize the distinctive features of the various modeling and design environments, programming languages, simulators and tools for hybrid systems that we have discussed in the previous sections.

Table 5.3 shows the approaches adopted by each language for modeling the basic hybrid system structure. The first column shows how the discrete automata are described in the respective languages. While

Table 5.3 Comparing the modeling approaches: modeling the basic hybrid system structure.

Name	Automata Definition	State-to-Dynamics Mapping	Supported Dynamics	Guards	Invariants	Reset Maps
SIMULINK/ STATEFLOW	STATEFLOW and SIMULINK switches	STATEFLOW output selecting state evolution	No limitations	Conditions on STATEFLOW inputs and threshold crossing detector	Not supported	Integrator's reset from STATEFLOW output
MODELICA	Not explicitly defined	Events enabling equations	No limitations	Triggering relations on variables (`when` statement)	Not an explicit language feature	Through `reinit` statement
HYVISUAL	Explicit finite state machine representation	Discrete-state refinement	No restrictions	Triggering conditions on state variables	Not supported	Assignment on the FSM edges
SCICOS	Not explicitly defined	Events switching dynamics	No restrictions	Threshold detectors	Threshold detectors	Reinitialization of integrators' state
SHIFT	Textual definition of locations and transitions	Flows as locations' arguments	No restrictions	Conditions on system variables	Conditions on system variables	Assignment statements
CHARON	Mode compositions and refinement	Differential and algebraic constraints inside modes	No restrictions	Enabling conditions on system variables	Constraints on system variables	Assignment statements
HYTECH	Explicit declaration of locations and transitions	Flows defined in each location	Convex predicate over derivatives of state variables	Conjunction of linear constraints	Convex predicate over state variables	Assignment statements
PHAVER	Explicit declaration of locations and transitions	Flows defined in each location	Affine	Conjunction of linear constraints	Convex predicate over state variables	Assignment statements

(continued)

Table 5.3 *(Contd.)*

Name	Automata Definition	State-to-Dynamics Mapping	Supported Dynamics	Guards	Invariants	Reset Maps
HSolver	Declaration of modes and jump constraints	Flows defined in each mode	General linear and non-linear constraints	General constraints	General constraints	Assignment statements
CheckMate	Stateflow	Mode selector from Stateflow to a set of dynamics	Linear or non-linear (simulation only or approximation to linear dynamics)	Affine inequalities	Not supported	Affine maps
`d/dt`	Explicit declaration of locations and transitions	Flows defined in each location	Linear	Convex polyhedra	Convex polyhedra	Not supported in the version shipped to us
Hysdel	Logic formulas on Boolean variables	Mode selectors	Discrete Time and Linear	Threshold conditions on system variables	Not supported	Modeled as one step dynamics

most of them provide support to describe finite state machines, discrete states cannot be clearly distinguished in SIMULINK/STATEFLOW, MODELICA and SCICOS. In SIMULINK/STATEFLOW the discrete automata can be described using a STATEFLOW chart but it is also possible to use SIMULINK blocks to encode state (as we did in the case of the full-wave rectifier). MODELICA does not define locations and transitions. It is up to the user to define discrete states and derive a finite state machine using the statements that the language provides. SCICOS follows an approach similar to SIMULINK as it offers a library of components that can be interconnected to build a hybrid system. Further, in SCICOS it is not easy to provide guidelines for building state machines in a way that can be easily reverse engineered.

Another basic feature is the association of a dynamical system to a specific state of the hybrid automaton. HYVISUAL and CHARON have perhaps the most intuitive syntax and semantics for this purpose. In HYVISUAL a state of the hybrid automaton can be refined into a continuous time system. CHARON allows a mode to be described by a set of algebraic and differential equations. In CHECKMATE, SIMULINK, and HYSDEL a hybrid system is modeled as two main blocks: a state machine and a set of dynamical systems. The automaton is described by a finite state machine where a transition can be triggered by an event coming from a particular event-generation block that monitors the values of the variables of the dynamical system. On the other hand, the finite state machine can generate events that are sent to a mode-change block whose purpose is to select a particular dynamics depending on the events. SCICOS implements the automaton directly as an interconnection of blocks whose events can affect the continuous state of those blocks that implement the continuous dynamics. In MODELICA, the occurrence of an event can enable or disable equations that affect the continuous evolution of the system variables.

The type of dynamics supported by each language depends on the main target of the corresponding tool. For tools targeting simulation, there are very few restrictions, dynamics can be linear or non-linear. Some tools, like HYTECH and `d/dt`, only allow linear dynamics. This restriction is needed in order to limit the complexity of the verification and synthesis algorithms. The same kind of restrictions are imposed on the specification of guard conditions and invariants. Other verification tools, like CheckMate and HSolver, allow one to use more complicated dynamics and perform an approximation of the trajectories. Their application is still limited to simple examples. PHAVer allows the specification of affine dynamics and it also supports composition of hybrid automata. The verification algorithm is very efficient and can be instructed by the user. It also has the capability of checking refinements and simulation relations. Invariants are only explicitly supported by Charon, HyTech, d/dt, PHAVer and HSolver while the other tools have triggering guards semantics.

While Simulink/Stateflow does not explicitly distinguish between discrete and continuous signals, all the other languages do. Some languages like Charon and Modelica use special type modifiers to indicate whether a variable is discrete or continuous. However, the **semantics is different** in the two cases. In Charon a discrete variable is defined to be constant between two events and, therefore it has a derivative equal to zero. In Modelica, instead, the derivative of discrete variables is not defined. Graphical languages like HyVisual, Simulink, and Scicos rely on attributes associated with ports. Also, signal types can be automatically inferred during compilation through a static analysis of the system topology. Hysdel and CheckMate describe the hybrid system as a finite state machine connected to a set of dynamical systems, which makes the separation of discrete and continuous signals very sharp.

Table 5.4 shows the features provided by the different tools. Tools are ordered from the one that gives more freedom to the designer to the most restrictive one.

Two very important features for modeling complex systems are hierarchy and composition. Not all languages support the composition of hybrid systems: CheckMate, d/dt and Hysdel only allow the designer to describe a monolithic model. Not supporting composition requires the user to input a hybrid automation that is the result of the cross product (composition) of the constituent automata. This usually leads to a model with a huge number of states.

An interesting and useful feature is object orientation (OO). By object orientation we mean the possibility of defining objects and

Table 5.4 Comparing the modeling approaches: language features.

Name	Hierarchy	Composition	OO	Causality	Algebraic Loops	Continuous/Discrete Interface
SIMULINK/ STATEFLOW	Yes	Through continuous variables (SIMULINK) and discrete events (STATEFLOW)	No	Causal	Solved through explicit instantiation of algebraic loops solvers	STATEFLOW outputs acting on SIMULINK blocks
MODELICA	Yes	Through connection statements	Yes	Non-causal classes and causal functions	Simulator dependent	Events enabling equations
HYVISUAL	Yes	Through ports exposing internal variables, both continuous and discrete	Yes	Causal	Not supported	States refined into dynamical systems and special conversion blocks
SCICOS	Yes	Through continuous and discrete variables	No	Causal	Not supported	Discrete states affecting continuous states
SHIFT	Yes	Through continuous variables, automata transitions synchronization and components	Yes	Causal	Not supported	Location associated with flows and reset maps
CHARON	Yes	Through connections of agents' variables	No	Causal	Not supported	Modes defining differential and algebraic constraints and reset maps

(Continued)

Table 5.4 *(Contd.)*

Name	Hierarchy	Composition	OO	Causality	Algebraic Loops	Continuous/Discrete Interface
HyTech	No	Synchronization of automata and shared variables	No	Non-Causal	Yes	Locations associated with flows and reset maps
PHAVer	No	Synchronization of automata and connection by name	No	Non-Causal	Yes	Locations associated with flows and reset maps
HSolver	No	No	No	Non-Causal	Yes	Modes associated with flows and reset maps
CheckMate	No	No	No	Causal	Not supported	Mode selectors switching dynamics and affine reset maps
`d/dt`	No	No	No	Causal	Yes	Location associated with flows
Hysdel	No	No	No	Causal	Not supported	Mode selectors switching dynamics

extending them through inheritance and field/method extension. From this viewpoint, Simulink is not object oriented since it is not possible to define a subsystem and then inherit its properties and add other capabilities.

Another very important feature is the possibility of modeling non-causal systems. Modelica and the verification tools are the only languages that allow non-causal modeling.

None of the simulation languages considered in this survey has a clear definition of the semantics of programs that contain algebraic loops. All of them rely on the simulation engine that cannot solve algebraic loops and will stop with an error message. We believe that a language has to give a meaning to programs containing algebraic loops and the meaning should be independent from the simulator's engine. The situation is different for verification tools that either do not allow the creation of algebraic loops by construction, or they handle algebraic loops symbolically.

The last column in Table 5.4 describes how discrete and continuous signals and blocks interact with each other. CheckMate and Hysdel use an event-generator and a mode-change block. HyVisual and Simulink provide special library blocks to convert between discrete and continuous signals. In Scicos, a block can have both continuous and discrete inputs as well as continuous and discrete states. Discrete states can influence continuous states. Charon and Modelica have special modifiers to distinguish between discrete and continuous signals. As in all other languages, assignments of one to the other are not allowed and can be statically checked (by a simple type checker).

6

The Future: Towards the Development of a Standard Interchange Format

We argued that a single environment cannot offer a complete solution to the needs of designers who use hybrid models to represent the system under development. Hence, having a framework where different tools can interact and exchange information is of paramount importance to advance the state-of-the-art in the field of hybrid systems. One way to accomplish this is to adopt a standard language with its syntax and semantics being the basis for the development of a number of design tools including simulation, formal verification and synthesis. While this would be highly desirable, it would require a massive restructuring of several of the available tools and environments, an almost impossible proposition. An alternative that has been successful in Electronic Design Automation (EDA) is to develop an *interchange format* that serves as a bridge among the different tools. We believe this path is feasible and we give some insight on how to design this format.

An *interchange format* is a file, or a set of files, that contains data in a given syntax that is understood by different interacting tools. It is not a database nor a data structure, but a simpler object whose goal is to foster the exchange of data among different tools and research groups. It is important to understand the differences between modeling languages and interchange formats.

The goal of a modeling language is enabling the formal representation of selected aspects of a system. As such, a modeling language is always restrictive (only selected aspects are modeled), formal (has well defined concrete syntax, abstract syntax, and semantics) and unambiguous. The goal of an interchange format is to communicate models among tools using different modeling languages. Accordingly, interchange formats are not restrictive (all syntactically and semantically sound models can be interchanged), syntax free (allow tools to use different domain specific concrete syntax) and unambiguous.

There are two opposite approaches for defining model interchange formats. In the *semantic free approach* the interchange format is nothing more than a common transfer format for models. In this case model transformers (semantic translators) must provide a pairwise mapping among the tool models based on their shared portion of the semantics. In the *semantically inclusive approach*, a common modeling language and transfer format is defined for model interchange. This has broad enough semantics to allow exporting and importing individual tool models to and from this shared language.

6.1 Semantic-free and semantically-inclusive interchange formats in EDA

In the early 1980s, the Integrated Circuit community observed a proliferation of tools from different companies and for different purposes. Given the relative immaturity of EDA, and driven by the necessity of maintaining market share, each EDA company based its set of tools on proprietary representations whose details were not known to other companies. In addition, the largest IC companies had significant internal EDA investments; their tools were incompatible with each other and with the EDA vendors' offerings making the construction of complete design flows technically very challenging if not impossible.

In 1983, representatives of the major IC companies, of some EDA companies and of the University of California at Berkeley formed the Electronic Design Interchange Format (EDIF) Steering Committee with the intent of defining a standard format for interchanging design information across EDA tools. EDIF was semantically "free" and defined exclusively the syntax of the interchange format. After the definition of the interchange format, each company started developing translators to write and read designs. Besides limitations in the expressiveness of the chosen syntax, the main problem with the early versions of EDIF was the ambiguity of the language whose free interpretation lead to the definition of many flavors of the same standard. The meaning of an EDIF description was indeed **encoded in the translators**. To solve this problem, the EDIF Committee realized that such ambiguities had to be ruled out by giving a more precise semantics to EDIF. This is why, in the latest version of the interchange format, an information model is attached to a description. The information model is described in the formal language EXPRESS and has a formally defined semantics.

The Library Exchange Format/Design Exchange Format (LEF/DEF) were defined by Cadence Design Systems to exchange data across synthesis and layout tools. These formats have been recently made publicly available as part of the Open Access initiative, an important project to define a common data base and data format for EDA. The approach followed in this case is to provide also a C++ application programming interface (API) that can be used to interface tools based on these formats and, that ultimately offer a unique semantic interpretation of these formats. The user of the interchange format does not directly read or write the models but rather uses the API to import and export the necessary information.

The Berkeley Logic Interchange Format (BLIF) is a hardware description language for the hierarchical description of sequential circuits which serves as an interchange format for synthesis and verification tools. The BLIF language has a very precise semantics that can be used to define the implementation of finite state machine in terms of latches and combinational logic.

Semantically-free interchange formats are very flexible but their interpretation of the models written in such format is ambiguous. These interchange formats cannot be used to capture models in a domain like hybrid systems where there are semantic differences among tools that a translator should be able to understand for a correct translation.

Semantically-inclusive interchange formats impose a specific model. The advantage in this case is that the interpretation of a model is unambiguous. In the case of BLIF, this approach is a valuable proposition because its model (boolean algebra and state machines) is universally accepted in the field of logic synthesis and verification. Semantically-inclusive interchange formats, though, reduce the degrees of freedom of the tools that share the data using the format. This may not be acceptable today in the case of the hybrid system domain where there is a great deal of semantic differences among simulation, verification and synthesis tools. We review next what has been done in this domain and we propose a novel approach that should solve the open issues in interchange formats for hybrid systems.

6.2 The hybrid system interchange format

The definition of a *standard* interchange format among tools that deal with hybrid systems would create a fertile ground for further growth of the field and for the pervasive use of hybrid technology in industry. In the U.S., the DARPA MoBIES project made the importance of a standard interchange format very clear and supported the development of the *Hybrid Systems Interchange Format* (HSIF) as a way of fostering interactions among its participants. HSIF has been developed by G. Karsai, R. Alur and colleagues at Vanderbilt University and the University of Pennsylvania. HSIF models represent a system as a network of hybrid automata. Each hybrid automaton is a finite state machine in which states include constraints on continuous behaviors and transitions describe discrete steps. Automata in a network communicate by means of variables that can be of two kinds: signals and shared variables. Signals are used to model predictable execution with synchronous communication between automata. Shared variables are used for asynchronous communication between loosely coupled automata. The current HSIF specification is given in [138, 161], while a synthetic analysis of its main features can be found in [43]. HSIF is based on a semantically inclusive approach. However, in its current stage, the HSIF specification has the following unresolved issues:

(1) It is semantically too rich to become a semantic free common transfer format, but semantically too restrictive to become a common modeling language. For example, it prevents "by-construction" zero-time loops among FSMs to eliminate

the risk of non-deterministic behavior stemming out of a combination of deterministic subsystems.

(2) It is syntactically too restrictive because it lacks support for hierarchical FSMs. This can be problematic as other models often allow the creation of a hierarchical network of FSMs. For instance, exporting a HyVisual hierarchical model into HSIF requires that the hierarchy of each FSM be flattened first, a transformation that is hard to reverse.

On the other hand, it must be noted that HSIF is not a completed proposal, but rather a work in progress. It helped MoBIES researchers understand some of the fundamental problems in forming a standardized semantics for tools and some of the hard issues of having different kinds of semantics in modeling languages. The jury is still out to determine whether interchange formats will evolve toward a semantic free or semantically inclusive direction. We argue that the elimination of semantically unsound behaviors should be up to the tools, particularly the synthesis tools, and not to the interchange format. Otherwise, the format may not be able to accept the description of legitimate systems in tools where a larger set of behaviors is accepted. While we advocate that tools should be very careful in adopting liberal models, we believe that the design methodology should be enforced by tools not by interchange formats.

6.3 Requirements for a standard interchange format

To further motivate our views, we offer here some considerations about interchange formats that are the result of experience in the field of Electronic Design Automation and of a long history in participating in the formation of standard languages and models for hardware design. The following list summarizes what we believe are fundamental characteristics of any interchange format for tools and designs (a more detailed discussion can be found in [145]). An interchange format must:

- support all existing tools, modeling approaches and languages in a coherent global view of the applications and of the theory;

- support heterogeneous modeling, i.e., the ability of representing and mixing different models of computation;
- be open, i.e., be available to the entire community at no cost and with full documentation;
- support a variety of export and import mechanisms;
- support hierarchy and object orientation (compact representation, entry error prevention).

By having these properties, an interchange format can become the formal backbone for the development of sound design methodologies through the assembly of various tools. In general, a design automation flow is composed of tools that have different purposes: specification, simulation, synthesis, formal verification. Hence, they are often based on different formalisms and operate on the design at different levels of abstraction. The role of the interchange format is to facilitate the translation of design specifications from one tool to the other. As illustrated in Figure 6.1, the process of moving from the design representation used by tool A to the one used by tool B is structured in two steps: first, a representation in the standard interchange format is derived from the design entry that is used by A, then a preprocessing step is applied to produce the design entry on which B can operate. Notice that tool B may not need all the information on the design that were

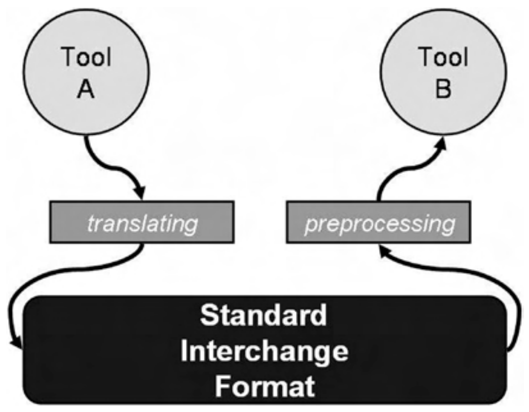

Fig. 6.1 Role of an interchange format for design tools.

used by A and, as it operates on the design, it may very well produce new data that will be written into the interchange format but that will never be used by A. Naturally, the semantics of the interchange format must be rich enough to capture and "protect" the different properties of the design at the various stages of the design process. This guarantees that there will be no loss going from one design environment to another due to the interchange format itself. The format is indeed a *neutral go-between.*

6.4 Metropolis-based abstract semantics for hybrid systems

Based on our previous discussions, we believe that *an interchange format should 1) be flexible enough to capture the largest possible class of models in use today and even tomorrow and 2) at the same time should have a precise semantics to avoid ambiguity.* Therefore, we believe that an interchange format must be based on a precise *abstract semantics* that can be refined into concrete semantics depending on the specific design tools that import/export a model.

In [145] we offered a proposal for an interchange format for hybrid systems whose formal semantics is based on the METROPOLIS Meta-Model [160]. METROPOLIS is an ambitious project supported by the GSRC (Gigascale System Research Center), CHESS (Center for Hybrid and Embedded Software Systems) and grants from industry. The idea is to provide an infrastructure based on a model with precise semantics, yet general enough to support the models of computation proposed so far and, at the same time, to allow the invention of new ones. The model, called METROPOLIS *Meta-Model* for its characteristics, is capable of not only capturing the functionality and the analysis, but also the architecture description and the mapping of functionality to architectural elements. Since the model has a precise semantics, it can be used to support a number of synthesis and formal analysis tools in addition to simulation. METROPOLIS does not dictate the use of a particular design language nor of a unified flow for all applications: the infrastructure is built so that it offers a translation path from specification languages to the metamodel. In addition, mechanisms are provided to allow the integration of external tools, thus alleviating the problems of building flows with tools that are developed independently and with different semantic models.

Metropolis proposes a design methodology for embedded system design based on the following key aspects. First of all, it leaves the designer relatively free to use the specification mechanism (graphical or textual language) of choice, as long as it has a sound semantic foundation (*model of computation* [65, 126]). Secondly, the same formalism is used to represent both the embedded system and some abstract relevant characteristics of its environment and implementation platform [149]. Finally, it separates orthogonal aspects [109], such as: computation vs. communication, functionality vs. architecture, behavior vs. performance indices. This separation results in better re-use, because it decouples independent aspects, that would otherwise be tied, e.g., a given functional specification to low-level implementation details, or to a specific communication paradigm, or to a scheduling algorithm. These techniques, combined, also facilitate the extensive use of synthesis, system-level simulation, and formal verification techniques in order to speed up the design cycle.

A detailed discussion of Metropolis can be found in [26, 27]. The complete definition of the metamodel is given in [160]. Finally [124] discusses the modeling of architectural resources in Metropolis.

The main challenge in defining an interchange format is to define a language with a formal semantics that remains general enough as it provides an easy translation path to/from all other languages of interest. In our proposal [145], the interchange format defines *processes* for the solution of equations and *media* for communicating results among processes. These are organized as a network that consists of several layers, each corresponding to a particular aspect of the hybrid computation, such as the discrete dynamics, the continuous dynamics and the specific equations involved in the description. However, while the meta-model semantics provides the basis for interpreting and evaluating the model, *the precise semantics of the network is left unspecified.* This is an essential aspect of the language architecture. To complete the description of the model, the user enters a separate view, which consists of a collection of schedulers that control the evolution of the network of processes, thereby describing the way in which the computation is performed. Because this view is also written using the Meta-Model, *the semantics of the model is part of the interchange format itself*, and is therefore accessible to tools and translators. By doing so, users of the interchange format are not only able to describe the structure of a model, but also the particular way in which the structure should be interpreted. This trades off flexibility at the expense of some additional complexity in the description of a system. It must be emphasized, however, that the characterization of a hybrid model in terms of the Meta-Model must be done only once. In this sense, the idea is similar to defining interpretation schemas in extensible mark-up languages such as XML [174]. The systems that use a specific model can then share the same scheduling network.

The abstract semantics of the interchange format proposed in [145] is reported in [144]. To facilitate the customization to a specific semantics, the model designer uses generic schedulers and refines their implementation by defining the behavior of certain abstract functions that are invoked during a scheduling cycle. These, for example, have to do with the initialization, the dynamic determination of the step (or integration) size and the resolution of the equations. The interchange format has also been designed to take advantage of the intrinsic hierarchy of the system. In particular, the function that determines the current valuation of the system is partitioned among the various components, thus enhancing modularity and maintaining encapsulation. In [144], we also illustrate how the interchange format can be used to create a design flow that includes tools as diverse as HyVisual, Modelica and CheckMate.

6.5 Conclusions

In our opinion, HSIF is an excellent model for supporting clean design of hybrid systems but not a true interchange format because it does not support the models of some important hybrid systems tools and it does not allow hierarchical representations. The Simulink/Stateflow internal format could be a *de facto standard* but it is not open nor does it have features that favor easy import and export. Modelica has full support of hierarchy and of general semantics that subsumes most if not all existing languages and tools. As such, it is indeed an excellent candidate but it is not open. In addition, all of them have not been developed with the goal of supporting heterogeneous implementations. On the other hand, the METROPOLIS metamodel has generality and can be used to represent a very wide class of models of computation. It has a clear separation between communication and computation as well as architecture and function. However, while the metamodel itself is perfectly capable to express continuous time systems, there is no tool today that can manage this information in METROPOLIS. In conclusion, we believe that no approach is mature enough today to be recommended for general adoption. However, we also believe that combining and leveraging HSIF, MODELICA, and the METROPOLIS metamodel, we can push for the foundations of a standard interchange format as well as a standard design capture language where semantics is favored over syntax. Consequently, we have made a first step in this direction by proposing a new interchange format and by presenting some examples of its application to the definition of a design flow that includes HYVISUAL, MODELICA and CHECKMATE to enter the design, simulate it and formally verify its properties [145, 144]. The new interchange format is at this point a proposal, since work still needs to be done to support it with the appropriate debugging and analysis tools and with translators to and from existing tools. We are confident that a variation of our proposal will be eventually adopted by the community interested in designing embedded systems with particular emphasis on control. We are open to any suggestion and recommendation to improve our proposal.

Acknowledgements

We gratefully acknowledge the discussions on this topic with Janos Stzipanovits of Vanderbilt University and his team, Edward Lee, Jonathan Sprinkle and Shankar Sastry of UC Berkeley, Marika Di Benedetto of University of L'Aquila, Albert Benveniste of INRIA, Tiziano Villa of University of Udine, Goran Frehse of Verimag, Thao Dang of CNRS, the PARADES team, and in particular Alberto Ferrari and Andrea Balluchi. This work has been supported in part by the Columbus Project of the European Community, the GSRC (MARCO Award#: 2003-DT-660), the Artist 2 and HYCON Networks of Excellence, and by CHESS (the Center for Hybrid and Embedded Software Systems), which receives funding from the National Science Foundation (NSF award number CCF-0424422). Also, this work has been partially supported by the following companies: Cadence Design Systems, General Motors, Infineon, Intel, Pirelli, United Microelectronics, and United Technologies. Finally, the authors would like to thank the reviewers for their extensive suggestions for improving the completeness and the presentation of the work.

References

[1] R. Alur, C. Courcoubetis, and D. Dill, "Model checking in dense real time," *Information and Computation*, vol. 104, pp. 2–34, May 1993.

[2] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P. H, X. Nicollin, J. Sifakis, and S. Yovine, "The algorithmic analysis of hybrid systems," *Theoretical Computer Science*, vol. 138, pp. 3–34, February 1995.

[3] R. Alur, T. Dang, J. Esposito, R. Fierro, Y. Hur, F. Ivancic, V. Kumar, I. Lee, P. Mishra, G. Pappas, , and O. Sokolsky, "Hierarchical hybrid modeling of embedded systems," in *EMSOFT 2001: Embedded Software, First International Workshop*, (T. A. Henzinger and C. M. Kirsch, eds.), (Tahoe City, CA, USA), pp. 14–31, Springer-Verlag, 2001.

[4] R. Alur, T. Dang, J. Esposito, Y. Hur, F. Ivancic, V. Kumar, I. Lee, P. Mishra, G. J. Pappas, and O. Sokolsky, "Hierarchical modeling and analysis of embedded systems," *Proceedings of the IEEE*, vol. 91, pp. 11–28, January 2003.

[5] R. Alur, A. Das, J. Esposito, R. Fierro, Y. Hur, G. Grudic, V. Kumar, I. Lee, J. P. Ostrowski, G. Pappas, J. Southall, J. Spletzer, and C. J. Taylor, "A framework and architecture for multirobot coordination," in *Proc. ISER00, 7^{th} Intl. Symp. on Experimental Robotics*, pp. 289–299, 2000.

[6] R. Alur and R. Grosu, "Modular refinement of hierarchic reactive machines," in *Principles of Programming Languages*, pp. 390–402, ACM Press, 2000.

[7] R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee, "Modular refinement of hierarchic reactive machines," in *Proc. of the 27th Annual ACM Symp. on Principles of Programming Languages*, pp. 390–402, 2000.

[8] R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee, "Modular specification of hybrid systems in Charon," in *Proc. of the Third Intl. Work. on Hybrid*

Systems: Computation and Control, (N. A. Lynch and B. H. Krogh, eds.), pp. 6–19, Springer-Verlag, 2000.

[9] R. Alur and T. A. Henzinger, “Modularity for timed and hybrid systems,” in *CONCUR ’97: Eight International Conference on Concurrency Theory*, pp. 74–88, Springer-Verlag, 1997.

[10] R. Alur, T. A. Henzinger, and P. H. Ho, “Automatic symbolic verification of embedded systems,” in *Proc. of the 14th Annual Real-time Systems Symp.*, pp. 2–11, 1993.

[11] R. Alur, T. A. Henzinger, and P. H. Ho, “Automatic symbolic verification of embedded systems,” *IEEE Transactions on Software Engineering*, vol. 22, pp. 181–201, March 1996.

[12] R. Alur, T. A. Henzinger, and E. D. Sontag, eds., *Hybrid Systems III: Verification and control, Proceedings of the DIMACS/SYCON Workshop, October 22-25, 1995, Rutgers University, New Brunswick, NJ, USA*, Springer, 1996.

[13] R. Alur, S. Kannan, and S. La Torre, “Polyhedral flows in hybrid automata,” in *HSCC ’99: Proceedings of the Second International Workshop on Hybrid Systems*, (F. W. Vaandrager and J. H. van Schuppen, eds.), (London, UK), pp. 5–18, Springer-Verlag, 1999.

[14] R. Alur and G. J. Pappas, eds., *Hybrid Systems: Computation and Control, 7th International Workshop, HSCC 2004, Philadelphia, PA, USA, March 25–27., 2002*, Springer, 2004.

[15] A. Angermann, M. Beuschel, M. Rau, and U. Wohlfarth, *MATLAB, Simulink, Stateflow: grundlagen, toolboxen, beispiele (MATLAB, Simulink, Stateflow: fundamentals, toolboxes, examples)*. Oldenbourg-Verlag, 2003.

[16] M. Antoniotti and A. Gollu, “SHIFT and Smart AHS: A language for hybrid system engineering modeling and simulation,” in *Proceedings of the Conference on Domain-Specific Languages*, (Santa Barbara, CA, USA), Oct. 15-17 1997.

[17] P. J. Antsaklis, “Special issue on hybrid systems: Theory and applications,” *Proc. of the IEEE*, vol. 88, July 2000.

[18] P. J. Antsaklis, W. Kohn, A. Nerode, and S. Sastry, eds., *Hybrid Systems II*, Springer, 1995.

[19] P. J. Antsaklis, W. Kohn, A. Nerode, and S. Sastry, eds., *Hybrid Systems IV*, Springer, 1997.

[20] P. J. Antsaklis and A. Nerode, “Hybrid control systems, special issue,” *IEEE Transactions on Automatic Control*, vol. 43, April 1998.

[21] E. Asarin, O. Bournez, T. Dang, , and O. Maler, “Approximate reachability analysis of piecewise linear dynamical systems,” in *HSCC 00: Hybrid Systems—Computation and Control*, (B. Krogh and N. Lynch, eds.), pp. 20–31, Springer-Verlag, 2000.

[22] E. Asarin, O. Bournez, T. Dang, O. Maler, and A. Pnueli, “Effective synthesis of switching controllers for linear systems,” *Proceedings of the IEEE*, vol. 88, pp. 1011–1025, July 2000.

[23] E. Asarin, T. Dang, and O. Maler, “d/dt: A verification tool for hybrid systems,” in *Proc. of the 40th IEEE Conf. on Decision and Control*, pp. 2893–2898, 2001.

[24] E. Asarin, T. Dang, and O. Maler, "The d/dt tool for verification of hybrid systems," in *Proc. of the 14th Intl. Conf. on Computer-Aided Verification*, pp. 365–370, 2002.

[25] R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill, "Possibly not closed convex polyhedra and the Parma Polyhedra Library," 2002.

[26] F. Balarin, H. Hsieh, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, and Y. Watanabe, "Metropolis: An integrated design environment for electronic system design," *IEEE Micro*, vol. 36, pp. 45–52, April 2003.

[27] F. Balarin, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, M. Sgroi, and Y. Watanabe, "Modeling and designing heterogeneous systems," Tech. Rep. 2002/01, Cadence Berkeley Laboratories, January 2002.

[28] A. Balluchi, A. Casagrande, P. Collins, A. Ferrari, T. Villa, and A. L. Sangiovanni-Vincentelli, "Ariadne: A framework for reachability analysis of hybrid automata," in *17th International Symposium on Mathematical Theory of Networks and Systems (MTNS)*, July 2006.

[29] A. Bemporad, F. Borrelli, and M. Morari, "Piecewise linear optimal controllers for hybrid systems," in *American Control Conference*, (Chicago, USA), pp. 1190–1194, 2000.

[30] A. Bemporad, F. Borrelli, and M. Morari, "On the optimal control law for linear discrete time hybrid systems," in *International Workshop on Hybrid Systems: Computation and Control*, (Stanford, California, USA), pp. 105–119, 2002.

[31] A. Bemporad and M. Morari, "Control of systems integrating logic, dynamics, and constraints," *Automatica*, vol. 35, pp. 407–427, March 1999.

[32] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL—a tool suite for automatic verification of real-time systems," in *Hybrid Systems III*, pp. 208–219, Springer-Verlag, 1996.

[33] A. Benveniste, "Compositional and uniform modelling of hybrid systems," *IEEE Transactions on Automatic Control*, vol. 43, pp. 579–584, April 1998.

[34] A. Benveniste and P. Le Guernic, "Hybrid dynamical systems theory and the signal language," *IEEE Transactions on Automatic Control*, vol. 35, pp. 535–546, May 1990.

[35] V. Bertin, E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine, "Taxis = Esterel + Kronos: A tool for verifying real-time properties of embedded systems," in *Proc. of the 40th IEEE Conf. on Decision and Control*, Springer-Verlag, 2001.

[36] D. Bobrow, G. Kiczales, and J. Rivieres, *The Art of the Metaobject Protocol.* MIT Press, 1991.

[37] G. Bobrow, L. Demichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon, "Common Lisp object specification," *Lisp and Symbolic Computation*, vol. 1, January 1989.

[38] G. Booch, I. Jacobson, and J. Rumbaugh, *Unified Modeling Language User Guide.* Addison Wesley, 1997.

[39] O. Botchkarev and S. Tripakis, "Verification of hybrid systems with linear differential inclusions using ellipsoidal approximations," in *HSCC*, pp. 73–88, 2000.

[40] O. Bournez, O. Maler, and A. Pnueli, "Orthogonal polyhedra: representation and computation," in *HSCC '99: Proceedings of the Second International Workshop on Hybrid Systems*, (F. W. Vaandrager and J. H. van Schuppen, eds.), (London, UK), pp. 46–60, Springer-Verlag, 1999.

[41] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine, "Kronos: A model-checking tool for real-time systems," in *Proc. of the 10th Intl. Conf. on Computer-Aided Verification*, (A. J. Hu and M. Y. Vardi, eds.), pp. 546–550, Springer-Verlag, 1998.

[42] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill, "Symbolic model checking: 10^{20} states and beyond," *Information and Computation*, vol. 98, no. 2, pp. 142–170, 1992.

[43] L. P. Carloni, M. Di Benedetto, R. Passerone, A. Pinto, and A. Sangiovanni-Vincentelli, "Modeling techniques, programming languages and design toolsets for hybrid systems," Tech. Rep., IST - Columbus Project, 2004. available at `www.columbus.gr/documents/public/WPHS/Columbus_DHS4_0.2.pdf`.

[44] A. Casagrande, A. Balluchi, L. Benvenuti, A. Policriti, T. Villa, and A. L. Sangiovanni-Vincentelli, "Improving reachability analysis of hybrid automata for engine control," in *Proc. of CDC 2004, 44th IEEE Conference on Decision and Control*, (Atlantis, Paradise Island, Bahamas), pp. 2322–2327, December 2004.

[45] P. Caspi, A. Curic, A. Maignan, C. Sofronis, and S. Tripakis, "Translating discrete-time Simulink to Lustre," in *Proc. of the Third Intl. Conf. on Embedded Software (EMSOFT). Philadelphia, PA*, (R. Alur and I. Lee, eds.), (Berlin), pp. 84–99, Springer Verlag, October 2003.

[46] A. Chutinan, *Hybrid system verification using discrete model approximations.* PhD thesis, Carnegie Mellon University, 1999.

[47] A. Chutinan and B. H. Krogh, "Computing polyhedral approximations to flow pipes for dynamic systems," in *37th IEEE Conf. on Decision and Control: Session on Synthesis and Verification of Hybrid Control Laws (TM-01)*, 1998.

[48] E. Clarke and E. Emerson, "Design and synthesis of synchronization skeletons using branching-time temporal logic," in *Workshop on Logic of Programs*, Springer-Verlag, 1981.

[49] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking.* MIT Press, 2000.

[50] E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine, "Taxys: A tool for the development and verification real-time embedded systems," in *Proc. of the 13th Intl. Conf. on Computer-Aided Verification*, Springer-Verlag, 2001.

[51] P. Collins, "Continuity and computability of reachable sets," *Theoretical Computer Science*, vol. 341, pp. 162–195, 2005.

[52] J. B. Dabney and T. L. Harman, *Mastering Simulink.* Prentice Hall, 2003.

[53] T. Dang, *Verification and synthesis of hybrid systems.* PhD thesis, INPG, 2000.

[54] T. Dang and O. Maler, "Reachability analysis via face lifting," in *HSCC 98: Hybrid Systems—Computation and Control*, (T. A. Henzinger and S. Sastry, eds.), pp. 96–109, Springer-Verlag, 1998.

[55] A. David, G. Behrmann, K. G. Larsen, and W. Yi, “A Tool architecture for the next generation of UPPAAL,” in *Proceedings of UNU/IIST 10th Anniversary Colloquium: Formal Methods at the Crossroads: from Panacea to Foundational Support*, (B. K. Aichernig and T. Maibaum, eds.), pp. 208–219, Springer-Verlag, 2002.

[56] J. Davis, M. Goel, C. Hylands, B. Kienhuis, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, and Y. Xiong, “Overview of the Ptolemy project,” Tech. Rep. UCB/ERL M99/37, Univ. of California at Berkeley, 1999.

[57] C. Daws, A. Olivero, S. Tripakis, and S. Yovine, “The tool Kronos,” in *Hybrid Systems III*, pp. 208–219, Springer-Verlag, 1996.

[58] A. Deshpande, D. Godbole, A. Gollu, L. Semenzato, R. Sengupta, D. Swaroop, and P. Varaiya, “Automated highway system tool interface format,” Tech. Rep., California PATH Technical Report, January 1996.

[59] A. Deshpande, D. Godbole, A. Gollu, and P. Varaiya, “Design and evaluation tools for automated highway systems,” in *Hybrid Systems III*, Springer-Verlag, 1996.

[60] A. Deshpande, A. Gollu, and P. Varaiya, “Shift: A formalism and a programming language for dynamic networks of hybrid automata,” in *Hybrid Systems IV*, pp. 113–134, Springer-Verlag, 1997.

[61] A. Deshpande, A. Gollu, and P. Varaiya, “The SHIFT programming language for dynamic networks of hybrid automata,” *IEEE Transactions on Automatic Control*, vol. 43, pp. 584–7, April 1998.

[62] A. Deshpande and P. Varaiya, “Viable control of hybrid systems,” in *Hybrid Systems II*, Springer-Verlag, 1995.

[63] M. D. Di Benedetto and A. L. Sangiovanni-Vincentelli, eds., *Hybrid Systems: Computation and Control, 4th International Workshop, HSCC 2001, Rome, Italy, March 28-30, 2001*, Springer, 2001.

[64] R. Djenidi, C. Lavarenne, R. Nikoukhah, Y. Sorel, and S. Steer, “From hybrid simulation to real-time implementation,” in *ESS’99 11th European Simulation Symposium and Exhibition*, pp. 74–78, October 1999.

[65] S. Edwards, L. Lavagno, E. Lee, and A. Sangiovanni-Vincentelli, “Design of embedded systems: Formal models, validation and synthesis,” *Proc. of the IEEE*, vol. 85, pp. 366–390, March 1997.

[66] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, J. Ludwig, S. Neuendorffer, S. Sachs, and Y. Xiong, “Taming heterogeneity—The Ptolemy approach,” *Proceedings of the IEEE*, vol. 91, pp. 127–144, January 2003.

[67] H. Elmqvist, “Dymola - user’s manual,” Tech. Rep., DynaSim AB, Research Park Ideon, Lund, Sweden, 1993.

[68] F. Eskafi, D. Khorramabadi, and P. Varaiya, “Design and evaluation tools for automated highway systems,” *Transpn. Res. - C*, vol. 3, no. 1, pp. 1–17, 1995.

[69] G. Frehse, “PHAVer: Algorithmic verification of hybrid systems past HyTech,” in *HSCC*, pp. 258–273, 2005.

[70] P. Fritzson, *Principles of Object-oriented Modeling and Simulation with Modelica 2.1.* J. Wiley & Sons, 2004.

[71] P. Fritzson and V. Engelson, "Modelica - a unified object-oriented language for system modeling and simulation," in *ECCOP '98: Proc. of the 12th Eur. Conf. on Object-Oriented Programming*, (London, UK), pp. 67–90, Springer-Verlag, 1998.

[72] A. Girard and G. J. Pappas, "Approximate bisimulations for constrained linear systems," in *Proc. of the 44th IEEE Conf. on Decision and Control*, December 2005.

[73] A. Girard and G. J. Pappas, "Approximate bisimulations for nonlinear dynamical systems," in *Proc. of the 44th IEEE Conf. on Decision and Control*, December 2005.

[74] D. Godbole, J. Lygeros, E. Singh, A. Deshpande, and E. Lindsey, "Design and verification of communication protocols for degraded modes of operation of AHS," in *Conference on Decision and Control*, IEEE, 1995.

[75] A. Gollu, *Object management systems.* PhD thesis, UC Berkeley, 1995.

[76] A. Gollu and P. Varaiya, "Smart AHS: A simulation framework for automated vehicles and highway systems," *Mathematical and Computer Modeling*, vol. 27, pp. 103–28, May-June 1998.

[77] C. Gomez, *Engineering and Scientific Computing with Scilab.* Birkhauser Verlag, 1999.

[78] T. Grandpierre, C. Lavarenne, and Y. Sorel, "Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors," in *Proc. of CODES'99*, pp. 74–78, 1999.

[79] T. Grandpierre and Y. Sorel, "From algorithm and architecture specifications to automatic generation of distributed real-time executives: A seamless flow of graphs transformations," in *MEMOCODE2003, Formal Methods and Models for Codesign Conference*, p. 123, June 2003.

[80] R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, eds., *Hybrid systems*, Springer, 1993.

[81] J. Haddon, D. Godbole, A. Deshpande, and J. Lygeros, "Verification of hybrid systems: monotonicity in the AHS control system," in *Proceedings of the DIMACS/SYCON workshop on Hybrid systems III : verification and control*, (Secaucus, NJ, USA), pp. 161–172, Springer-Verlag New York, Inc., 1996.

[82] N. Halbwachs, *Synchronous Programming of Reactive Systems.* Kluwer Academic Publishers, 1993.

[83] N. Halbwachs, P. Raymond, and Y.-E. Proy, "Verification of linear hybrid systems by means of convex approximation," in *SAS 94: Static Analysis Symposium*, (B. LeCharlier, ed.), pp. 233–237, Springer-Verlag, 1994.

[84] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, p. 231:274, July 1987.

[85] W. P. M. H. Heemels, B. De Schutter, and A. Bemporad, "Equivalence of hybrid dynamical models," *Automatica*, vol. 37, pp. 1085–1091, July 2001.

[86] T. Henzinger and P. H. Ho, "A note on abstract-interpretation strategies for hybrid automata," in *Hybrid Systems II*, (P. Antsaklis, A. Nerode, W. Kohn, and S. Sastry, eds.), pp. 252–264, Springer-Verlag, 1995.

[87] T. Henzinger and P. H. Ho, “HyTeCh: The Cornell hybrid technology tool,” in *Hybrid Systems II*, (P. Antsaklis, A. Nerode, W. Kohn, and S. Sastry, eds.), pp. 265–293, Springer-Verlag, 1995.

[88] T. Henzinger, P. H. Ho, and H. Wong-Toi, “A user guide to HyTeCH,” in *TACAS 95: Tools and Algorithms for the Construction and Analysis of Systems*, (E. Brinksma, W. Cleaveland, K. Larsen, T. Margaria, and B. Steffen, eds.), pp. 41–71, Springer-Verlag, 1995.

[89] T. Henzinger, P. H. Ho, and H. Wong-Toi, “Algorithmic analysis of nonlinear hybrid systems,” *IEEE Transactions on Automatic Control*, vol. 43, pp. 540–554, April 1998.

[90] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, “Symbolic model checking for real-time systems,” *Information and Computation*, vol. 111, no. 2, pp. 193–244, 1994.

[91] T. Henzinger, J. Preussig, and H. Wong-Toi, “Some lessons from the HyTech experience,” in *Proc. of the 40th IEEE Conf. on Decision and Control*, pp. 2886–2892, 2001.

[92] T. A. Henzinger, “The theory of hybrid automata,” in *Logic in Computer Science*, pp. 278–292, IEEE Computer Society Press, 1996.

[93] T. A. Henzinger, “Masaccio: A formal model for embedded components,” in *TCS 00: Theoretical Computer Science*, (J. van Leeuwen, O. Watanabe, M. Hagiya, P. D. Mosses, and T. Ito, eds.), pp. 549–563, Springer-Verlag, 2000.

[94] T. A. Henzinger and P.-H. Ho, “Model checking strategies for linear hybrid systems,” in *Proc. Workshop on Hybrid Systems and Autonomous Control*, (Ithaca, NY), 1994.

[95] T. A. Henzinger, P. H. Ho, and H. Wong-Toi, “HyTech: A model checker for hybrid systems,” *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1–2, pp. 110–122, 1997.

[96] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya, “What’s decidable about hybrid automata?,” in *Proc. 27th Annual ACM Symp. on Theory of Computing (STOC)*, pp. 373–382, 1995.

[97] T. A. Henzinger, M. Minea, and V. Prabhu, “Assume-guarantee reasoning for hierarchical hybrid systems,” in *HSCC 01: Hybrid Systems—Computation and Control*, (M. di Benedetto and A. Sangiovanni-Vincentelli, eds.), pp. 275–290, Springer-Verlag, 2001.

[98] T. A. Henzinger and S. Sastry, eds., *Hybrid Systems: Computation and Control, First International Workshop, HSCC’98, Berkeley, California, USA, April 13-15, 1998*, Springer, 1998.

[99] C. A. R. Hoare, *Communicating Sequential Processes.* Prentice-Hall, 1985.

[100] G. J. Holzmann, “The Model Checker SPIN,” *Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[101] G. P. Hong and T. G. Kim, “The DEVS formalism: A framework for logical analysis and performance,” in *Fifth Annual Conference on AI, Simulation and Planning in High Autonomy Systems*, (Gainesville, Florida), pp. 170–278, 1994.

[102] A. Hsu, F. Eskafi, S. Sachs, and P. Varaiya, "Protocol design for and automated highway system," in *Discrete Event Dynamic Systems: Theory and Applications 2*, pp. 183–206, 1993.

[103] C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, and H. Zheng, "HyVisual: A hybrid system visual modeler," Tech. Rep. UCB/ERL M03/1, UC Berkeley, 2003. available at `http://ptolemy.eecs.berkeley.edu/hyvisual/`.

[104] K. Inan and P. Varaiya, "Finitely recursive process models for discrete-event systems," *IEEE Transactions on Automatic Control*, vol. 33, pp. 626–639, July 1988.

[105] *Specification and description language SDL*. 1988.

[106] *Estelle - A formal description technique based on extended state transition model*.

[107] J. Jang, R. Teo, and C. Tomlin, "Embedded software design for the Stanford DragonFly UAV," Tech. Rep., Stanford Univ., Dept. of Aeronautics and Astronautics, 2002.

[108] J. S. Jang and C. Tomlin, "Design and implementation of a low cost, hierarchical and modular avionics architecture for the DragonFly UAVs," in *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, (Monterey), August 2002.

[109] K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, "System level design: orthogonalization of concerns and platform-based design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, pp. 1523–1543, December 2000.

[110] A. B. Kurzhanski and P. Valyi, *Ellipsoidal Calculus for Estimation and Control*. Birkhaeuser, Boston, 1997.

[111] A. B. Kurzhanski and P. Varaiya, "Ellipsoidal techniques for reachability analysis," in *HSCC 00: Hybrid Systems—Computation and Control*, (B. Krogh and N. Lynch, eds.), pp. 202–214, Springer-Verlag, 2000.

[112] A. B. Kurzhanski and P. Varaiya, "Ellipsoidal techniques for reachability analysis: Internal approximation," *Systems and Control Letters*, vol. 41, pp. 201–211, 2000.

[113] A. B. Kurzhanski and P. Varaiya, "On ellipsoidal techniques for reachability analysis," *Optimization Methods and Software*, vol. 17, pp. 177–237, 2000.

[114] A. B. Kurzhanski and P. Varaiya, "On ellipsoidal techniques for reachability analysis - Part I: external approximations," *Optimization Methods and Software*, vol. 17, no. 2, pp. 177–206, 2002.

[115] A. B. Kurzhanski and P. Varaiya, "On ellipsoidal techniques for reachability analysis - Part II: internal approximations based-valued constraints," *Optimization Methods and Software*, vol. 17, no. 2, pp. 207–337, 2002.

[116] A. B. Kurzhanski and P. Varaiya, "Reachability analysis for uncertain systems - the ellipsoidal technique," *Dynamics of Continuous, Discrete and Impulsive Systems*, vol. 9, no. 3, pp. 347–367, 2002.

[117] A. B. Kurzhanski and P. Varaiya, "Ellipsoidal techniques for hybrid dynamics: the reachability problem," in *New Directions and Applications in Control Theory*, (A. Lindquist, W. Dayawensa, and Y. Zhou, eds.), pp. 193–206, Springer-Verlag, 2005.

[118] A. B. Kurzhanski and P. Varaiya, “On verification of controlled hybrid dynamics through ellipsoidal techniques,” in *Proceedings of the 44th IEEE Conference on Decision and Control, and the European Control Conference 2005, Seville, Spain*, pp. 4682–4686, December 2005.

[119] A. A. Kurzhanskiy and P. Varaiya, “Ellipsoidal techniques for reachability analysis of discrete-time linear systems,” *IEEE Transactions on Automatic Control*, Submitted for Publication, June 2005.

[120] A. A. Kurzhanskiy and P. Varaiya, *Ellipsoidal toolbox - technical report.* University of California, Berkeley, http://www.eecs.berkeley.edu/~akurzhan/ellipsoids, 2006.

[121] M. Kvasnica, P. Grieder, M. Baotic, and M. Morari, “Multi-parametric toolbox (MPT).,” in *HSCC*, pp. 448–462, 2004.

[122] G. Lafferriere, G. J. Pappas, and S. Yovine, “A new class of decidable hybrid systems,” in *HSCC ’99: Proceedings of the Second International Workshop on Hybrid Systems*, (F. W. Vaandrager and J. H. van Schuppen, eds.), (London, UK), pp. 137–151, Springer-Verlag, 1999.

[123] L. Lamport, “Proving the correctness of multiprocess programs,” *IEEE Transactions on Software Engineering*, vol. 2, pp. 125–143, February 1977.

[124] L. Lavagno, J. Moondanos, T. Meyerowitz, and Y. Watanabe, “Modeling of architectural resources in Metropolis,” Internal Document, Cadence, 2002.

[125] E. A. Lee and S. Neuendorffer, “Concurrent models of computation for embedded software,” *IEE Proceedings*, vol. 153, pp. 239–250, March 2005.

[126] E. A. Lee and A. L. Sangiovanni-Vincentelli, “A framework for comparing models of computation,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, pp. 1217–1229, December 1998.

[127] E. A. Lee and Y. Xiong, “System-level types for component-based design,” in *Embedded Software. Proceeding of the First International Workshop, EMSOFT 2001. Tahoe City, CA*, (T. A. Henzinger and C. M. Kirsch, eds.), (Berlin), pp. 237–253, Springer Verlag, October 2001.

[128] E. A. Lee and H. Zheng, “Operational semantics of hybrid systems,” in *HSCC*, pp. 25–53, 2005.

[129] J. Lygeros, C. Tomlin, and S. Sastry, “Controllers for reachability specifications for hybrid systems,” in *Automatica, Special Issue on Hybrid Systems*, 1999.

[130] N. Lynch, R. Segala, F. Vaandrager, and H. Weinberg, “Hybrid I/O automata,” in *Hybrid Systems III: Verification and Control*, pp. 496–510, Springer-Verlag, 1996.

[131] N. A. Lynch and B. H. Krogh, eds., *Hybrid Systems: Computation and Control, Third International Workshop, HSCC 2000, Pittsburgh, PA, USA, March 23-25, 2000*, Springer, 2000.

[132] O. Maler, ed., *Hybrid and Real-Time Systems, International Workshop. HART’97, Grenoble, France, March 26-28, 1997, Proceedings*, Springer, 1997.

[133] O. Maler, Z. Manna, and A. Pnueli, “From timed to hybrid systems,” in *Real-Time: Theory in Practice, REX Workshop*, pp. 447–484, Springer-Verlag, 1991.

[134] O. Maler and A. Pnueli, eds., *Hybrid Systems: Computation and Control, 6th International Workshop, HSCC 2003 Prague, Czech Republic, April 3–5, 2003*, Springer, 2003.

[135] Matisse, "Available at `http://wiki.grasp.upenn.edu/~graspdoc/hst/`,".

[136] K. L. McMillan, *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[137] R. Milner, *A Calculus of Communicating Systems. Lecture Notes in Computer Science*, Springer-Verlag, 1980.

[138] MoBIES Group, "HSIF syntax (version 3)," Internal Document, Vanderbilt University, October 22, 2002.

[139] Modelica Association, "Modelica - A unified object-oriented language for physical systems modeling, language specification," December 2000.

[140] M. Morari and L. Thiele, eds., *Hybrid Systems: Computation and Control, 8th International Workshop, HSCC 2005, Zurich, Switzerland, March 9–11, 2005*, Springer, 2005.

[141] P. Mosterman, "On the normal component of centralized frictionless collision sequences," *ASME Journal of Applied Mechanics*, 2005.

[142] S. Neema, "Analysis of Matlab Simulink and Stateflow data model," Tech. Rep. ISIS 01-204, Vanderbilt University, Nashville, TN, March 2001.

[143] R. Nikoukhah and S. Steer, "SCICOS - a dynamic system builder and simulator user's guide - version 1.0," Tech. Rep. 0207, INRIA, Rocquencourt, France, June 1997.

[144] A. Pinto, L. P. Carloni, R. Passerone, and A. L. Sangiovanni-Vincentelli, "Interchange formats for hybrid systems: Abstract semantics," in *Proceedings of the The 9th International Workshop on Hybrid Systems: Computation and Control (HSCC 2006)*, (J. P. Hespanha and A. Tiwari, eds.), (Santa Barbara, California), pp. 491–506, Springer-Verlag, March 2006.

[145] A. Pinto, A. L. Sangiovanni-Vincentelli, L. P. Carloni, and R. Passerone, "Interchange formats for hybrid systems: Review and proposal," in *Proceedings of the The 8th International Workshop on Hybrid Systems : Computation and Control (HSCC 2005)*, (M. Morari, L. Thiele, and F. Rossi, eds.), pp. 526–541, March 2005.

[146] A. Puri and P. Varaiya, "Driving safely in smart cars," in *American Control Conference*, pp. 3597–3599, 1995.

[147] S. Ratschan and Z. She, "Safety verification of hybrid systems by constraint propagation based abstracition refinement," in *HSCC*, pp. 573–589, 2005.

[148] M. Rivoire and J. L. Ferrier, *MATLAB Simulink Stateflow avec des Exercices d'Automaticue Rsolus (MATLAB Simulink Stateflow with Solved Exercises in Automatic Control).* Editions TECHNIP, 2001.

[149] A. L. Sangiovanni-Vincentelli, "Defining platform-based design," in *EEDesign. Available at www.eedesign.com/story/OEG20020204S0062)*, February 2002.

[150] L. Semenzato, A. Deshpande, and A. Gollu, "Shift reference manual," Tech. Rep., California PATH, June 1996.

[151] B. I. Silva, K. Richeson, B. Krogh, and A. Chutinan, "Modeling and verifying hybrid dynamic systems using CheckMate," in *Proceedings of 4th International Conference on Automation of Mixed Processes*, pp. 323–328, September 2000.

[152] B. I. Silva, O. Stursberg, B. H. Krogh, and S. Engell, "An assessment of the current status of algorithmic approaches to the verification of hybrid systems," in *Proc. of the 40th IEEE Conf. on Decision and Control*, pp. 2867 – 2874, 2001.

[153] T. Simsek, "SHIFT tutorial: A first course for SHIFT programmers," Tech. Rep., UC Berkeley, 1999.

[154] T. Simsek, "The λ-SHIFT specification language for dynamic networks of hybrid automata," Tech. Rep., UC Berkeley, 2000.

[155] E. D. Sontag, "Nonlinear regulation: the piecewise linear approach," *IEEE Transactions on Automatic Control*, vol. 26, pp. 346–357, April 1981.

[156] Y. Sorel, "Massively parallel computing systems with real time constraints - the "Algorithm Architecture Adequation" methodology," in *Massively Parallel Computing Systems Conference*, pp. 44–54, 1994.

[157] G. Steele, *Common Lisp: The Language.* Digital Press, Second edition ed., 1990.

[158] D. Stipanovic, G. Inalhan, and C. Tomlin, "Decentralized overlapping control of a formation of unmanned aerial vehicles," in *Proceedings of the 41st IEEE Conference on Decision and Control*, (Las Vegas, NV), December 2002.

[159] O. Stursberg, S. Kowalewski, J. Preussig, and H. Treseler, "Block-diagram based modelling and analysis of hybrid processes under discrete control," *Journal Europeen des Systemes Automatises (JESA)*, vol. 32, no. 9-10, pp. 1097–1118, 1998.

[160] The Metropolis Project Team, "The Metropolis meta model version 0.4," Tech. Rep. UCB/ERL M04/38, University of California, Berkeley, September 2004.

[161] The University of Pennsylvania MoBIES Group, "HSIF semantics (version 3, synchronous edition)," Internal Document, The University of Pennsylvania, August 22, 2002.

[162] M. M. Tiller, *Introduction to physical modeling with Modelica.* Kluwer Academic Publishers, 2001.

[163] C. Tomlin and M. R. Greenstreet, eds., *Hybrid Systems: Computation and Control, 5th International Workshop, HSCC 2002, Stanford, CA, USA, March 25-27, 2002*, Springer, 2002.

[164] F. D. Torrisi and A. Bemporad, "Discrete-time hybrid modeling and verification," in *Proc. of the 40th IEEE Conf. on Decision and Control*, pp. 2899 – 2904, 2001.

[165] F. D. Torrisi and A. Bemporad, "HYSDEL - a tool for generating computational hybrid models for analysis and synthesis problems," *IEEE Transactions on Control Systems Technology*, vol. 12, pp. 235–249, March 2004.

[166] F. D. Torrisi, A. Bemporad, G. Bertini, P. Hertach, D. Jost, and D. Mignone, "Hysdel 2.0.5 - user manual," Tech. Rep., ETH Zurich, 2002.

[167] A. C. Uselton and S. A. Smolka, "A compositional semantics for Statecharts using labeled transition systems," in *Concurrency Theory*, pp. 2–17, Springer-Verlag, 1994.

[168] F. W. Vaandrager and J. H. van Schuppen, eds., *Hybrid Systems: Computation and Control, Second International Workshop, HSCC'99, Bergen Dal, The Netherlands, 1999*, Springer, 1999.

[169] P. Varaiya, "Smart cars on smart roads: problems of control," *IEEE Transactions on Automatic Control*, vol. 38, pp. 195–207, February 1993.

[170] P. Varaiya, "Reach set computation using optimal control," in *KIT Workshop on Verification of Hybrid Systems, Grenoble, France*, October 1998.

[171] S. M. Veres, *User's manual - reference of the geometric bounding toolbox - version 7.3.* SysBrain Ltd, Southampton, United Kingdom, url = http://sysbrain.com/gbt/, March 2004.

[172] K. Weihrauch, *Computable Analysis - An Introduction. Texts in Theoretical Computer Science*, Berlin: Springer-Verlag, 2000.

[173] S. Wolfram, *The* MATHEMATICA *book, fifth edition.* Wolfram Media, 2003.

[174] XML, "see `http://www.w3.org/XML/`,".

[175] B. Zeigler, *Multifaceted Modeling and Discrete Event Simulation.* Academic Press, London, 1984.