HyTech: The Next Generation[*][†]

Thomas A. Henzinger Pei-Hsin Ho Howard Wong-Toi

Computer Science Department
Cornell University
Ithaca, NY 14853

(hytech|tah|ho|howard)@cs.cornell.edu

Abstract. We describe a new implementation of HyTech[1], a symbolic model checker for hybrid systems. Given a parametric description of an embedded system as a collection of communicating automata, HyTech automatically computes the conditions on the parameters under which the system satisfies its safety and timing requirements. While the original HyTech prototype was based on the symbolic algebra tool Mathematica, the new implementation is written in C++ and builds on geometric algorithms instead of formula manipulation. The new HyTech offers a cleaner and more expressive input language, greater portability, superior performance (typically two to three orders of magnitude), and new features such as diagnostic error-trace generation. We illustrate the effectiveness of the new implementation by applying HyTech to the automatic parametric analysis of the generic railroad crossing benchmark problem [HJL93] and to an active structure control algorithm [ECB94].

1 Introduction

There has been increasing use of embedded software and hardware for controlling physical systems in real time. Many of these embedded controllers occur in safety-critical applications where correctness is essential. However, traditional methods of program analysis are not immediately applicable to such *hybrid systems*, which involve a mixture of both discrete and continuous components. *Model checking* is an algorithmic verification technique that determines whether an automaton model of a discrete finite-state system satisfies its temporal-logic requirements. Model checking has been extended to real-time systems that are modeled as *timed automata* [AD94, HNSY94], and to hybrid systems that are modeled as *linear hybrid automata* [ACHH93, ACH$^+$95]. In these cases, the state space is infinite, and the formal analysis must proceed symbolically (*i.e.* not enumeratively), by describing infinite state sets using linear constraints.

[*]This research was supported in part by the ONR YIP award N00014-95-1-0520, by the NSF CAREER award CCR-9501708, by the NSF grants CCR-9200794 and CCR-9504469, by the AFOSR contract F49620-93-1-0056, and by the ARPA grant NAG2-892.

[†]A preliminary version of this paper appeared in the *Proceedings of the 16th Annual IEEE Real-time Systems Symposium* (RTSS 1995), pp. 56–65.

[1]The tool HyTech, together with its user guide, is publicly available through the World-Wide Web via the URL http://www.cs.cornell.edu/Info/People/tah/hytech.html.

We present a completely new implementation of the symbolic model checker HyTech for linear hybrid automata. Previously, we developed a prototype to test the feasibility of this approach [AHH93, HH95a]. The prototype was written in Mathematica for rapid development and easy experimentation, at the cost of portability and performance. Motivated by the prototype's success, we have reimplemented the tool. The new version offers a marked improvement in efficiency, convenience, and generality. First, infinite state sets are represented as polyhedra in multidimensional real space, instead of Mathematica formulas. Polyhedra provide a uniform representation that is implemented, entirely in C++, using standard data structures and geometric algorithms for manipulation [Che68]. For example, for computing the set of states that can be reached by a time delay, we compute the "shadow" of a polyhedron (which is easy) instead of eliminating an existential quantifier from a Mathematica formula (which is expensive). Our reimplementation uses an efficient C++ library of polyhedral operations [Hal93, HRP94]. Second, the input language is now substantially cleaner and more expressive. By incorporating generalized linear conditions on the rates of analog variables, nondeterministic and simultaneous assignments, and urgent transitions into the input language, we allow the specification and analysis of a wider class of hybrid systems. Careless system descriptions can be caught quickly by the parser. Third, there is an enriched and user-programmable command language which includes macros for common verification tasks such as reachability analysis and diagnostic error-trace generation. Last, the new verifier is portable, because it no longer relies on Mathematica.

We have recomputed more than 20 case studies that had been analyzed with the HyTech prototype [AHH93, ACH+95, HH95a, HH95b, HH95c, HWT95]. Our results show a verification-time improvement of roughly two to three orders of magnitude. For example, using our new implementation, the Philips audio control protocol [BPV94] can be analyzed in 19 seconds as opposed to 5.0 hours [HWT95].[2] Indeed, without sacrificing generality, the performance of HyTech is now comparable to automatic verifiers for more specialized types of real-time systems. Three examples of tools for the symbolic analysis of timed automata are Kronos [DY95], Veriti [DWT95], and Uppaal [LPY95]. Another verification tool for linear hybrid automata is Polka [HRP94], which focuses on abstract-interpretation techniques.

In Section 2, we give a brief review of the hybrid automaton model and corresponding analysis techniques. In Section 3, we present the new implementation of HyTech. For more detail, the user guide [HHWT95] may be consulted. In Section 4, we include two case studies that, previously, have not been formalized using hybrid automata. Neither one of these case studies was designed by us, and both were published at the last symposium in this series [HL94, ECB94].

The generic railroad crossing (GRC) problem, which is derived from the train-gate crossing of [LS85], was posed in [HJL93] as a challenge benchmark for formal methods for real-time systems. Solutions to the problem have been formally verified using a number of techniques, including modecharts [JS88], process algebras [GL90], Alur-Dill timed automata [ACD+92], machine-assisted theorem proving [Sha93], model checking [WM93], and Lynch-Vaandrager timed automata [HL94]. Using linear hybrid automata, we provide the first automatic synthesis of critical timing constraints, namely, the maximal amount of time the controller can wait before commanding the gate to lower.

The active control structure problem was posed and verified in [ECB94] using the Concurrency Workbench. A sensor and an actuator are coordinated using a pulse control algorithm to apply controls to an active structure. We use HyTech to automatically synthesize the precise lower and upper bounds on the delays between successive pulse applications.

[2]All figures obtained on a Sun 670MP.

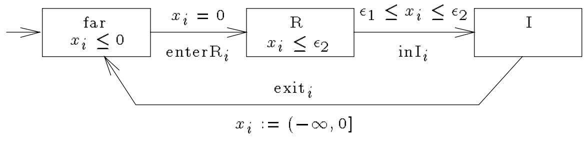

Figure 1: Train automaton T_i

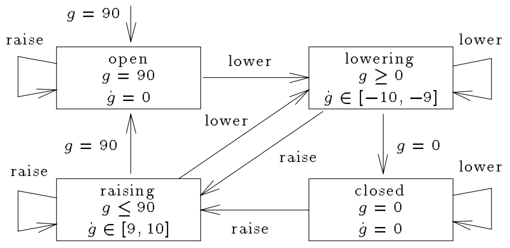

Figure 2: Gate automaton

2 Linear Hybrid Automata

We model embedded systems as the parallel composition of coordinating linear hybrid automata [ACHH93]. Informally, a linear hybrid automaton consists of a finite set X of real-valued variables and a labeled multigraph (V, E). The edges E represent discrete events, each labeled with conditions on the old and new values of X. The vertices V represent control modes, each labeled with conditions on the slopes of X. The automaton state changes either instantaneously through the action associated with a discrete event, or while time elapses, through the continuous activity associated with a control mode. Our automata are more expressive than those of the prototype [HH95a] in that we allow more general instantaneous actions (arbitrary linear conditions on old and new variable values), more general continuous activities (arbitrary linear conditions on slopes), and urgent events (which must take place as soon as they are enabled).

We use the linear hybrid automata that model the GRC problem as a running example. The GRC system consists of two trains, a gate, and a controller. The role of the controller is to ensure that the gate is always closed whenever there are trains in the track intersection, and that the gate is not closed unnecessarily long. Each train is initially some distance away from the intersection, and the gate is open. As a train approaches the intersection, it triggers a sensor signaling its upcoming entry to the controller. The controller waits until the last possible moment to send a lower command to the gate. When the train has left the intersection (and the other train is at a safe distance from the intersection), the controller commands the gate to be raised. The system is described in more detail in Section 4. The linear hybrid automata for the trains and the gate appear in Figures 1 and 2.

2.1 Definition

A *linear expression* over a set X of real-valued variables is a linear combination of variables from X with rational coefficients. A *linear inequality* over X is an inequality between linear expressions over X. A *convex predicate* over X is a conjunction of linear inequalities over X; and a *linear predicate* is a disjunction of convex predicates.

A *linear hybrid automaton* A consists of the following components.

Variables A finite ordered set $X = \{x_1, x_2, \ldots, x_n\}$ of real-valued *variables*. A *valuation* is a point $(s_1, s_2, \ldots, s_n)$ in the n-dimensional real space $\mathbb{R}^n$, or equivalently, a function that maps each variable x_i to a value s_i. A *convex zone* is a convex polyhedron in $\mathbb{R}^n$; and a *linear zone* is a finite collection of convex zones. Each convex (linear) predicate ϕ defines a convex (linear) zone $[\![\phi]\!] \subseteq \mathbb{R}^n$ such that $s \in [\![\phi]\!]$ iff $\phi[X := s]$ is true.

In the GRC example, the position of each train T_i is determined by the value of the variable x_i, which models a clock that indicates how long the train will take to reach the sensor placed ahead of the intersection. The variable g models the angle of the gate. When $g = 90$, the gate is open; when $g = 0$, it is closed.

Locations A finite set V of vertices called *locations*, used to model control modes. For example, the gate automaton has the four locations *open*, *raising*, *lowering*, and *closed*.

A *state* (v, s) of the automaton A consists of a location $v \in V$ and a valuation $s \in \mathbb{R}^n$. A *linear region* $\bigcup_{v \in V}\{(v, S_v)\}$ is a collection of linear zones $S_v \subseteq \mathbb{R}^n$, one for each location $v \in V$. A *state predicate* is a collection $\bigcup_{v \in V}\{(v, \phi_v)\}$ of linear predicates ϕ_v, one for each location $v \in V$. Each state predicate $\varphi = \bigcup_{v \in V}\{(v, \phi_v)\}$ defines the linear region $[\![\varphi]\!] = \bigcup_{v \in V}\{(v, [\![\phi_v]\!])\}$. When writing state predicates, we freely use boolean operators between state predicates, and we use the *location counter loc*, which ranges over the set V of locations. For example, we write $loc = v \vee x = 3$ for the state predicate $\{(v, \mathit{true})\} \cup \bigcup_{v' \neq v}\{(v', x = 3)\}$.

Initial condition A state predicate φ^0 called the *initial condition*. For example, the gate automaton has the initial condition $loc = \mathit{open} \wedge g = 90$.

Location invariants A labeling function *inv* that assigns to each location $v \in V$ a convex predicate $\mathit{inv}(v)$ over X, the *invariant* of v. The state (v, s) is *admissible* if $s \in \mathit{inv}(v)$. Control of A may reside in location v only while $\mathit{inv}(v)$ is satisfied, so the invariants may be used to enforce progress in the automaton. Nonconvex invariants can be modeled by splitting locations.

In the gate automaton, $\mathit{inv}(\mathit{open}) = (g = 90)$, $\mathit{inv}(\mathit{lowering}) = (g \geq 0)$, $\mathit{inv}(\mathit{raising}) = (g \leq 90)$, and $\mathit{inv}(\mathit{closed}) = (g = 0)$. The invariant at location *lowering* implies that the gate can only be lowered until it is closed, at which point control moves to location *closed*. In the graphical representation, we omit invariants of the form *true*.

Transitions A finite multiset E of edges called *transitions*, used to model discrete events. Each transition (v, v') identifies a source location $v \in V$ and a target location $v' \in V$. For example, the gate automaton has ten transitions.

Instantaneous actions A labeling function *jump* that assigns an *update set* and a *jump condition* to each transition $e \in E$. The update set $\mathit{upd}(e)$ is a subset of X. The jump condition $\mathit{jump}(e)$ is a convex predicate over $X \cup Y'$, where $Y = \{y_1, \ldots, y_k\} = \mathit{upd}(e)$, and $Y' = \{y'_1, \ldots, y'_k\}$. The variable x_i refers to its value before the transition, and the primed variable y'_i refers to the value of y_i after the transition. Only variables in $\mathit{upd}(e)$ are updated by the transition. Formally, for the transition $e = (v, v')$, the binary *transition-step relation* $\rightarrow_e$ on the admissible states is defined such that $(v,s) \rightarrow_e (v',s')$ iff (1) $jump(e)[X, Y' := s, s'[Y]]$[3] is true, and (2) for all variables $x_i \in X \setminus Y$, $s_i = s'_i$. The transition e is *enabled* at the state (v,s) if there exists a state (v',s') such that $(v,s) \rightarrow_e (v',s')$. Nonconvex jump conditions can be modeled by splitting transitions.

When writing update sets and jump conditions, we often use nondeterministic guarded assignments to closed intervals (bounded or unbounded). For example, we write $\phi \rightarrow y_i := [l,u]$ for the update set $\{y_i\}$ and the jump condition $\phi \wedge l \leq y'_i \leq u$, where l and u are linear expressions over X. This jump condition is enabled in the valuation s if the guard ϕ is satisfied, *i.e.* $s \in [\![\phi]\!]$. Then y_i is updated nondeterministically to any value in the interval $[s(l), s(u)]$, where $s(l)$ is the value of l interpreted in s. In the graphical representation, we use unlabeled edges to indicate empty update sets.

Urgency flags A partial labeling function *asap* that assigns the *urgency flag* ASAP to some transitions in E. The transitions in the domain of *asap* are called *urgent*. The state (v,s) is *urgent* if some urgent transition is enabled at (v,s).

Continuous activities A labeling function *rate* that assigns a *rate condition* to each location $v \in V$. The rate condition $rate(v)$ is a convex predicate over $\dot{X} = \{\dot{x}_1, \dot{x}_2, \ldots, \dot{x}_n\}$. The variable $\dot{x}_i$ denotes the rate of change (the first derivative) of x_i. As long as control of A resides in location v, the variables change along smooth trajectories whose first derivatives satisfy the rate condition. Formally, for the nonnegative real $\delta \in \mathbb{R}_{\geq 0}$, the *time-step relation* $\rightarrow_\delta$ on the admissible states is defined such that $(v,s) \rightarrow_\delta (v',s')$ iff (1) $v' = v$, (2) $\delta > 0$ implies (v,s) is not urgent, and (3) either $\delta = 0$ and $s' = s$, or $\delta > 0$ and $rate(v)[\dot{X} := \delta \cdot (s' - s)]$ is true. The real δ represents a delay of duration δ. Nonconvex rate conditions can be modeled by splitting locations.

In the gate automaton, $rate(open) = (\dot{g} = 0)$, $rate(raising) = (9 \leq \dot{g} \leq 10)$, $rate(closed) = (\dot{g} = 0)$, and $rate(lowering) = (-10 \leq \dot{g} \leq -9)$. The rate condition at location *lowering* implies that the gate is lowered at a rate that varies between 9 and 10 degrees per second. The variables x_i of the train automata T_i have constant rate 1, and therefore measure time. They are called clocks. Formally, the variable x is *discrete* (a *clock*) if for all locations $v \in V$, $rate(v)$ implies $\dot{x} = 0$ ($\dot{x} = 1$). The variable x is a *stopwatch* if for all locations $v \in V$, $rate(v)$ implies $\dot{x} \in \{0,1\}$. Clocks are useful for measuring delays between transitions, and stopwatches are useful for measuring durations, such as the accumulated time spent in a set of locations. The discrete variable x is a *parameter* if every jump condition implies $x' = x$. Parameters have fixed values. In the graphical representation, we omit conjuncts that specify the rates of clocks and discrete variables in the rate conditions, and we omit conjuncts that specify the jumps of parameters in the jump conditions (when the type of a variable is clear from the context).

Synchronization labels A finite set L of *synchronization labels*, and a labeling function *syn* that assigns a synchronization label from $L \cup \{\tau_A\}$ to each transition in E. The *internal label* τ_A is specific to the automaton A, and does not occur in the label set of any other automaton. The synchronization labels are used to define the parallel composition of automata. In the gate automaton, $syn(open, lowering) = lower$. In the graphical representation, τ_A is omitted.

The polyhedral library [Hal93] supports only *nonstrict* linear inequalities. This limitation imposes two restrictions on the automata that can be analyzed by the current implementation of HyTech. First, we require that each urgent transition is *invariant-enabled*, where the transition $e = (v, v')$ is invariant-enabled if e is enabled at all admissible states of the form (v,s). In this case, we call the source location v *urgent*. Second, we require that all rate conditions define bounded polyhedra. Neither restriction is required in theory.

[3]Given a valuation s, we write $s[Y]$ for the restriction of s to the variables in Y.

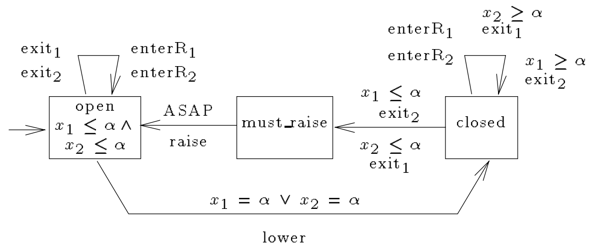

Figure 3: Controller automaton

2.2 Parallel composition

A hybrid system typically consists of several components that operate concurrently and communicate with each other. The component automata coordinate both through shared variables and synchronization labels. The automaton that models the entire system is obtained from the component automata using a product construction.

Let A_1 be the linear hybrid automaton $(X_1, V_1, \varphi_1^0, inv_1, E_1, upd_1, jump_1, asap_1, rate_1, L_1, syn_1)$, and define A_2 similarly. In the product $A_1 \times A_2$, two transitions e_1 and e_2 from the two component automata A_1 and A_2 are executed simultaneously if $syn_1(e_1) = syn_2(e_2)$. They are interleaved if $syn_1(e_1) \neq syn_2(e_2)$ and the label $syn_1(e_1)$ does not occur in A_2's label set, nor the label $syn_2(e_2)$ in A_1's label set. For reasons of efficient verification, we do not permit the "accidental" simultaneity of component transitions. In the GRC example, the system is composed of the train, gate, and controller automata of Figures 1–3. The controller communicates with the trains by synchronizing on *enter* and *exit* transitions. It issues commands to the gate on the synchronized events *raise* and *lower*.

Formally, the *product* $A_1 \times A_2$ of A_1 and A_2 is the linear hybrid automaton $A = (X_1 \cup X_2, V_1 \times V_2, \varphi_1^0 \wedge \varphi_2^0, inv, E, upd, jump, asap, rate, L_1 \cup L_2, syn)$.

- Each location (v_1, v_2) in $V_1 \times V_2$ has the invariant $inv(v_1, v_2) = inv_1(v_1) \wedge inv_2(v_2)$, and the rate condition $rate(v_1, v_2) = rate_1(v_1) \wedge rate_2(v_2)$.
- E contains the transition $e = ((v_1, v_2), (v_1', v_2'))$ if

  (1) $e_1 = (v_1, v_1') \in E_1$, $v_2' = v_2$, and $syn_1(e_1) \notin L_2$; or
  (2) $e_2 = (v_2, v_2') \in E_2$, $v_1' = v_1$, and $syn_2(e_2) \notin L_1$; or
  (3) $e_1 = (v_1, v_1') \in E_1$, $e_2 = (v_2, v_2') \in E_2$, and $syn_1(e_1) = syn_2(e_2)$.

  In case (1), $syn(e) = syn_1(e_1)$, $upd(e) = upd_1(e_1)$, $jump(e) = jump_1(e_1)$, and the partial function *asap* defines e to be urgent iff e_1 is urgent; in case (2), $syn(e) = syn_2(e_2)$, $upd(e) = upd_2(e_2)$, $jump(e) = jump_2(e_2)$, and the partial function *asap* defines e to be urgent iff e_2 is urgent; and in case (3), $syn(e) = syn_1(e_1) = syn_2(e_2)$, $upd(e) = upd_1(e_1) \cup upd_2(e_2)$, $jump(e) = jump_1(e_1) \wedge jump_2(e_2)$, and the partial function *asap* defines e to be urgent iff either e_1 or e_2 is urgent.

The product automaton A is well-defined if all urgent transitions of A are invariant-enabled, and all rate conditions of A are bounded.

2.3 Reachability and verification

Let A be a linear hybrid automaton with n variables. The state space $\Sigma \subseteq V \times \mathbb{R}^n$ of A is the set of admissible states. We define the binary *successor relation* $\rightarrow_A$ on the state space Σ as $\bigcup_{e \in E} \rightarrow_e \cup \bigcup_{\delta \in \mathbb{R}_{\geq 0}} \rightarrow_\delta$. For a region W, we define the *successor region* $post(W)$ to be the set of states that are reachable from some state in W via a single transition or time step, *i.e.* $post(W) = \{s \mid \exists s' \in W \text{ such that } s' \rightarrow_A s\}$. The region $post^*(W)$ *forward reachable* from W is the set of states reachable from W by a finite number of steps, *i.e.* $post^*(W) = \bigcup_{i \geq 0} post^i(W)$. The *predecessor region* $pre(W)$ is the set of states from which it is possible to reach a state in W via a single transition or time step. The region $pre^*(W)$ *backward reachable* from W is the infinite union $\bigcup_{i \geq 0} pre^i(W)$. If W is linear, then $post(W)$ and $pre(W)$ are also linear regions, and they can be computed effectively [ACHH93].

In practice, many verification problems can be posed in a natural way as reachability problems. Often, the system is composed with a special monitor process that "watches" the system and enters a violation state whenever the execution of the system violates a given safety or timing requirement. This technique is used in the verification and synthesis of the active control structure case study of Section 4. A state (v, s) is *initial* if (v, s) is admissible and $(v, s) \in [\![\varphi^0]\!]$. A system with initial states I is correct with respect to violation states U if $post^*(I) \cap U = \emptyset$, or equivalently, if $pre^*(U) \cap I$ is empty.

HyTech computes the forward reachable region $post^*(I)$ by finding the limit of the infinite sequence I, $post(I)$, $post^2(I), \ldots$ of regions. Analogously, the backward reachable region $pre^*(U)$ is found by iterating pre. These iteration schemes are semi-decision procedures; there is no guarantee of termination. However, it has been shown that for a large class of hybrid systems, the *initialized rectangular automata* of [HKPV95], termination is guaranteed after a simple preprocessing step.

2.4 Parametric analysis

A system description often contains parameters. The system is incorrect for parameter values for which there exists a state in the region $post^*(I) \cap U$ [CH78, AHV93]. Thus we may obtain necessary and sufficient conditions for system correctness by performing reachability analysis followed by existential quantification over all variables that are not parameters.

Our study of the GRC demonstrates this technique. The controller decides when to issue *lower* commands to the gate based on the amount of time since each train last passed the sensor ahead of the intersection. We introduce the parameter α in the controller automaton (Figure 3) so that α corresponds to the amount of time the controller waits before a *lower* command is sent to the gate. HyTech determines that the GRC system includes violations (the gate is not closed and there is a train in the intersection) if and only if α is greater than or equal to 20, from which we deduce correctness for α strictly less than 20.

3 HyTech

There have been three generations of HyTech. The earliest prototype we developed [AHH93] was written entirely in Mathematica. Regions are represented as symbolic expressions denoting state predicates. The definition of a successor region uses existential quantification over the reals, which is easily encoded in Mathematica. Mathematica offers powerful symbolic manipulation, and allows rapid development and experimentation with algorithms and heuristics. However, its operations over state predicates are computationally inefficient; in particular, the computation of time-step successor states requires expensive quantifier elimination operations. HyTech [HH95a] was rewritten to avoid this bottleneck. The second prototype uses a Mathematica main program and computes time-step successors by calling efficient C++ routines from a library for manipulating polyhedra [Hal93]. However, this prototype requires inefficient conversions between Mathematica expressions and C++ data structures. It also still relies on Mathematica for computing transition-step successor states. The total speed-up achieved is roughly one order of magnitude. The new generation HyTech that we introduce here avoids Mathematica altogether and is built entirely in C++. It is roughly two to three orders of magnitude faster again than the second generation verifier.

The input language of our reimplementation is both cleaner and more expressive. The language consists of two parts, a system-description language, and an analysis-command language. The system-description language extends that of the prototype in a uniform way by allowing linear predicates as jump and rate conditions. This allows us to model nondeterministic and simultaneous assignments of variables (previously only deterministic assignments were allowed), linearly dependent rates of variables (previously only conjunctions of finite upper and lower bounds on individual rates were allowed), and urgent transitions (see below). The analysis-command language consists of a flexible programming language for writing system analysis scripts. It includes macros for common verification tasks such as reachability analysis and error-trace generation. In the following, we describe only a couple of aspects of each language. More detail, and formal definitions of both languages, can be found in [HHWT95].

3.1 System-description language

The user describes a system as the composition of a collection of components. Each component is given as a direct textual representation of a linear hybrid automaton.

Type checking Variable type declarations allow more readable descriptions and enable simple static checking by the parser. Variables may be of the following types: discrete, clock, stopwatch, parameter, or analog. Variables of type discrete, clock, and parameter are said to be *fixed-rate* variables. Their rates are fixed by their type, and need not be given by the user.

Urgent transitions The introduction of urgent transitions serves two purposes. First, it often allows far simpler descriptions of a system. Urgent transitions could have been modeled using a special additional clock x_{urg}. Every urgent location v would then have an invariant $x_{urg} = 0$, and the clock x_{urg} would be reset to 0 on entering v. By contrast, our input language clearly labels urgent transitions as such, syntactically distinguishing them from the other transitions. This makes the modeling of parallel components easier. Suppose that a process is waiting in a location with an urgent transition waiting to synchronize with another process on a given label. In general, we cannot specify this behavior in a clean modular way using the clock x_{urg} from above. This is because we may need to identify all locations in the product automaton where the urgent transitions may synchronize. The correct timing information for the clock x_{urg} may not be achievable by adding invariants and jump conditions to the modular components. Our approach avoids this problem, and enables us to simply model each component by individually labeling urgent transitions. The only restriction is that urgent transitions must be invariant-enabled.

Second, our method admits more efficient implementation. Urgent transitions are implemented efficiently by using a boolean flag to distinguish each urgent location: there is no need to compute time-step successors for urgent locations. More importantly, it also avoids the need for the extra clock x_{urg}, thereby lowering the dimension of the continuous state space. This saving can be significant, because computation on polyhedra is exponential in the dimension, *i.e.* the number of variables.

3.2 Analysis-command language

HyTech provides an iterative programming language that enables the user to write her own verification programs using while loops, conditional statements, and primitive operations on regions. The region primitives supported by HyTech include boolean operations, *pre*, *post*, existential quantification on variables and locations (see below), and abstract interpretation operators such as convex hull. These operations allow the model checking of branching temporal-logic requirements [HNSY94, AHH93], and the abstract interpretation of hybrid automata [HRP94, HH95c]. The analysis-command language defines convenient macros for reachability analysis, and for the verification and parametric analysis of safety and timing requirements. There is also a routine for outputting trajectories (sequences of transition and time steps) between regions, which is useful for diagnostic error-trace generation (see below).

Parametric analysis A major strength of HyTech is its ability to perform exact parametric analysis [HWT95]. Usually we are only interested in the values of the parameters that cause a system to fail; the specific values of other variables when failure occurs is often irrelevant. However, this information is not easily inferred from a complete listing of the reachable states that are violating, because $post^*(I) \cap U$ includes relationships between the parameters and other variables. HyTech provides commands to existentially quantify "irrelevant" information. First, the user may specify any subset of the variables in X to be existentially quantified. Second, the user may request location information be ignored. For example, given a region W, HyTech can compute a linear predicate that defines the valuations s for which there exists a location v such that $(v, s) \in W$.

Error-trace generation To enhance HyTech's usefulness as a debugging tool, we added a facility to produce diagnostic information for systems failing to meet their safety and timing requirements. If a violation state in U is reachable from an initial state in I, then a trajectory from I to U provides an error trace. HyTech produces error traces that are minimal in the number of transition steps required to reach a violation state.

3.3 Implementation

All linear regions are stored using polyhedra to represent convex zones. Symbolic analysis of a linear hybrid automaton requires boolean operations on linear regions and the computation of successor and predecessor regions. The new HyTech uses an efficient C++ library for manipulating polyhedra [Hal93, HRP94]. The library supports boolean operations. We now show how to compute successor and predecessor regions. The polyhedral library offers two internal representations of polyhedra. One is a set of linear constraints defining a polyhedron in the natural way. The other is a *frame* that consists of (1) a nonempty finite set $\mathcal{P} = \{s_i\}_{1\leq i\leq p}$ of points in $\mathbb{R}^n$, (2) a finite set $\mathcal{R} = \{\rho_j\}_{1\leq j\leq r}$ of rays in $\mathbb{R}^n$ with the origin as source, and (3) a finite set of lines $\mathcal{L} = \{\gamma_k\}_{1\leq k\leq l}$ in $\mathbb{R}^n$ passing through the origin. The points, rays, and lines are viewed as generators defining the set of all states that can be written as a linear combination $\sum_{1\leq i\leq p} \lambda_i s_i + \sum_{1\leq j\leq r} \mu_j \rho_j + \sum_{1\leq k\leq l} \nu_k \gamma_k$, for real-valued constants λ_i, μ_j, and ν_k, where for all i, $\lambda_i \geq 0$, $\sum_{1\leq i\leq p} \lambda_i = 1$, and for all j, $\mu_j \geq 0$. Different operations on convex zones require different representations of polyhedra, so we switch between the two as needed. The library supplies routines for adding rays, points and lines to a frame representation of a polyhedron, adding constraints to a constraint representation, converting between the representations, and computing intersection.

The computation of successor regions requires the computation of time-step successors and transition-step successors.

Time-step successors Time-step successors are computed using the frame representation by

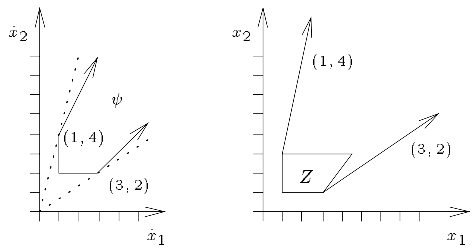

Figure 4: Computation of time-step successors

adding rays that delineate the "shadow" created by the continuous evolution of the variables. Suppose that a location has rate condition ψ, and suppose that there is a set $\mathcal{R}' = \{\rho'_j\}_{1 \leq j \leq m}$ of rays such that $\{\sum_{1 \leq j \leq m} \lambda_j \rho'_j \mid \lambda_j \geq 0\} = \{k \cdot u \mid \psi[\dot{X} := u]$ is true and $k \geq 0\}$. It can be shown that for every bounded rate condition such a set of rays exists. Computing the time-step successors involves adding these rays to the frame representation of a convex zone. For example, in a system with two variables x_1 and x_2, given the rate condition $\dot{x}_1 \geq 1 \wedge \dot{x}_2 \geq 2 \wedge \dot{x}_2 \geq \dot{x}_1 - 1 \wedge \dot{x}_2 \leq 2\dot{x}_1 + 2$, we add rays in the directions $(1,4)$ and $(3,2)$ when computing the time-step successors of the convex zone Z, as shown in Figure 4. The polyhedron obtained after adding rays is then intersected with the polyhedron for the location's invariant.

In order to compute the time-step predecessors for the *pre* operation, we use the inverse rays instead, *e.g.* $(-1,-4)$ and $(-3,-2)$ for the above example.

Transition-step successors Consider the transition $e = (v, v')$. To compute the successor states under $\rightarrow_e$ of a region $W = (v, Z)$ for a convex zone Z, we start with the constraint representation of Z. We temporarily augment the dimension of the state space with an added variable x'_i for each $x_i \in upd(e)$. We add the constraints for the jump condition $jump(e)$. For every $x_i \in upd(e)$, x_i is existentially quantified, and x'_i is "renamed" x_i. Last, we intersect with the invariant of v'. Existential quantification of the variable x is achieved by converting to the frame representation and adding the line whose direction has entry 1 for the component corresponding to x, and 0 for all other entries. Renaming consists of reconverting to a constraint matrix, copying the coefficients for the primed variables to their unprimed counterparts, and then decreasing the dimension of the constraint matrix back to n by disposing of entries for the primed variables.

To compute the set of predecessor states for each transition, we perform an analogous sequence of operations.

3.4 Performance

We have tested our verifier on a number of examples. Two of these — the generic railroad crossing and the active structure controller — are new examples of automatic parametric analysis, and are described in more detail in Section 4 below. In addition, we first analyzed the Philips audio control protocol [BPV94] in [HWT95], and provide comparative performance data for the new generation HyTech. The protocol communicates bit sequences using the timing-based Manchester encoding. The sender and receiver processes operate with unsynchronized clocks whose rates are subject to bounded drift. We verify that the protocol is correct for Philips' given clock drift of 1/20. We also synthesize the maximal possible clock drift required for correctness. Our model uses 320

	HyTech	prototype	factor faster
Audio control			
verifying correctness	19 sec	5.0 hrs	950
verifying timing	25 sec	5.0 hrs	723
parametric analysis	253 sec	27.9 hrs	396
Mutual exclusion			
2 processes – param.	2.6 sec	128 sec	49
3 processes – param.	26 sec	26.4 min	61
4 processes – param.	6.1 min	10.0 hrs	98
5 processes – param.	49.8 min	$**$	
Train-gate control			
verification	0.96	65	67
parametric analysis	1.4	110	78
GRC			
verification	9	2.4 hrs $*$	961
parametric analysis	34	1.3 hrs $*$	134
Active structure			
verification	153	N/A	N/A
parametric analysis	384	N/A	N/A

$*$ model uses deterministic assignment on exiting I
$**$ execution abandoned after 48 hrs

Figure 5: Comparative performance

locations and up to 7 variables. Second, we synthesized timing parameters for the correctness of Fischer's mutual exclusion protocol [AL92] with perfect clocks. We provide data for analyzing various numbers of concurrent processes contending for a resource. Third, we synthesize a critical upper bound on the controller's response time for the simple train-gate crossing of [AHH93].

Figure 5 compares HyTech's performace with that of the second generation verifier [HH95a]. All analysis was done on a Sun 670MP, and unless otherwise stated all times are in seconds. Data for the prototype over the active structure control examples are not available, since the prototype's input language does not allow a modular specification of the system's urgent transitions. The improvement in computational efficiency over the prototype is substantial. The reduced computational times are most dramatic in the audio control example, which involves a large product construction. This is explained by the fact that the prototype's code for forming products is written entirely in Mathematica, whereas its reachability-analysis code is a mixture of Mathematica and C++ routines.

4 Case Studies

We demonstrate HyTech's symbolic analysis techniques and diagnostic capabilities, by presenting new parametric results for the benchmark generic railroad crossing problem of [HJL93] and the active control structures of [ECB94].

4.1 Generic railroad crossing

We provide the first automatic synthesis of critical timing constraints for this system involving trains entering an intersection from multiple tracks. The GRC problem is stated in [HJL93] as follows.

The system to be developed operates a gate at a railroad crossing. The railroad crossing I lies in a region of interest R, i.e., $I \subset R$. A set of trains travel through R on multiple tracks in both directions. A sensor system determines when each train enters and exits region R. To describe the system formally, we define a gate function $g(t) \in [0, 90]$, where $g(t) = 0$ means the gate is closed and $g(t) = 90$ means the gate is open. We also define a set $\{\lambda_i\}$ of *occupancy intervals*, where each occupancy interval is a time interval during which one or more trains are in I. The ith occupancy interval is represented as $\lambda_i = [\tau_i, \nu_i]$, where τ_i is the time of the ith entry of a train into the crossing when no other train is in the crossing and ν_i is the first time since τ_i that no train is in the crossing (i.e., the train that entered at τ_i has exited as have any trains that entered the crossing after τ_i). Given two constants ξ_1 and ξ_2, $\xi_1 > 0$, $\xi_2 > 0$, the problem is to develop a system to operate the crossing gate that satisfies the following two properties:

Safety:	$t \in \cup_i \lambda_i \Rightarrow g(t) = 0$	The gate is down during all occupancy intervals.
Utility:	$t \notin \cup_i [\tau_i - \xi_1, \nu_i + \xi_2] \Rightarrow g(t) = 90$	The gate is up as often as possible.

4.1.1 System description

The model of the system is the parallel composition of the component automata appearing in Figures 1–3. To limit state-space explosion, we restrict our attention to the case of two trains.

Trains. Each train automaton T_i uses a clock x_i that indicates how long it will be until it reaches a sensor placed ahead of the intersection. A train sends a signal to the controller on entering the region R, and at some time between ϵ_1 and ϵ_2 time units later, it enters the intersection. There is no restriction on when it must leave the intersection. The variable delay between entering R and arriving at the intersection is modeling by the guard on the transition to I. The invariant $x_i \leq \epsilon_2$ on location R ensures that the train enters the intersection no later than its maximum delay time. The nondeterministic assignment on exiting I indicates it is unknown when the train will reenter R.

Gate. The locations of the gate automaton correspond to whether the gate is stationary in the open or closed position, or in the process of being raised or lowered. The rate at which the gate moves is variable, ranging between 9 and 10 degrees per time unit.

Controller. The controller receives sensing signals that indicate when each train enters R or leaves I. Recall that its operation is governed by the parameter α, which corresponds to the latest possible moment it may delay before sending the gate a *lower* command. In order to guarantee that the gate be closed when any trains are in the intersection, the value chosen for α must ensure that a train approaching as fast as possible will not enter the intersection before the slowest lowering of the gate is complete. This is achieved by monitoring the times since the trains entered the region R, and keeping the gate open provided all trains are sufficiently far from entering the intersection.

In two of its locations (*open* and *closed*), the controller waits while time passes. In location *open*, the controller believes the gate is either open or in the process of opening. The *closed* location is similar. Control may remain in the *open* location provided both trains are sufficiently far from entering their intersections. This is specified by the invariant $x_1 \leq \alpha \wedge x_2 \leq \alpha$. If either train reaches a critical distance from the intersection, the lower command is immediately issued.

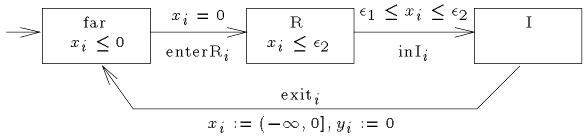

Figure 6: Revised train automata for verifying the utility property

From its *closed* location, the controller is essentially waiting for a train to leave the intersection, so it may issue a command to raise the gate. However, even if a train should leave the intersection, it may not be appropriate to raise the gate, since the other train may already be near, or even within, the intersection. If the other train is far enough away, the controller enters location *must_raise*. Then the urgent transition to location *open* is taken, corresponding to sending an immediate raise command to the gate.

4.1.2 System requirements

Both the safety and utility properties of [HJL93] can be expressed as reachability properties. The controlled system satisfies the safety property provided it never reaches a state in which a train is in location I, and the gate is not in location *closed*. By symmetry we need only consider the case concerning train 1. The violating states are expressed by the state predicate $loc[T_1] = I \wedge loc[gate] \neq closed$, where state predicates specifying the location of the component automaton A_i are written using the location counter $loc[A_i]$. The utility property states that the gate is open unless there is a train that was recently in the intersection, or is about to be in the intersection. To express the amount of time elapsed since T_i last exited the intersection, we need to introduce an extra clock y_i. The augmented automaton for the ith train appears in Figure 6. The utility property is violated precisely when it is possible to reach a state in which the gate is in location *open* and neither train is close to being in the intersection. The violating states are expressed by the state predicate:

$$\begin{gathered} loc[gate] \neq open \wedge loc[T_1] \neq I \wedge loc[T_2] \neq I \wedge \\ x_1 \le \epsilon_2 - \xi_1 \wedge y_1 \ge \xi_2 \wedge x_2 \le \epsilon_2 - \xi_1 \wedge y_2 \ge \xi_2 \end{gathered}$$

4.1.3 Verification

We verified that the system satisfies both the safety and utility properties when the parameters are fixed to appropriate values. We set ϵ_1 to 30 and ϵ_2 to 40. A train could enter I as soon as 30 time units after entering R, and the gate takes up to 10 time units to lower. Therefore the controller must send the gate a signal 10 time units before the train's clock reaches 30, *i.e.* within 20 time units of the train's entering the region R. In order to avoid a race condition, we set the parameter α in the controller to 19. With these parameters set, we also verify the utility property for $\xi_1 = 22$ and $\xi_2 = 11$. To see that ξ_1 must be greater than 21, observe that a slow train may enter the intersection 21 time units after the controller orders the gate to lower. Similarly, ξ_2 must be greater than 10, since a slow gate may take 10 time units to be opened after a train exits. HyTech takes 9 seconds to verify both these properties.

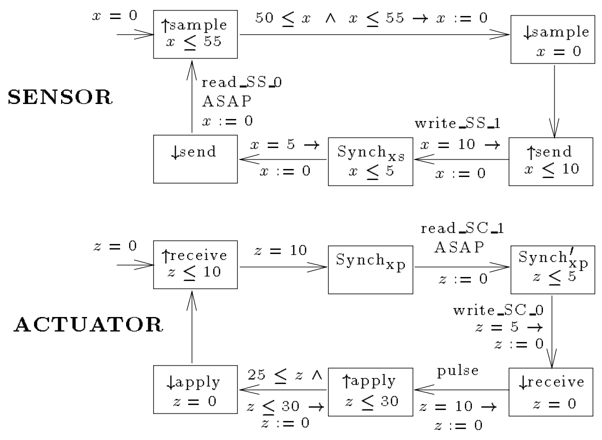

Figure 7: Sensor and actuator automata

4.1.4 Parametric analysis

We use HyTech to determine precisely which values of α will yield controllers satisfying the safety property. The cutoff point α for lowering the gate is treated as a parameter. Our tool takes 34 seconds to determine that the controller is correct whenever the parameter α has value strictly less than 20.

4.2 Active structure control

A formal description of an intelligent structural control system appears in [ECB94]. The control system uses a pulse control algorithm that performs three basic tasks: sampling the state of the structure, updating its model of the structure, and applying a pulse to the structure. The system consists of three main components — the sensor, the actuator, and the controller. The role of the sensor is to sample important system data such as velocities and displacements and to communicate this information to the controller. Sampling may take from 50 to 55 time units (given as tenths of a millisecond). After sampling the data, the sensor sends it to the controller, and waits for the controller to reactivate its sampling. The actuator operates in a similar fashion to the sensor, except that instead of sensing data it applies pulses to the structure being controlled. Pulses last from 25 to 30 time units. The controller's role is to coordinate the sensor's sampling and the actuator's pulse applications, as well as updating its own model of the structure, and calculating the magnitude of pulses. It repeatedly activates the sensor. When it receives data back from the sensor, it decides whether to update its model of the structure and reactivate the sensor, or to first calculate the appropriate pulse and then signal the actuator to proceed with a pulse application.

It is verified in [ECB94] that the control system satisfies lower and upper bounds on the time between consecutive pulse applications. The system is modeled in Modechart [JS88], a graphical language for hierarchical state machines subject to timed enabling conditions. A translation technique from Modechart into Temporal CCS is provided. The Temporal CCS description is then verified using the Concurrency Workbench [CPS93].

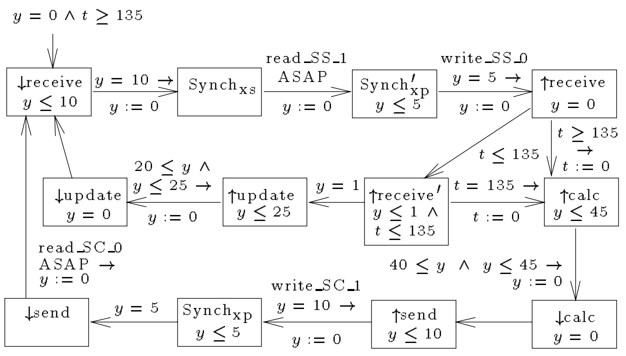
$y = 0 \wedge t \geq 135$
↓receive $y \leq 10$
$y = 10 \rightarrow$ $y := 0$
Synch$_{xs}$
read_SS_1 ASAP $y := 0$
Synch$'_{xp}$ $y \leq 5$
write_SS_0 $y = 5 \rightarrow$ $y := 0$
↑receive $y = 0$
$t \geq 135 \rightarrow t := 0$
$t \leq 135$
↓update $y = 0$
$20 \leq y \wedge y \leq 25 \rightarrow$ $y := 0$
↑update $y \leq 25$
$y = 1$
↑receive$'$ $y \leq 1 \wedge t \leq 135$
$t = 135 \rightarrow$ $t := 0$
↑calc $y \leq 45$
read_SC_0 ASAP $\rightarrow$ $y := 0$
$40 \leq y \wedge y \leq 45 \rightarrow$ $y := 0$
↓send
$y = 5$
Synch$_{xp}$ $y \leq 5$
write_SC_1 $y = 10 \rightarrow$ $y := 0$
↑send $y \leq 10$
↓calc $y = 0$

Figure 8: Controller automaton

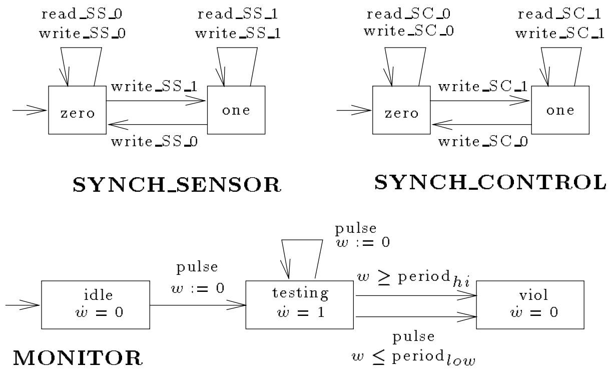

Figure 9: Synchronization variables and monitor

4.2.1 System description

The translation from the Modechart specification to linear hybrid automata is relatively straightforward. The system's sensor, actuator, and controller, as well as the two auxiliary binary variables *synch_sensor* and *synch_control* are modeled as the linear hybrid automata in Figures 7–9. The variables x, y, z, and t are all clocks. In the sensor automaton, the upper bound of 55 on the sampling time is enforced by the invariant on location ↑sample. Communication is coordinated through the control variable *synch_sensor*. After completing its sampling, the sensor takes 10 time units to prepare the data for transmission (in location ↑send), then sends the data in 5 time units after first setting *synch_sensor* to 1. It then waits for the controller to reset the variable to 0 as a signal to reactivate sampling. The actuator automaton is similar to the sensor. Coordination with the controller is achieved through the variable *synch_control*.

The controller follows two basic loops; one for sampling data (the upper loop in Figure 8) and one for prompting the actuator. Each time a cycle is completed and control returns to location ↑receive, the controller decides whether to execute a sampling cycle or an actuating cycle. If, after checking continuously for 1 time unit, insufficient time (no more than 135 time units) has passed since taking the last actuating cycle, it follows the sampling branch. Otherwise, it reactivates the actuator. To do so, it first calculates how strong and how long the pulse should be before setting the variable *synch_control* to 1 to alert the actuator. The clock t is used to measure the time elapsed since control last entered location $\uparrow$calc at the beginning of the actuating cycle.

Each synchronization variable is modeled using a separate location for each value it may take. Reading the value of the variable is dependent on the current location, and writing causes the variable to move to the location for the new value. Note that we need separate automata to model the variables *synch_control* and *synch_sensor*, rather than declaring them as explicit variables, because there are urgent transitions dependent on their values.

4.2.2 System requirements

According to [ECB94], the system should meet certain periodic constraints on the times between the starts of successive pulses. The delays between pulses are required to lie between 37 and 145 time units.

4.2.3 Parametric analysis

Rather than simply verifying these bounds are met, we instead synthesize the exact bounds on the system's period. Following [ECB94], we add a monitor automaton to the system (see Figure 9). It "watches" the executions of the control system and enters a special violation location whenever the system violates its safety specification. In our case, we also introduce two parameters, period_{low} and period_{hi}. A violation consists of a pulse event followed by another pulse event before period_{low} time units, or a pulse event followed by more than period_{hi} time units without a pulse event. HyTech shows that the period can only take values in the interval $[135, 145]$, where the bounds are tight.

4.2.4 Error-trace generation

In our analysis of the system, HyTech revealed a typographical error in the presentation of the Modechart specification in [ECB94]. The typographical error admits a subtle, but critical, race condition in the controller that enables delays of up to 210 time units between pulses, thereby violating the system's periodic requirements. We use this example to demonstrate HyTech's diagnostic capabilities.

By literally translating the Modechart in [ECB94] for the controller into a hybrid automaton, the transition (see Figure 8) out of location $\uparrow$receive with guard $t \leq 135$ would have destination location $\uparrow$update, and location $\uparrow$receive$'$ and all associated transitions would not appear. For the resulting system, HyTech synthesizes a lower (upper) bound of 135 (210) time units on the delay between successive pulse applications. The tool also generates a non-trivial minimal-length error trace with 49 transition steps in 123 seconds of CPU time. In the execution trace, the controller follows its actuating cycle the first time it reaches location $\uparrow$receive. After the controller sets the variable *synch_control* to 1, the actuator applies a pulse 15 time units later. The controller then progresses to location $\uparrow$receive with the clock t having value 70. The ensuing sampling cycle takes place in exactly 65 time units; the sensor can immediately begin sampling in location $\uparrow$sample (a delay of 50 time units), then set *synch_sensor* to the value 1 (a 10 time unit delay), followed by a delay of 5 time units in the controller before resetting *synch_sensor* to 0. Thus the controller enters location $\uparrow$receive with t having value 135. The controller may then follow another sampling cycle, which delays the application of the next pulse longer than the tolerable upper bound of 145 time units.

References

[ACD$^+$92] R. Alur, C. Courcoubetis, D. Dill, N. Halbwachs, and H. Wong-Toi. An implementation of three algorithms for timing verification based on automata emptiness. In *Proc. of 13th IEEE Real-time Systems Symposium*, pp. 157–166, 1992.

[ACH$^+$95] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[ACHH93] R. Alur, C. Courcoubetis, T.A. Henzinger, and P.-H. Ho. Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, LNCS 736, pp. 209–229. Springer, 1993.

[AD94] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183-235, 1994.

[AHH93] R. Alur, T.A. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. In *Proc. of 14th IEEE Real-time Systems Symposium*, pp. 2–11, 1993.

[AHV93] R. Alur, T.A. Henzinger, and M.Y. Vardi Parametric real-time reasoning. *Proc. of 25th ACM Symposium on Theory of Computing*, pp. 592–601, 1995.

[AL92] M. Abadi and L. Lamport. An old-fashioned recipe for real time. In *Real Time: Theory in Practice*, LNCS 600, pp. 1–27. Springer, 1992.

[BPV94] D. Bosscher, I. Polak, and F. Vaandrager. Verification of an audio-control protocol. In *FTRTFT 94: Formal Techniques in Real-time and Fault-tolerant Systems*, LNCS 863, pp. 170–192. Springer, 1994.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. of 5th ACM Symposium on Principles of Programming Languages*, pp. 84–97, 1978.

[Che68] N.V. Chernikova. Algorithm for discovering the set of all solutions of a linear programming problem. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 8(6):282–293, 1968.

[CPS93] R.J. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: a semantics-based tool for the verification of finite-state systems. *ACM Trans. on Programming Languages and Systems*, 15(1):36–72, 1993.

[DWT95] D.L. Dill and H. Wong-Toi. Verification of real-time systems by successive over and under approximation. In *CAV 95: Computer-aided Verification*, LNCS 939, pp. 409–422. Springer, 1995.

[DY95] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with Kronos. This volume.

[ECB94] W.M. Elseaidy, R. Cleaveland, and J.W. Baugh Jr. Verifying an intelligent structural control system: a case study. In *Proc. of 15th IEEE Real-Time Systems Symposium*, pp. 271–275, 1994.

[GL90] R. Gerber and I. Lee. A proof system for communicating shared resources. In *Proc. of 11th IEEE Real-time Systems Symposium*, pp. 288–299, 1990.

[Hal93] N. Halbwachs. Delay analysis in synchronous programs. In *CAV 93: Computer-aided Verification*, LNCS 697, pp. 333–346. Springer, 1993.

[HH95a] T.A. Henzinger and P.-H. Ho. HyTech: The Cornell Hybrid Technology Tool. In *Proc. of 1994 Workshop on Hybrid Systems and Autonomous Control.* Springer, 1995.

[HH95b] T.A. Henzinger and P.-H. Ho. Algorithmic analysis of nonlinear hybrid systems. In *CAV 95: Computer-aided Verification*, LNCS 939, pp. 225–238. Springer, 1995.

[HH95c] T.A. Henzinger and P.-H. Ho. A note on abstract-interpretation strategies for hybrid automata. In *Proc. of 1994 Workshop on Hybrid Systems and Autonomous Control.* Springer, 1995.

[HHWT95] T. A. 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.* Springer, 1995.

[HJL93] C.L. Heitmeyer, R.D. Jeffords, and B.G. Labaw. A benchmark for comparing different approaches for specifying and verifying real-time systems. In *Proc. of 10th IEEE Int. Workshop on Real-time Operating Systems and Software*, 1993.

[HKPV95] T.A. Henzinger, P.W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? In *Proc. of 27th ACM Symposium on Theory of Computing*, pp. 373–382, 1995.

[HL94] C. Heitmeyer and N. Lynch. The generalized railroad crossing: a case study in formal verification of real-time systems. In *Proc. of 15th IEEE Real-time Systems Symposium*, pp. 120–131, 1994.

[HNSY94] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[HRP94] N. Halbwachs, P. Raymond, and Y.-E. Proy. Verification of linear hybrid systems by means of convex approximation. In *SAS 94: Static Analysis Symposium*, LNCS 864, pp. 223–237. Springer, 1994.

[HWT95] P.-H. Ho and H. Wong-Toi. Automated analysis of an audio control protocol. In *CAV 95: Computer-aided Verification*, LNCS 939, pp. 381–394. Springer, 1995.

[JS88] F. Jahanian and D.A. Stuart. A method for verifying properties of modechart specifications. In *Proc. of 9th IEEE Real-time Systems Symposium*, pp. 12–21, 1988.

[LPY95] K.G. Larsen, P. Pettersson, and W. Yi. Compositional and symbolic model-checking of real-time systems. This volume.

[LS85] N. Leveson and J. Stolzy. Analyzing safety and fault tolerance using timed petri nets. In *Proc. of Int. Joint Conference on Theory and Practice of Software Development*, LNCS 186, pp. 339–355. Springer, 1985.

[Sha93] N. Shankar. Verification of real-time systems using PVS. In *CAV 93: Computer-aided Verification*, LNCS 697, pp. 280–291. Springer, 1993.

[WM93] F. Wang and A.K. Mok. A verifier for distributed real-time systems with bounded integer variables. In *Proc. of 8th IEEE Conference on Computer Assurance*, pp. 135–151, 1993.