A User Guide to HyTech[*][†]

Thomas A. Henzinger
EECS Department
University of California
Berkeley, CA
tah@eecs.berkeley.edu

Pei-Hsin Ho
Intel Development Labs
Intel Corporation
Hillsboro, OR
pho@ichips.intel.com

Howard Wong-Toi
Cadence Berkeley Labs
Cadence Design Systems
Berkeley, CA
howard@cadence.com

Abstract

HyTech is a tool for the automated analysis of embedded systems. This document, designed for the first-time user of HyTech, guides the reader through the underlying system model, and through the input language for describing and analyzing systems. The guide gives installation instructions, several examples of usage, some hints for gaining maximal computational efficiency from the tool, and the complete grammar for the input language.

This guide describes version 1.04 of HyTech. The latest update occurred in October 1996[1]. HyTech is available through the World-Wide Web at http://www.eecs.berkeley.edu/˜tah/HyTech.

1 Introduction

The control of physical systems with embedded hardware and software is a growing application area for computerized systems. Since many embedded controllers occur in safety-critical situations, it is important to have reliable design methodologies that ensure that the controllers operate correctly. HyTech aids in the design of embedded systems by not only checking systems requirements, but also performing parametric analysis. Given a parametric system description, HyTech returns the exact conditions on the parameters for which the system satisfies its safety and timing requirements.

For completeness, we begin with a brief presentation of the underlying theoretical framework of *linear hybrid automata* [ACHH93, ACH$^+$95], which we use to describe system specifications and requirement specifications. These automata model the continuous activities of analog variables (such as temperature, time, and distance), as well as discrete events (such as interrupts and output signals). Communication is modeled through event synchronization and shared variables. HyTech's input consists of two parts: a system description and analysis commands. The system-description language allows us to represent linear hybrid automata textually. The tool forms the parallel composition of a collection of automata, each describing a modular component of an embedded system. The analysis-command language allows us to write simple iterative programs for performing tasks such as reachability analysis and error-trace generation.

We illustrate the use of the tool on several examples taken from the literature, and provide hints for a verification engineer to gain the maximal possible efficiency from HyTech.

Outline Section 2 reviews linear hybrid automata, their semantics, parallel composition, and associated analysis techniques. A brief history of HyTech appears in Section 3. Sections 4 and 5 describe the HyTech

[*] This research was supported in part by the Office of Naval Research Young Investigator award N00014-95-1-0520, by the National Science Foundation CAREER award CCR-9501708, by the National Science Foundation grants CCR-9200794 and CCR-9504469, by the Air Force Office of Scientific Research contract F49620-93-1-0056, by the Advanced Research Projects Agency grant NAG2-892.

[†] An abbreviated version of this paper appeared in the Proceedings of the *First Workshop on Tools and Algorithms for the Construction and Analysis of Systems* (TACAS), Lecture Notes in Computer Science 1019, Springer-Verlag, 1995, pp. 41–71.

[1] The only changes from version 1.02 of February 1996 are bug fixes, porting to Linux, DEC Ultrix, Digital Unix, and HP-UX in May 1996, and replacement of the closure difference operator `cldiff` with the exact difference operator `diff` in October 1996.

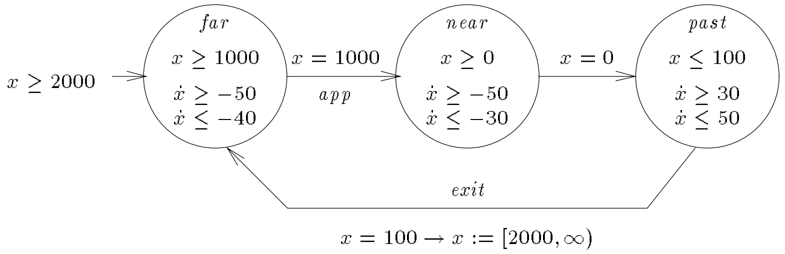

Figure 1: Train automaton

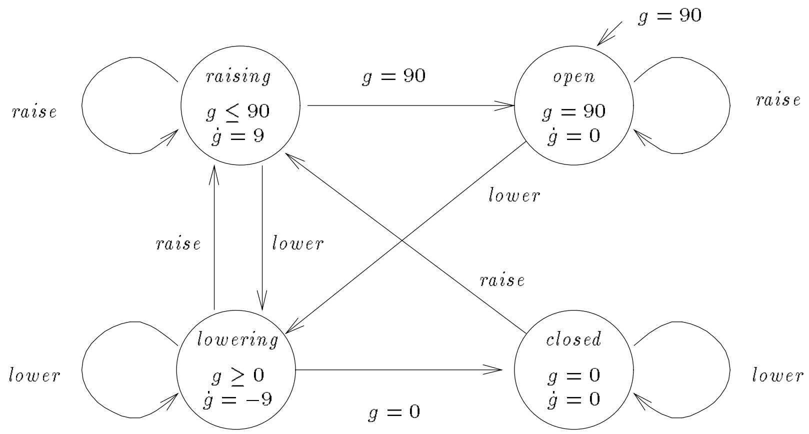

Figure 2: Gate automaton

input language, first the system-description part, and then the analysis-command part. Section 6 illustrates the use of the tool on several examples. Section 7 is a short guide to designing specification requirements using HyTech's command language. Section 8 provides information on installing and running HyTech. Section 9 contains hints for the efficient use of HyTech. Appendix A contains complete input and output files. The grammar of HyTech's input language appears in Appendix B.

2 Linear Hybrid Automata

We model systems as the parallel composition of a collection of linear hybrid automata [ACHH93, ACH$^+$95]. Informally, a linear hybrid automaton consists of a finite set X of real-valued variables and a labeled multigraph. The vertices represent control modes, each with its own constraints on the slopes of variables in X. The edges represent discrete events and are labeled with guarded assignments to X. The state of the automaton changes either through the instantaneous action associated with an event or, while time elapses, through the continuous activity associated with a control mode. We also explicitly model *urgent* events, which must take place as soon as they are enabled (unless another instantaneous action disables them).

We use the linear hybrid automata that model a simple railroad crossing [LS85, AHH93] as a running example. The system consists of three components: a train, a gate, and a controller. The train is initially some distance — at least 2000 feet — away from the track intersection with the gate fully raised. As the train approaches, it triggers a sensor — 1000 feet ahead of the intersection — signaling its upcoming entry to the

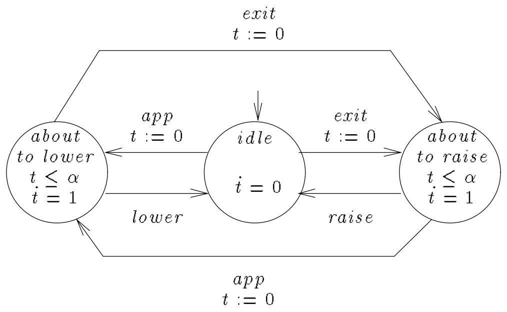

Figure 3: Controller automaton

controller. The controller then sends a lower command to the gate, after a delay of up to α seconds. When the gate receives a lower command, it lowers at a rate of 9 degrees per second. After the train has exited the intersection and is 100 feet away, it sends an exit signal to the controller. The controller then commands the gate to be raised. The role of the controller is to ensure that the gate is always closed whenever the train is in the intersection, and that the gate is not closed unnecessarily long. The linear hybrid automata for the train, the gate, and the controller appear in Figures 1, 2 and 3.

2.1 Definition

We give an informal description of linear hybrid automata, and refer the reader to [AHH93, HHWT95a] for detailed definitions. A *linear hybrid automaton* consists of the following components.

Variables The automaton uses a finite ordered set $X = \{x_1, x_2, \ldots, x_n\}$ of real-valued *variables* to model continuous activities. For example, the position of the train is determined by the value of the variable x, which represents the distance of the train from the intersection. The variable g models the angle of the gate. When $g = 90$, the gate is completely open; when $g = 0$, it is completely closed.

A *valuation* is a point $(a_1, a_2, \ldots, a_n)$ in the n-dimensional real space $\mathbb{R}^n$, or equivalently, a function that maps each variable x_i to its value a_i. A *linear expression* over a set X of variables is a linear combination of variables in X with rational coefficients. A *linear inequality* is an inequality between linear expressions. A *convex predicate* is a finite conjunction of linear inequalities, *e.g.* $x_1 \geq 3 \wedge 3x_2 \leq x_3 + 5/2$. A *predicate* is a finite disjunction of convex predicates, defining a set of valuations.

Locations Control modes are modeled using a finite set of vertices called *locations*. For example, the gate automaton has the locations *open*, *raising*, *lowering*, and *closed*. A *state* (v, s) of the automaton A consists of a location v and a valuation s. We use the term *region* to refer to a set of states. The valuations *associated with* a location v within a region W are the valuations s such that $(v, s) \in W$.

Initial condition There is a designated initial location and an initial predicate ϕ_0 over X defining the set of initial values of the variables. For example, the gate is initially in location *open* with the value of g equal to 90. In the graphical representation, a small incoming arrow identifies the initial location, and is labeled with the predicate ϕ_0.

Invariant conditions Each location v is labeled with a convex predicate $inv(v)$ over X, the *invariant* of v. The automaton control may reside in location v only while its invariant is true, so the invariants can be used to enforce progress in the automaton. For example, in the gate automaton, $inv(open) = (g = 90)$, $inv(lowering) = (g \geq 0)$, $inv(raising) = (g \leq 90)$, and $inv(closed) = (g = 0)$. The invariant at location *lowering* implies that the gate can only be lowered until it is fully closed, at which point control moves out to location *closed*. In the graphical representation, the invariant *true* is omitted.

We are primarily interested in states (v, s) where the valuation s satisfies the location's invariant $inv(v)$. Such states are called *admissible.*

Transitions Discrete events are modeled using edges between locations, which are called *transitions.* For example, the train automaton has three transitions; one from location *far* to location *near* for entering the region immediately surrounding the intersection, one from *near* to *past* for going through the intersection, and one from *past* to *far* for exiting the region around the intersection.

Each transition is labeled with an *update set* and a *jump condition.* The update set Y is a subset of X. The jump condition is a convex predicate ψ over $X \cup Y'$, where $Y = \{y_1, \ldots, y_k\}$, 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 Y are updated by the transition. The transition t may take place between admissible states (v, s) and (v', s'), denoted $(v, s) \xrightarrow{t} (v', s')$, if and only if (1) $\psi[X, Y' := s, s'[Y]]$[2] is true, and (2) for all variables $x_i \in X \setminus Y$, $s_i = s'_i$, *i.e.* only variables in the update set have their values changed. The transition t is enabled at state (v, s) if and only if there is an admissible state (v', s') such that $(v, s) \xrightarrow{t} (v', s')$. Nonconvex jump conditions can be modeled by splitting transitions.

When writing update sets and jump conditions, we often use nondeterministic guarded assignments to intervals (bounded or unbounded). For example, we write $\phi \rightarrow y_i := [l, u)$ for the update set $\{y_i\}$ and the jump condition $\phi \wedge l \leq y'_i < u$, where l and u are linear expressions over X. This jump condition is enabled in the valuation s if the guard ϕ is satisfied. 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. Transitions may optionally be assigned the *urgency flag* Asap. Transitions so labeled are called *urgent.*

In the train automaton, the transition between locations *past* and *far* is labeled with the guarded command $x = 100 \rightarrow x := [2000, \infty)$. In the graphical representation, we omit the guard *true*.

We define the binary *transition-step* relation, $\xrightarrow{\sigma}$, over admissible states such that $(v, s) \xrightarrow{\sigma} (v', s')$ iff the state (v', s') can be reached from the state (v, s) by taking a transition. We assume that for every urgent transition u, if $(v, s) \xrightarrow{u} (v', s')$, then for all valuations s_0 satisfying $inv(v)$ there exists a valuation s'_0 such that $(v, s_0) \xrightarrow{u} (v', s'_0)$. A location v is urgent if there exists a valuation s and an urgent transition u, such that u is enabled at (v, s). No time is allowed to pass in such a location.

Each transition is optionally given a synchronization label. The synchronization labels are used to define the parallel composition of hybrid automata. For example, in the gate automaton, the transition from *open* to *lowering* has the synchronization label *lower*, and this synchronizes (*i.e.* must be taken simultaneously) with the transition labeled *lower* in the controller automaton.

Rate conditions We denote the rate of change of the variable $x \in X$ by $\dot{x}$, and we let $\dot{X}$ be the set $\{\dot{x}_1, \dot{x}_2, \ldots, \dot{x}_n\}$. Each control location v is labeled with a convex predicate $act(v)$ over $\dot{X}$, called the *rate condition* of v. For a given location, the rate condition restricts the rates of change of the variables. In the gate automaton, the rate condition for locations *open* and *closed* is $\dot{g} = 0$, for location *raising*, it is $\dot{g} = 9$, and for *lowering*, it is $\dot{g} = -9$. There is a technical restriction on the rate conditions allowed. All predicates that define closed and bounded sets over $\dot{X}$ are permitted, and all examples in this guide meet this condition[3].

We define the *time-step* relation, $\xrightarrow{\tau}$, such that $(v, s) \xrightarrow{\tau} (v', s')$ iff $v = v'$, and there exists a real $\delta \geq 0$ such that $\delta > 0$ implies v is not urgent, and there is a function $f : [0, \delta] \rightarrow \mathbb{R}^n$ such that (1) $f(0) = s$, (2) $f(\delta) = s'$, (3) for all $t \in [0, \delta]$, $f(t)$ satisfies $inv(v)$, and (4) for all time $t \in (0, \delta)$ $(df_1(t)/dt, df_2(t)/dt, \ldots, df_n(t)/dt)$ satisfies $act(v)$, where $f_i(t)$ denotes the value of variable x_i in the valuation $f(t)$.

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

[3] The precise condition for the rate condition ψ to be allowed is that ψ defines a closed set, and the set of vectors $\{\dot{y} \mid \dot{y}$ does not satisfy ψ, and $\exists k \in \mathbb{R}$ such that $0 \leq k \leq 1$ and $\dot{x}$ satisfying ψ such that $\dot{y} = k\dot{x}\}$ is bounded. In theory, the condition we require is not essential: however, it guarantees that the computation of time-step successors is efficient.

2.2 Parallel composition

A hybrid system typically consists of several components which operate concurrently and communicate with each other. Each component is described as a separate linear hybrid automaton. The component automata coordinate through shared variables, and synchronization labels on the transitions are used to model message-type coordination. The linear hybrid automaton for the entire system is then obtained from the component automata using a product construction.

The control locations of the parallel composition of two automata A_1 and A_2 are pairs of locations, the first from A_1 and the second from A_2. The location (v_1, v_2) has the conjunction of v_1 and v_2's invariants as its invariant, and the conjunction of their rate conditions as its rate condition. A location is initial iff its components are initial in their respective automata. The initial convex predicate is the conjunction of the components' initial convex predicates. Transitions from the components are interleaved, unless they share the same synchronization label, in which case they are synchronized and executed simultaneously, if at all. In the train-gate controller example, the system is composed of the train, gate, and controller automata of Figures 1, 2 and 3. The controller communicates with the train by synchronizing on approach and exit events. It issues commands to the gate on the synchronized events *raise* and *lower*. The train's transition from location *near* to *far* is unlabeled, so it does not synchronize with any of the other components. In particular, this means the controller does not know the precise time at which the train enters the intersection.

We require a technical condition that the composition be well-formed: whenever two components synchronize on a label, if one transition is urgent then the other must either be urgent, or have a jump condition expressible as a guarded command with its guard being either the predicate *true* or the predicate *false*.

2.3 Reachability and safety verification

At any time instant, the state of a hybrid automaton specifies a location and the values of all variables. If the automaton has the location set V and n variables, the state space is defined as $V \times \mathbb{R}^n$. We define the binary successor relation $\rightarrow_A$ over states as $\xrightarrow{\tau} \cup \xrightarrow{\sigma}$. For a region W, we define $post(W)$ to be the set of all successor states of W, *i.e.* all states reachable from a state in W via a single transition or time step. The region *forward reachable* from W is defined as the set of all states reachable from W after a finite number of steps, *i.e.* the infinite union $post^*(W) = \bigcup_{i \geq 0} post^i(W)$. Similarly, we define $pre(W)$ to be the set of all predecessor states of W, and we let the region *backward reachable* from W be the infinite union $pre^*(W) = \bigcup_{i \geq 0} pre^i(W)$.

In practice, many problems to be analyzed 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 violates a given safety requirement. Indeed all timed safety requirements [Hen92], including bounded-time response requirements, can be verified in this way. See Section 7. A state (v, s) is *initial* if v is the initial location, and s satisfies the initial predicate. A system with initial states I is correct with respect to violation states Y iff $post^*(I) \cap Y = \emptyset$, or equivalently iff $pre^*(Y) \cap I$ is empty.

HyTech computes the forward reachable region by finding the limit of the infinite sequence I, $post(I)$, $post^2(I)$, … of regions. Analogously, the backward reachable region is found by iterating *pre*. These iteration schemes are semidecision procedures: there is no guarantee of termination. Nevertheless, we find that in practice, HyTech's reachability procedures terminate on most examples we have attempted. In addition, it has been shown that for a large class of systems [HKPV95], a linear hybrid automaton can be automatically preprocessed into an equivalent automaton over which the iterations converge.

2.4 Parametric analysis

A major strength of HyTech is its ability to perform parametric analysis. Often a system is described using parameters, and the system designer is interested in knowing which values of the parameters are required for correctness. Since the system is incorrect for parameter values for which there exists a state in the region $post^*(I) \cap Y$, we may obtain necessary and sufficient conditions for system correctness by performing reachability analysis followed by existential quantification [CH78].

Our study of the train-gate controller demonstrates this technique. The controller decides when to issue *lower* commands to the gate based on the amount of time since the train last passed the sensor located

1000 feet ahead of the intersection. We consider the problem of determining exactly how long the controller can wait before issuing commands, while maintaining the requirement that the gate be closed whenever the train is within 10 feet of the intersection. The parameter α corresponds to the latest possible moment the controller can wait. We then use HyTech to determine that the composed system includes violations whenever α is greater than or equal to 49/5. Thus we conclude that the system is correct for values of the parameter strictly less than 49/5.

3 A Brief History of HyTech

3.1 Implementation

There have been three generations of HyTech. The very earliest prototype [AHH93] was written entirely in the symbolic computation tool Mathematica. Regions were represented as symbolic formulas. The evaluation of time-step successors used existential quantifications that are easily encoded in this language. While Mathematica offers powerful symbolic manipulation, and allows rapid development and experimentation with algorithms and heuristics, its operations over predicates turned out to be computationally inefficient. In particular, quantifier-elimination operations for computing time-step successors were expensive. HyTech [HH95b] was rewritten to avoid this bottleneck in Mathematica. The second version of the verifier used a Mathematica main program that called efficient C++ routines from Halbwachs' polyhedral manipulation library [Hal93, HRP94] for computing time-step successors. While this verifier achieved a total speed-up of roughly one order of magnitude, it required inefficient conversions between Mathematica expressions and C++ data structures. It still relied on Mathematica for computing transition-step successors by substitution.

The third generation HyTech described here[4] 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. In addition, the input automata now allow nondeterministic assignments to variables, simultaneous assignments, more general rate conditions, and urgent events.

3.2 A guide to HyTech-related papers

The following papers explain the theory behind linear hybrid automata in more detail, provide examples of their use, and discuss HyTech and related tools.

Theory of hybrid automata Hybrid automata are based on timed automata [AD94] and were introduced in [ACHH93]. A related model appeared in the same volume [NOSY93]. Analysis methods included reachability and state-space minimization. The specification language Integrator Computation Tree Logic (ICTL) and a model-checking algorithm were introduced in [AHH93]. Approximations and abstract interpretation strategies for the algorithmic analysis of hybrid automata are discussed in the papers [HRP94, OSY94, HH95c]. The paper [ACH$^+$95] provides an overview of the analysis techniques, including approximations. The analysis of non-linear automata by translations to linear automata is described in [HH95a, HWT95a]. Decidability results appear in [Cer92, ACH93, KPSY93, AD94, MV94, PV94, BER94a, BER94b, BR95, MPS95, Hen95, HHK95, HKPV95]. In particular, [HKPV95] shows that the reachability problem is decidable, and HyTech's analysis terminates, on the class of *rectangular automata*, where all convex predicates are of the form $a \leq x \leq b$ ($a \leq \dot{x} \leq b$).

HyTech The earliest version of HyTech is mentioned in [AHH93], and performs full model checking of ICTL formulas. The second generation of HyTech is discussed in [HH95b]. The thesis [Ho95] describes the first two generations of HyTech in more detail, as well as summarizing much of the theory of hybrid automata. The current version of HyTech is described in [HHWT95a]. A shorter version of this guide appears in [HHWT95b].

Case studies Numerous examples have been analyzed using linear hybrid automata. We mention only the first appearances of examples in the hybrid automata literature. A gas burner is studied in [ACHH93],

[4] This guide describes version 1.04, which contains some extensions to August 1995's version 1.00, together with some syntax changes in the input language.

```
define(raise_rate,9)
define(lower_rate,-9)

automaton gate
synclabs: raise, lower;
initially open & g=90;
loc raising: while g<=90 wait {dg=raise_rate}  -- gate is being raised
        -- gate is fully raised
        when g=90 goto open;
        -- selfloops for input enabledness
        when True sync raise goto raising;
        when True sync lower goto lowering;
loc open: while True wait {dg=0}               -- wait for command
        when True sync raise goto open;
        when True sync lower goto lowering;
loc lowering: while g>=0 wait {dg=lower_rate}  -- gate is being lowered
        -- gate is fully lowered
        when g=0 goto closed;
        when True sync lower goto lowering;
        when True sync raise goto raising;
loc closed: while True wait {dg=0}             -- wait for command
        when True sync raise goto raising;
        when True sync lower goto closed;
end -- gate
```

Figure 4: HyTech input for the gate automaton

together with a simple water monitor. The trajectories of a billiard ball, and the temperature of a reactor core are modeled in [NOSY93]. Fischer's timing-based mutual exclusion protocol is considered in [AHH93]. The paper [HH95b] includes a parametric analysis. A simple train-gate controller and a scheduler appear in [AHH93]. A manufacturing robot system and Corbett's distributed control system are also discussed in [HH95b]. The paper [HWT95b] describes the verification (see also [HH95b]) and error analysis of an audio control protocol. The benchmark generic railroad crossing example and an active structure controller are considered in [HHWT95a]. A nonlinear temperature controller appears in [HH95a], and a predator-prey system in [HWT95a].

Related Tools The analysis of linear hybrid automata supported by HyTech is based on symbolic region manipulation techniques first presented for real-time systems [HNSY94]. For the restricted case of real-time systems, these techniques have also been implemented in the tools Kronos [NSY92, DOY94, ACH$^+$95, DY95] and Uppaal [LPY95]. Polka [Hal93, HRP94] is a tool for analyzing hybrid systems that concentrates on abstract interpretation strategies.

4 Input Language: System Description

HyTech's input consists of a text file containing a system description and a list of iterative analysis commands. The language is case-sensitive.

The system-description language is a straightforward textual representation of linear hybrid automata. The user describes a system as the composition of a collection of components. Each component is given as a linear hybrid automaton. The system analyzed is taken as the product of all components given.

HyTech first passes its input through the macro preprocessor **m4**, allowing clear definition of constants in the system[5]. For example, we may declare and use the constant *raise_rate* in the gate automaton of Figure 2, as shown in the sample HyTech input appearing in Figure 4. The complete HyTech input file

[5] For details of the Unix command m4, type man m4.

for the parametric analysis of the train-gate-controller appears in Appendix A. Whitespace (blank spaces, tabs, new lines) between tokens is ignored. The syntax is described in more detail below. The grammar appears in Appendix B.

Comments The rest of an input line after two adjacent dashes (`--`) is taken as a comment.

Variables All variables in the system are declared at the top of the description, in a single declaration. Variables may be of the following types: discrete, clock, stopwatch, parameter, analog. The type declarations allow more readable descriptions and enable simple static checking by the parser. A clock variable always has rate 1, and a discrete variable always has rate 0. The rate of a stopwatch must be either 0 or 1. Parameters have rate 0 in all locations, and may never be assigned values. Analog variables have no syntactic restrictions. Variables of type discrete, clock and parameter are said to be *fixed rate* variables, since their rate intervals are fixed by their type, namely 0, 1 and 0 respectively. Constraints on their rates are automatically added to the rate conditions for each location; indeed, it is illegal for the user to constrain explicitly the rate of a fixed rate variable. For example, the variables for the train-gate controller example are declared as

```
var  x,                -- distance from intersection
     g: analog;        -- angle of gate
     t: stopwatch;     -- controller's timer
                       -- cutoff point for controller
     alpha: parameter; -- to issue commands
```

Linear terms, expressions and constraints A linear term is either (a) a variable multiplied by a rational coefficient, or (b) a rational number. A linear expression is an additive combination of linear terms. A linear constraint is an inequality[6] (`<`, `<=`, `>=`, `>`) or equality (`=`) between linear expressions. Note that rational coefficients must either (a) be an integer, (b) have an integer as numerator and a nonzero integer as denominator, or (c) be omitted, in which case it is understood to be 1. For example, `1/2x - 24/5y < z + 5t -6 + y` is a syntactically legal linear constraint.

Automaton components Each automaton is given a name which may be used later in the specification. Its synchronization labels are declared. Its initial location and the initial condition on its variables must also be provided. For example, the header for the train automaton is as follows:

```
automaton train
synclabs : app,       -- approach signal
           exit;      -- signal that train is leaving
initially far & x>=2000;
```

Each automaton component includes a list of locations, described below, terminated by the keyword `end`.

Locations Each location is named and labeled with its invariant. Rate conditions may also be provided, where the term `dx` is used to denote $\dot{x}$. The syntax `dg in [10,20]` is shorthand for `dg >= 10, dg <= 20`. For example, `loc far:  while x>=100 wait` {`dx in [-50,-40]`} is the header for the location *far* with invariant $x \geq 100$, and rate condition $-50 \leq \dot{x} \leq -40$. HyTech terminates whenever it detect illegal rate conditions.

Invariants may be conjunctions of linear constraints, such as `x>=1/2 & y<=2/3+x`, but must *not* be disjunctions[7]. Conjunctive rate conditions are separated by commas, as in `wait` {`dx = dz, dy in [2,4]`}.

Each location is associated with a list of transitions originating from it.

Transitions Each transition lists a guard on enablement and the successor location. Both the synchronization label and the update information are optional. For example, the following are legal transitions.

[6] Strict inequalities, previously unavailable in HyTech, are now supported.

[7] In order to model a disjunctive invariant, split the location into several locations, one for each disjunct [AHH93].

```
var
   final_reg, init_reg : region;

init_reg :=   loc[train] = far & x>=2000 & loc[controller] = idle
            & loc[gate] = open & g=90;
final_reg :=    loc[gate] = raising & x<=10 | loc[gate]=open & x<=10
             |  loc[gate] = lowering & x<=10;
print omit all locations
       hide non_parameters in
            reach forward from init_reg endreach & final_reg
       endhide;
```

Figure 5: Analysis commands for train-gate controller

```
when True goto far;
when x=1 & y<=2 do {} goto far;
when x=0 do { 1 <=x', x'<2, g' >=x+3} sync exit goto far;
when asap sync exit do {y'>=5} goto far;
```

Again, notice that guards may be conjunctions of linear constraints, but not disjunctions (use multiple transitions). Also, the order of the synchronization information and the assignments is interchangeable, if they appear at all, but the guard must appear first and the successor location last. It is assumed that the update set of the transition is precisely the set of variables for which the primed counterparts appear in the jump condition. The syntax `x' = x'` can be used to "assign" the variable x a nondeterministic value. The Asap guard on the last transition listed indicates it is an urgent transition which must take place as soon as possible. Recall that there is a syntactic restriction that non-trivial guards are not permitted on urgent transitions or any transitions in other components with the same synchronization label as an urgent transition.

Composition It is assumed that the system being described is the parallel composition of all listed components.

5 Input Language: Analysis Commands

The analysis section of the input consists of two parts: declaration of variables for regions, and a sequence of iterative command statements. Analysis commands provide a means of manipulating and outputting regions. Commands are built using objects of two basic types: *region expressions* for describing regions of interest, and *boolean expressions* 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 example, the specification commands in Figure 5 are for analyzing the train-gate controller. Their overall effect is to determine the critical bound on the parameter α. First, the two regions *final_reg* and *init_reg* are declared. The first two statements assign values to these regions using direct constraints on the states. Notice that disjunctions may be used. The third statement outputs the constraint on the parameter α under which the system is not correct. This printing command is given by the prefix **`print omit all locations`**, which tells HyTech to output the region enclosed between the words **`hide`** and **`endhide`**, but only after hiding all information about locations. We choose to omit all location information since for any particular value of α the specific final location reached is irrelevant. HyTech evaluates the region expression between the hide keywords by first performing reachability analysis from the initial region specified by *init_reg*, intersecting the reachable states with the final region (*final_reg*), and then existentially quantifying out all variables that are not declared as parameters. After 1.72 seconds computation on a Sparcstation 20, HyTech produces the following output, showing that the system is correct whenever $\alpha < 49/5$.

```
 5alpha >= 49
```

5.1 Region expressions

Region expressions are built from linear inequalities, constraints on locations, and region names, by existential quantification, *pre*, *post*, and convex hull operations, reachability, conjunction, and disjunction. Each region expression defines a region. The symbol ⟨*reg_exp*⟩ denotes an arbitrary region expression.

Linear inequalities The most basic region expression is a linear inequality. For example, `x <= 100` is a region expression, defining the set of all states where the variable x has value no greater than 100.

Location constraints `loc[`⟨*aut_name*⟩`] =` ⟨*loc_name*⟩.
The location name ⟨*loc_name*⟩ must be the name of a location in the automaton ⟨*aut_name*⟩. For example, the region expression `loc[gate] = open` defines the set of all states where the location component corresponding to the gate is *open*.

Boolean combinations `~`⟨*reg_exp*⟩ , ⟨*reg_exp*⟩ `&` ⟨*reg_exp*⟩ , ⟨*reg_exp*⟩ `|` ⟨*reg_exp*⟩
The negation of a region expression, written using the operator `~`, is a region expression (representing the complement of the its operand), as is the disjunction of region expressions (representing the union of its operands), written using the operator `|`, and the conjunction of region expressions (representing the intersection of its operands) is written with the operator `&`. The negation operator has highest precedence, followed by the `&` operator. An expression without parentheses is considered to be a disjunction of conjunctions. In addition, the boolean constants `True` and `False` have the expected meaning.

Difference `diff(`⟨*reg_exp*⟩`,` ⟨*reg_exp*⟩`)`
The set difference expression evaluates to the region representing all states satisfying its first argument but not its second argument. The region expression `diff(r1,r2)` is equivalent to the region expression `r1 & ~r2`.

Parentheses Any region expression may be enclosed in parentheses. For example, `x<=4 & (y<=5 | y>=5)` is equivalent to `x<=4`.

Region name A region expression may be any declared region variable. There is no automatic check that the region variable has been assigned a value. The value of the expression is the region most recently assigned to the variable.

Existential quantification `hide` ⟨*var_list*⟩ `in` ⟨*reg_exp*⟩ `endhide`
The `hide` expression evaluates to the region obtained by existentially quantifying a list of variables. For example, the command `print hide x in x<=1 & x=y endhide` outputs the region where $y \leq 1$. In general, quantified variables may be listed, separated by commas, as in `print hide x, z in x<=1 & y<=x+3 & z = y-x endhide`. Alternatively, the list ⟨*var_list*⟩ may be replaced by the keywords `all` (for all variables) or `non_parameters` (for all variables not declared as parameters).

Pre/Post `pre(`⟨*reg_exp*⟩`)`, `post(`⟨*reg_exp*⟩`)`
The `pre` and `post` expressions evaluate to the regions obtained by applying *pre* and *post* respectively to their arguments.

Convex hull `hull(`⟨*reg_exp*⟩`)`
The expression `hull(`⟨*reg_exp*⟩`)` returns the region where each location v is associated with the convex hull of all valuations s for which (v, s) is in the region defined by ⟨*reg_exp*⟩. For example,

```
loc1 := loc[P1]=loc_a & loc[P2]=loc_b_1;
loc2 := loc[P1]=loc_a & loc[P2]=loc_b_2;
approx := hull(loc1 & x=0 | loc1 & x=1 | loc2 & x=1);
```

assigns `approx` the region represented by `loc1&0<=x&x<=1 | loc2&x=1`.

Reachability `reach forward from` ⟨*reg_exp*⟩ `endreach`
`reach backward from` ⟨*reg_exp*⟩ `endreach`

There are two specialized expressions for returning the set of states reachable from any arbitrary region: one for forward reachability and one for backward reachability. For example, the expression `reach forward from init_reg endreach` appearing in the analysis commands in Figure 5 evaluates to the region reachable from *init_reg* by iterating *post*. The backward reachability expression iterates *pre* until convergence.

5.2 Boolean expressions

Boolean expressions are built from region comparisons and region emptiness checks using boolean operators. Boolean expressions are used in conditional statements and while loops. The symbol ⟨*bool_exp*⟩ denotes an arbitrary boolean expression.

Comparison between regions ⟨*reg_exp*⟩ ⟨*relop*⟩ ⟨*reg_exp*⟩

The relational operator ⟨*relop*⟩ is one of the symbols `<`, `<=`, `=`, `>=`, and `>`, representing the binary set comparison operators $\subsetneq$, $\subseteq$, $=$, $\supseteq$, and $\supsetneq$ respectively. For example, the following are legal boolean expressions.

```
init_reg = final_reg
init_reg >= loc1 & x <= 5
```

Emptiness `empty(`⟨*reg_exp*⟩`)`

The unary predicate `empty` applied to a region expression evaluates to true iff its argument contains no states. For example, the following code could be used to determine whether the system satisfies its safety requirement.

```
reached := reach forward from init_reg endreach;
if empty(reached & final_reg)
   then prints "System verified";
   else prints "System contains violations";
endif;
```

Boolean combinations ⟨*bool_exp*⟩ `and` ⟨*bool_exp*⟩, ⟨*bool_exp*⟩ `or` ⟨*bool_exp*⟩
`not` ⟨*bool_exp*⟩

Boolean expressions may be combined to yield boolean expressions. The negation of a boolean expression is a boolean expression. For example, `not empty(reached)` is a boolean expression. The conjunction and disjunction of boolean expressions are boolean expressions, with the natural meaning, written using the keywords `and` and `or`. Note that region expressions use the symbols `&` and `|`. Negation has highest priority and conjunctions bind more tightly than disjunctions.

5.3 Command statements

There are commands to perform common tasks such as error-trace generation and parametric analysis. Command statements are built from primitives for printing and assigning regions. Command statements may also occur within conditional statements and while statements. Each command is terminated by a semicolon.

Printing There are four basic commands for outputting information. All output appears on `stdout`.

`print` ⟨*reg_exp*⟩ The basic print command outputs the states in the region defined by its region expression argument. For example, the command `print init_reg` (see Figure 5) would produce the output

```
Location: far.idle.open
   g = 90 & x >= 2000
```

The print command prints out a list of locations and predicates defining the states associated with them. Non-convex predicates are output as disjunctions of convex predicates. Locations for which there are no associated valuations in the region do not appear in the output. The string `far.idle.open` indicates that the valuations satisfying the convex predicate $g = 90 \wedge x \geq 2000$ are associated with the control location where the train component is far from the intersection, the controller component is in its idle location, and the gate component in its open location. Note that location information is printed with periods separating the locations for each component, and that components are listed in the order in which they are declared.

`print omit` ⟨*loc_list*⟩ `locations` ⟨*reg_exp*⟩ This command generalizes the basic print command by first eliminating information about the locations of all components listed after the `omit` keyword. For example, if *strange_reg* is first assigned to

```
init_reg | loc[gate]=closed & 1000<=x & loc[train]=far
```

then `print omit gate, controller locations strange_reg` produces the output

```
  Location: far..
     x >= 1000
```

indicating that the region given includes only locations in the product automaton for which the train component is in its far location, and that all valuations for which the value of x is greater than or equal to 1000 appear in some such location. The absence of a location name for the second and third component automata indicates that information for these components' locations has been existentially quantified.

As shorthand, the keyword `all` may appear in place of a list of automata names, in which case all location information is quantified, as in Figure 5.

`prints` ⟨*string*⟩ This command prints strings, enclosed in double quotes, directly to `stdout`. For example, the statement `prints "Hi there"` outputs the string "`Hi there`" followed by a carriage return.

`printsize` ⟨*reg_name*⟩ This command prints information about the "size" of the region stored in the region variable given as an argument. Information output includes the number of product locations for which the associated predicate is nonempty and the total number of convex predicates used in representing the region.

Assignment ⟨*reg_name*⟩ `:=` ⟨*reg_exp*⟩
Any region expression may be assigned to any region name. For example, we may initialize the final region with the statement

```
final_reg := x<=10 & (  loc[gate] = raising
                      | loc[gate] = open
                      | loc[gate] = lowering);
```

which is equivalent to the assignment appearing in Figure 5.

Conditional The `if-then` and `if-then-else` statements have the expected meaning. For example, the following are legal conditional statements.

```
if init_reg<=final_reg then prints "Hi"; print strange_reg; endif;
if init_reg=final_reg then prints "Equal";
   else prints "Not equal"; endif;
```

The boolean expression comparing regions is first evaluated, and then the appropriate list of statements (if any) is executed.

Iteration The `while` statement has the expected meaning. For example,

```
reached := init_reg;
old := init_reg;
reached := post(old);
while not ( reached <= old ) do
   old := reached;
   reached := post(reached);
endwhile;
```

computes the set of reachable states from the initial states by iterating the *post* operation until a fixpoint is obtained.

Error trace generation `print trace to` ⟨*reg_exp*⟩ `using` ⟨*reg_name*⟩

HyTech provides a simple facility for generating error traces for faulty systems. One must first use the built-in reachability utility (see Subsection 5.1), which causes HyTech to store internal information that can be used to generate traces. Second, the command to generate traces is issued, specifying both the target region of the traces, and the name of the region variable previously used to store the result of the reachability analysis. This is best illustrated by an example. Suppose we are using forward reachability analysis to see whether any state in the violation region *final_reg* is reachable from the initial region *init_reg*. The following sequence of commands causes HyTech to generate an error trace, if one exists.

```
reached := reach forward from init_reg;
if empty(reached & final_reg)
  then prints "System verified";
  else prints "System contains violations";
       print trace to final_reg using reached;
endif;
```

The trace output consists of regions, *i.e.* sets of states, not individual states. Each region will be accessible from the previous via a time step allowing the continuous variables to evolve, followed by a transition step. The trace generated is minimal in length, and includes the synchronization labels, if any, for transitions between regions along the trace. Regardless of whether forward or backward reachability is used, the trace is always printed in an absolute forward direction.

Note: this command is rather fragile, and should be used with some care. The error trace generation command always assumes — without any automatic checks — that the region variable appearing after the keyword `using` (`reached` in the above example) has been assigned a reachable region using the built-in `reach` expression, and that no `reach` expression has since been evaluated.

5.4 Additional features

The following functions are also available in HyTech's command language. However, we advise you to use these features with care for two possible reasons; they may be extremely inefficient, or their usage is error-prone with their semantics perhaps not as you intend.

Freeing memory `free` ⟨*reg_name*⟩

There is a command statement for freeing the memory used to store a region. For example, the statement `free B1` frees the region stored in the variable `B1`.

The user must ensure any variable freed is not accessed again until it has been reassigned a value. Note that it is not necessary to free variables before assigning them new values: this is done automatically. Also, intermediate regions created in evaluating expressions are also freed automatically.

Weak difference operator `weakdiff (`⟨*reg_exp*⟩`,`⟨*reg_exp*⟩`)`

Region expressions can be formed using the weak difference operator `weakdiff`. This operator offers a fast method of gaining an overapproximation to the set difference of two regions. Its semantics are not straightforward, and it may result in unexpected behavior. For each location, the operator computes the *weak difference* of predicates associated with that location. The weak difference of predicates ϕ_1

and ϕ_2 is defined to be the disjunction of convex predicates occurring in ϕ_1 for which there does not exist an enclosing convex predicate occurring in ϕ_2. HyTech represents predicates as a disjunction of conjunctions. However, our tool does not use a unique representation for a given predicate: to do so would require inefficient normalization. Therefore predicates can be stored in numerous forms. Thus the result of the **weakdiff** operator depends not so much on the valuations satisfying the predicate as on their internal representation. For example, let ϕ_1 be represented as $x \geq 3 \vee x \leq 6$, ϕ_2 as $x \geq 2$, and ϕ_3 as $x \leq 4 \vee x \geq 4$. Let $\mathit{weakdiff}(\psi_1, \psi_2)$ denote the weak difference of ψ_1 and ψ_2. Then we have

$$\begin{aligned}
\mathit{weakdiff}(\phi_1, \phi_2) &= x \leq 6 \\
\mathit{weakdiff}(\phi_1, \phi_3) &= \phi_1 \\
\mathit{weakdiff}(\phi_3, \phi_1) &= \mathit{false} \\
\mathit{weakdiff}(\phi_3, \phi_2) &= x \leq 4
\end{aligned}$$

Despite its drawbacks, the operator is considerably faster to compute than `cldiff` and typically results in far fewer disjuncts.

Weak comparisons between regions ⟨*reg_exp*⟩ ⟨*weak_op*⟩ ⟨*reg_exp*⟩

Boolean expressions can be formed using weak comparison operators (**weakle**, **weakge**, and **weakeq**) between region expressions. The operator **weakle** (**weakge**) evaluates to true iff the weak difference of its second (first) operand and its first (second) operand is empty. The operator **weakeq** evaluates to true iff the weak difference of each operand with the other operand is empty.

Iteration **iterate** ⟨*reg_name*⟩ **from** ⟨*reg_exp*⟩ **using** { ⟨*commands*⟩ }

Region expressions can be formed using the iteration expression which returns the fixpoint obtained by repeatedly executing the body of a loop of commands until the value of the iteration variable (the ⟨*reg_name*⟩ variable above) stabilizes. The precise termination condition is that the region stored in the iteration variable after the loop is weakly equivalent to the region stored in the variable before the loop. For example, the following expression may be used to compute the set of states reachable from the initial region *init_reg*.

```
reachable := iterate B1 from init_reg using {B1 := post(B1);};
```

After executing this statement, the variable `B1` also stores the set of reachable states. Beware: this construct has side effects. All variables are global, so any variables altered within the loop of the iteration expression have their global values changed. This may or may not be desirable, so use with extreme caution.

As an example, consider the following more efficient means of computing the reachable states.

```
reachable := init_reg;
B2 := iterate B2
        from init_reg
        using {
              B2 := post(B2);
              B2 := weakdiff(B2 , reachable);
              reachable := reachable | B2;
        };
```

This time we are iterating over `B2`, a variable which contains the *newly* reachable states at each iteration. The iteration terminates when the set of new states is empty, with the side effect that `reachable` contains the set of all reachable states.

6 Examples

Additional examples may be found in the directory `examples` of the software distribution. We discuss three of them here in more detail.

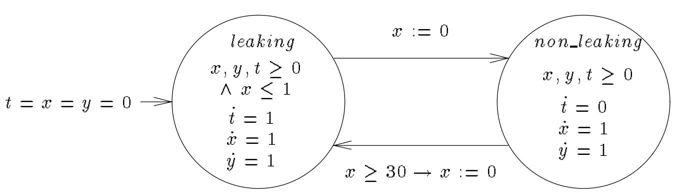

Figure 6: Automaton for the leaking gas burner

```
-- leaking gas burner
var  x,              -- time spent in current location
     y: clock;       -- total elapsed time
     t: stopwatch;   -- leakage time

automaton gas_burner
synclabs:;
initially leaking & t = 0 & x = 0 & y=0;
loc leaking: while x>=0 & y>=0 & t>=0 & x <=1 wait {dt=1}
   when True do {x'=0} goto not_leaking;
loc non_leaking: while x>=0 & y>=0 & t>=0  wait {dt=0}
   when x>=30 do {x'=0} goto leaking;
end

var init_reg, final_reg, b_reachable: region;

init_reg := loc[gas_burner] = leaking & x=0 & t=0 & y=0;
final_reg := y>=60 & t >= 1/20 y;
b_reachable := reach backward from final_reg endreach;
if empty(b_reachable & init_reg)
  then prints "Non-leaking duration requirement satisfied";
  else prints "Non-leaking duration requirement not satisfied";
endif;
```

Figure 7: Input file for the analysis of the gas burner

6.1 Gas burner

The "leaking gas burner" example has appeared in the early literature on formal methods applied to hybrid systems [CHR91, ACHH93]. We show how this simple system can be analyzed in HyTech. The gas burner is in one of two modes; it is either leaking or not leaking. Leakages are detected and stopped within 1 second. Furthermore, once a leakage has been stopped, the burner is guaranteed not to leak again until at least 30 seconds later. The system is initially leaking.

The linear hybrid automaton of Figure 6 models the gas burner. The clock x records the time elapsed since last entering the current location, and is sufficient for modeling the behavior of the system. However, in order to analyze the system, we need to add the auxiliary variables t and y. The stopwatch t measures the cumulative leakage time. It increases at rate 1 in the location *leaking*, and at rate 0 in location *non_leaking*. The clock y measures the total elapsed time. Using these auxiliary variables, we prove that if at least 60 seconds have passed, then the burner has been leaking for less than one twentieth of the total elapsed time. The requirement holds unless there is a state, forward reachable from the initial states, in which $y \geq 60$ and $t \geq y/20$. We compute the region backward reachable from all states satisfying $y \geq 60 \wedge t \geq y/20$. Since this region does not include any initial states, the requirement is satisfied. In fact, forward reachability for this system does not terminate. In general, it is not easy to determine ahead of time whether forward or backward reachability analysis is preferable.

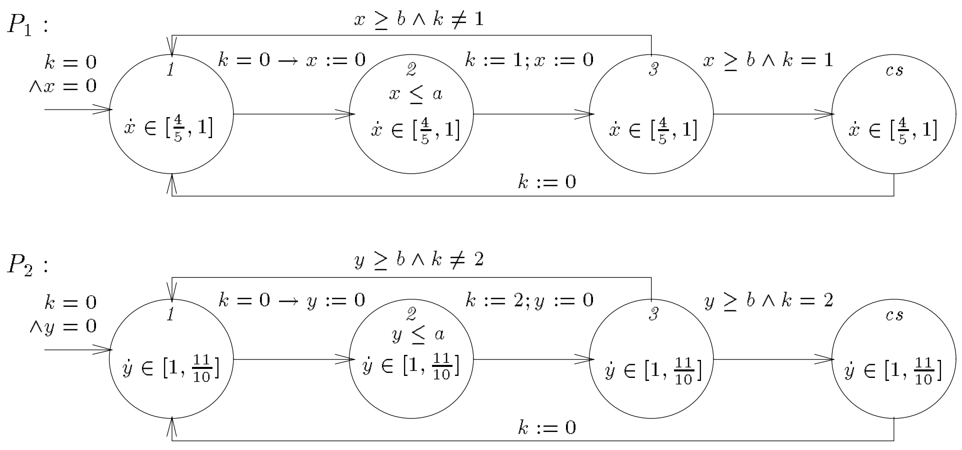

Figure 8: Automata for processes P_1 and P_2 in Fischer's mutual exclusion protocol

The complete input file for this example appears in Figure 7. HyTech outputs the string "Non-leaking duration requirement satisfied". The computation takes 0.62 seconds on a SparcStation 20, using a maximum of 0.73 MB of memory.

6.2 Fischer's mutual exclusion protocol

6.2.1 Parametric analysis

We demonstrate parametric analysis through a drifting clock version of the simple timing-based mutual-exclusion protocol due to Fischer [Lam87, AHH93]. The system consists of two processes, P_1 and P_2, each performing atomic read and write operations on a shared memory variable k. Each process P_i, for $i = 1, 2$, models the following algorithm:

```
repeat
    repeat
        await $k = 0$; $k := i$; delay $b$
    until $k = i$
    Critical section
    $k := 0$
forever
```

The instruction **delay** b delays a process for at least b time units as measured by its local clock. Process P_i is allowed to enter its critical section iff $k = i$. Furthermore, each process takes no more than a local time units to write a value into the variable k, *i.e.* the assignment $k := i$ occurs within a time units after the **await** statement completes. To complicate matters, the two processes use drifting clocks. Process P_1's clock is slow, and its rate may vary between 0.8 and 1, while that of P_2 is fast with rate between 1 and 1.1.

The automata for the two processes appear in Figure 8. Each process is modeled using the private clocks x and y, respectively. Each process has a critical section, represented by the location *cs* in each automaton. The invariants at location *2* ensure the upper time bound on the write access to k, while the guards on the transitions from location *3* to location *cs* model the lower time bound of the delay.

We perform parametric analysis to determine the values for a and b, if any, for which mutual exclusion holds. The "unsafe" region is characterized by the region expression `loc[P1]=cs & loc[P2]=cs`. As for the train-gate controller example, we are interested in the values of the parameters for which there exists a reachable unsafe state. These values are output using the `print omit all locations` analysis command, in conjunction with existential quantification of the non-parameter variables:

```
init_reg := loc[P1] = loc_1 & loc[P2] = loc_1 & k=0;
final_reg := loc[P1] = cs & loc[P2] = cs ;
print omit all locations hide non_parameters in
  reach forward from init_reg endreach & final_reg endhide;
```

HyTech's computation takes 1.46 seconds using 1.1 MB of memory, producing the following output, which indicates that the system is correct whenever $a < 8b/11$.

```
11a >= 8b & a >= 0
```

The full input and output files appear in Appendix A.

6.2.2 Error-trace generation

We demonstrate the generation of error traces for a system that does not meet the parametric requirements for correctness. If we set a to 5, and b to 6, then mutual exclusion will not hold. Since we are interested in generating error traces, it is helpful to label transitions, even if they do not participate in any synchronization. In an error trace, the synchronization labels give a clear indication of which transitions are taken. The full input and output files are found in Appendix A.

6.3 Corbett's distributed controller

Corbett [Cor94] describes a distributed robot control system, derived from [GL92]. The goal of the system is to provide timely robot commands based on recent measurements of the environment. The system consists of four main components: two sensors, a scheduler, and a controller. Each sensor repeatedly constructs readings for sending to the controller. Sensor 1 takes between 0.5 and 1.1 milliseconds of CPU time to take a reading, while sensor 2 requires between 1.5 and 2.0 milliseconds. The two sensors share a single processor with the controller executing on its own dedicated processor. Our model deviates slightly from Corbett's where the sensors and the controller all share one processor. The scheduler ensures that sensor 2 has priority over sensor 1. If sensor 1 is interrupted by sensor 2, it is rescheduled as soon as sensor 2 completes, and continues constructing its reading as if uninterrupted. Once a reading is complete, it is sent to the controller, as soon as the controller is ready to receive it. After forwarding its reading to the controller, a sensor must delay for 6 milliseconds before starting a new reading. Sensor readings are considered stale and are remeasured if they are not sent to the controller soon enough after being completed.

The role of the controller is to deliver appropriate commands to the robot based on both sensors' readings. Commands cannot be computed unless the two sensors' readings are received at most 10 milliseconds apart. Given two fresh readings, the controller requires from 3.6 to 5.6 milliseconds to calculate the robot command. Sensor readings are also acknowledged, and this takes between 0.9 and 1.0 milliseconds.

One property of interest to Corbett is whether the controller always generates a command signal within a certain time bound. We show here how HyTech can be used to determine the precise maximal delay between commands in our system.

The system is modeled using the linear hybrid automata appearing in Figure 9. Sensor 1 has four locations: *idle*, *read*, *wait*, and *send*. It uses the stopwatch variable y_1 to enforce its timing constraints. At the start of execution, it is in its *idle* location with $y_1 = 6$, indicating that it has delayed sufficiently long to initiate a read request via a $request_1$ event. In location *read*, the sensor waits until a $read_1$ event occurs, indicating that the scheduler has provided it with sufficient CPU time to construct a reading. The sensor remains in location *wait* for up to 4 milliseconds, modeled by the incoming reset $y_1 := 0$ and the location invariant $y_1 \leq 4$. During this time, it is ready to send its reading to the controller via a $send_1$ event. The transition labeled $send_1$ is an urgent transition that takes place as soon as the controller is ready to synchronize, *i.e.* receive, the reading. After sending the reading, sensor 1 waits in location $send_1$ for an acknowledgement, modeled by an ack_1 event, from the controller. It then delays 6 milliseconds before initiating a new reading. However, if the controller and sensor cannot synchronize a $send_1$ event in time, sensor 1 takes a transition back to its *read* location and requests CPU time for a completely new reading. Sensor 2 is identical in structure.

The scheduler allocates each sensor CPU time on a shared processor. It synchronizes with the two sensors on the labels $request_1$ and $request_2$, which correspond to the sensors' initiating a request for CPU time to

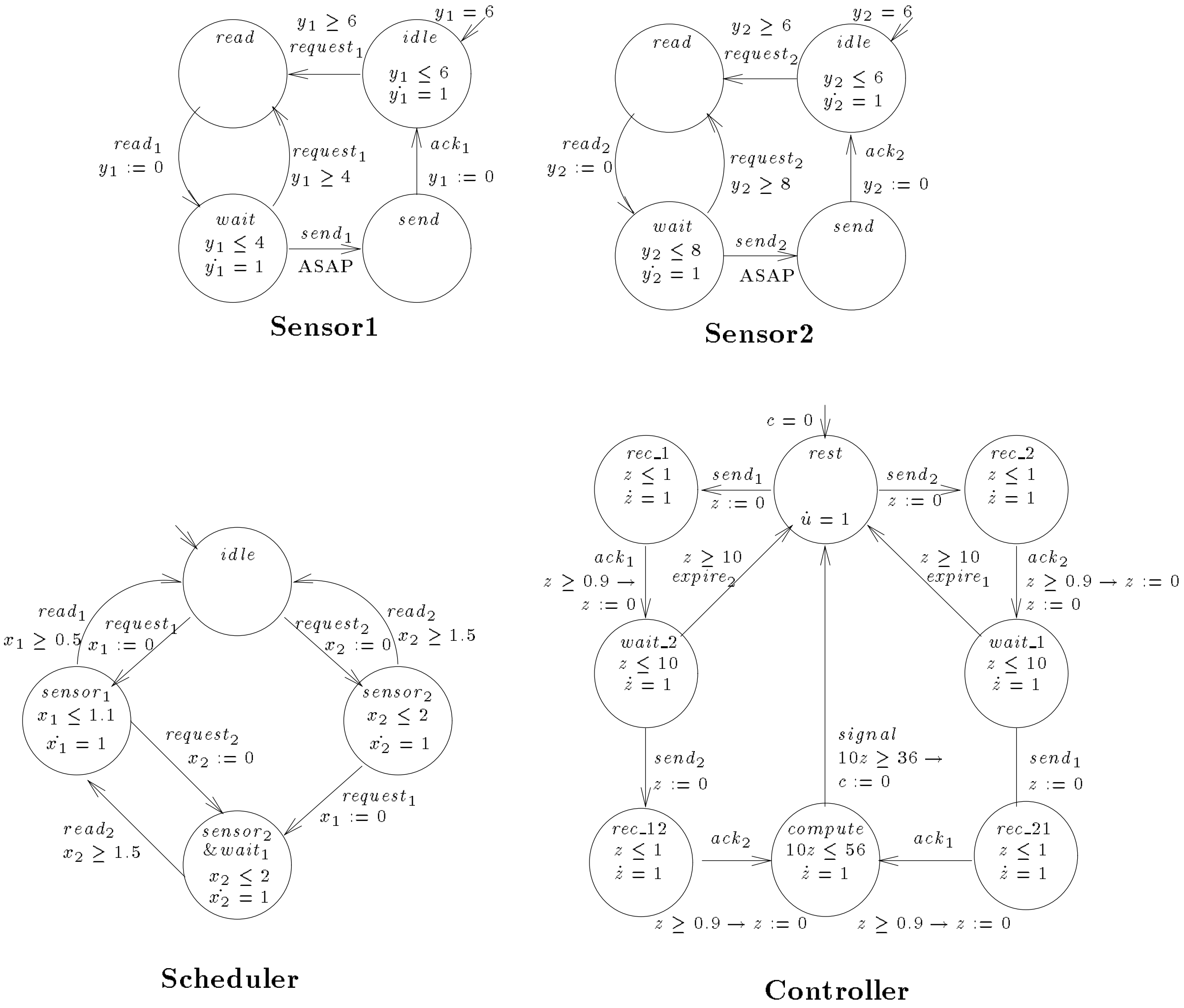

Figure 9: Corbett's distributed robot controller

construct a reading, and on the labels $read_1$ and $read_2$, which correspond to the completion of a reading. The automaton uses two stopwatch variables, x_1 and x_2 to model the CPU times that sensor 1 and sensor 2 have received since their last requests. For each x_i, the rate of x_i is 1 if sensor i is scheduled on the processor, and equals 0 otherwise. When sufficient time has accumulated for a sensor, the corresponding *read* transition may occur. The scheduler has four locations, one for each combination of pending requests: *idle* (no pending requests), $sensor_1$ (only sensor 1 has requested a reading), $sensor_2$ (only sensor 2 has a pending request), and $sensor_2\&wait_1$ (both sensors have pending requests). Priority for sensor 2 is achieved by giving sensor 2 the CPU when both sensors request it, *i.e.* in the location $sensor_2\&wait_1$ the variables' rates are given as $\dot{x}_2 = 1$ and $\dot{x}_1 = 0$.

The controller uses the clock variable z to control its behavior. Initially, it is waiting in location *idle* for a *send* signal from either sensor. The signal is acknowledged and a waiting location is entered. In location $wait_2$, up to 10 milliseconds is allowed to pass. If a $send_2$ event does not occur in time, the invariant $z \leq 10$ forces control to return to the *idle* location via an $expire_1$ event. If a second *send* event does occur in a timely fashion, it is duly acknowledged, and the invariant and outgoing guard for location *compute* model the time required to calculate the command to send to the robot.

For our analysis, we introduce an additional clock c which is used to measure the time elapsed since the last signal was sent to the robot, or since execution began if there has been no signal sent yet. The clock has initial value 0 and is reset every time a signal occurs. HyTech determines the maximal delay between signals by computing the set of values of c occurring in any reachable state. This information is output via the following analysis command that hides all information other than the value of c.

```
print omit all locations
    hide x1, x2, y1, y2, z in
        reach forward from init_reg endreach
    endhide;
```

The complete input file appears in Appendix A.

7 Designing Requirement Specifications

It is not always obvious how to specify requirements of systems. This section provides some hints to the verification engineer by outlining how to check for many common classes of requirements. All forms of specifications below rely on the use of reachability analysis.

7.1 Simple safety

A safety requirement intuitively asserts that “nothing bad ever happens”. Many specifications are expressed naturally as safety requirements. A system is said to be correct iff its reachable states all satisfy an invariant ϕ, defining a set of safe states: the “bad thing” to happen is to reach a state that does not satisfy the invariant[8]. For example, Fischer's mutual exclusion protocol should guarantee that processes P_1 and P_2 are never in their critical sections at the same time. Also, the train-gate controller is required to ensure that the gate is always down whenever the train is within 10 feet.

As discussed above (Subsection 2.3), safety requirements can be verified in HyTech using the region $\neg\phi$. One method is to perform forward reachability analysis from the system's initial states, and then check whether the intersection with the violating states $\neg\phi$ is empty. Assuming the region *init_reg* has been assigned the set of initial states, and *viol* has been set to the region $\neg\phi$, the following HyTech input checks the safety requirement, and generates an error trace if any exists.

```
f_reachable := reach forward from init_reg endreach;
reached_viol := f_reachable & viol;
if empty(reached_viol)
  then prints "System verified";
  else prints "System not verified";
       prints "The violating states reached are";
       print reached_viol;
       print trace to viol using f_reachable;
endif;
```

Alternatively, the analogous backward reachability analysis can be used.

```
b_reachable := reach backward from viol endreach;
init_reach_viol := b_reachable & init_reg;
if empty(init_reach_viol)
  then prints "System verified";
  else prints "System not verified";
       print trace to viol using b_reachable;
endif;
```

[8] The reader familiar with temporal logics should observe that such requirements are expressed in the form $\forall\Box\phi$, meaning intuitively that ϕ is always true for all reachable states of the system.

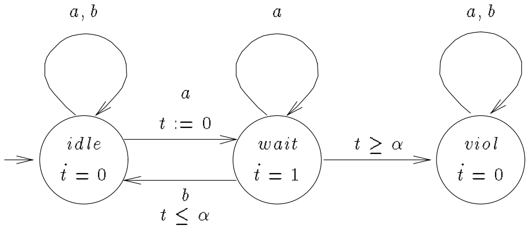

Figure 10: Generic bounded-response monitor automaton

7.2 Simple possibility

A simple possibility requirement asserts that "something good can always happen." If the notion of "something good" can be expressed as a region expression ϕ, then such requirements maintain that all states forward reachable from the initial states are backward reachable from a state in ϕ.[9] For example, we may wish to prove that for Fischer's mutual exclusion protocol, there is always a possibility that process P_1 will enter its critical section sometime in the future. The following HyTech code checks this assertion.

```
b_reachable := reach backward from loc[P1] = cs endreach;
f_reachable := reach forward from init_reg endreach;
if f_reachable <= b_reachable
  then prints "Requirement satisfied";
  else prints "Requirement not satisfied";
endif;
```

7.3 Simple real-time and duration requirements

Many simple real-time requirements can be specified by introducing clocks and stopwatches to measure delays between events, or the length of time a particular condition holds. In the gas burner example, we assert that as long as a minute or more has passed, the burner has been leaking no more than 5% of the time. In this case, we introduce a new variable for each time duration of interest. We need to know the total elapsed time and the time spent in location *leaking*. These quantities are measured by the clock y and stopwatch t respectively. The duration requirement we are interested in then becomes the safety requirement where the violating states are given by the predicate $y \geq 60 \wedge t \geq y/20$.

7.4 Additional requirements

By no means do all requirement specifications fall into the categories discussed above. However, a simple technique can be used to reduce many requirements to safety requirements. The idea is to build a separate monitor automaton for the requirement being checked [VW86]. The monitor typically contains special states which are only reachable by violating executions. The monitor must act strictly as an observer of the original system, without changing its behavior. Reachability analysis may then be performed on the parallel composition of the system and the monitor, with the system correct iff no violating state in the monitor is reached. To illustrate the technique, we use the category of bounded-response requirements.

7.4.1 Bounded response

A bounded-response requirement asserts that if something (a trigger event, a say) happens, then a response, b say, occurs strictly within a certain time limit α.[10] For example, one may assert that every approach of the train is followed by a *raise* command within 10 seconds. To verify these requirements, it is often easiest

[9] These requirements are expressed in temporal logics in the form $\forall\Box\exists\Diamond\phi$.

[10] This assertion is denoted $\forall\Box(a \Rightarrow \forall\Diamond_{<\alpha}b)$.

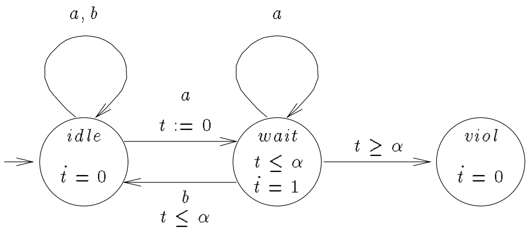

Figure 11: Bounded-response monitor automaton — strict bound

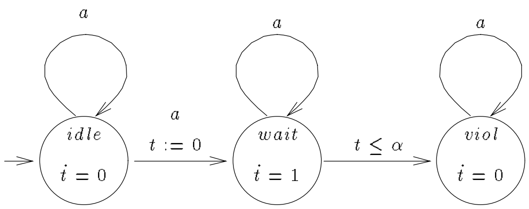

Figure 12: Monitor automaton for minimal event-separation time

to introduce a new stopwatch variable, t say, and build a monitor process with three locations: *idle*, *wait* and *viol*. Figure 10 depicts a generic automaton for bounded-response requirements. Control is initially in the *idle* location. When a trigger event occurs in a non-violating location, control may pass to the *wait* location and the clock t is reset. Response events cause control to return to the *idle* location. The unlabeled transition from the *wait* location to *viol* is only enabled when $t \geq \alpha$, *i.e.* time for the response event has passed by. This automaton will reach its violation location iff it is possible for α time units to pass after an a event without a b event occurring. Therefore, the violation location is not reachable iff every a event is followed by a b event occurring less than α time units later.

To assert that the response event may occur any time up to and including α time units after the trigger event, we may use the same monitor automaton as above, but checking that the violation location is only ever reached with the value of t being α.

Since bounded-response requirements occur frequently, we demonstrate how strict bounded-response requirements can be verified slightly more efficiently, *i.e.* the response event must occur before the response time — occurring when exactly α time units have passed is not acceptable. The monitor in Figure 11 is slightly more deterministic than that of Figure 10 and will generally lead to a less complex reachable region. Note that the selfloops on the violation location have been omitted. Although this affects the behavior of the system, it does so in a way that has no effect on its correctness, assuming we use forward reachability; once a violation has been detected, which additional states are reachable is irrelevant.

7.4.2 Minimal event separation

Monitor processes can be built to verify that events occur with some minimal separation time. For example, Figure 12 shows the automaton for verifying that no two instances of the event a occur within α time units of each other.

8 Installing and Running HyTech

8.1 Installation

HyTech runs under various flavors of Unix. While developed under SunOS, executables are available for a variety of platforms. Most jobs we have run require less than 20MB, many less than 10MB. However, obviously, the more memory the better.

The version of HyTech described here was first released in August 1995, and is freely available through the World-Wide Web via HyTech's home page http://www.eecs.berkeley.edu/~tah/HyTech. Download the main file `hytech.tar.Z`. It must be uncompressed to `hytech.tar`, and then expanded using the Unix `tar` command. The following sequence of commands will produce the directory `HyTech`.

```
uncompress hytech.tar.Z
tar -xf hytech.tar
```

The HyTech directory contains the subdirectories `src`, `bin`, `user_guide`, and `examples`, containing the source code, executable file for the Sun4 architecture (SunOS compiled), a copy of this user guide, and examples, respectively. The main directory also contains the files `README` and `license`. Please sign a copy of the license and follow the instructions given on the form. Licensed users will be assured of being informed about new releases of the software. We would also appreciate hearing about your experiences with HyTech and the applications you analyze with it.

HyTech has been successfully compiled on a number of platforms, and the following additional executables are also available through the Web page.

Sun4 (Solaris compiled)	`hytech-Solaris.exe`
PC x86 (Linux)	`hytech-Linux.exe`
DEC 5000	`hytech-DEC.exe`
DEC Alpha	`hytech-DEC-alpha.exe`
HP 9000	`hytech-HP.exe`

8.2 Executing HyTech

You must have the shell script file `hytech` (from the `bin` subdirectory) and a binary file called `hytech.exe` on your command search path. For example, to use HyTech under Linux, you should obtain the file `hytech-Linux.exe` from the Web page, rename it `hytech.exe`, and place it in a directory on your search path. HyTech also requires that the standard macro preprocessor `m4` to be on your command search path. Assuming your input file is called `a.hy`, the basic command to run HyTech is

```
hytech a.hy
```

The `.hy` suffix on the filename may be omitted. Output appears on `stdout`, so it is usually directed to a file via a command such as

```
hytech a.hy > a.log
```

HyTech creates and removes temporary input files in `/tmp`.

Options Available options are displayed by executing HyTech with no flags and no input file. Options are given in the form -⟨*flag_type*⟩⟨*n*⟩, and must occur before the filename on the command line. There are options for controlling the amount of output generated (`-p0`, `-p1`, and `-p2`, where the higher numbers indicate more verbose output), the format of the output (`-f0` for conjunctions output along a single line, and `-f1` for conjuncts listed one per line), whether to perform consistency checks on the input automata (`-c0` for no checks, `-c1` for checks), more information on the particular version of HyTech (`-i`), for performing simple reachability on the control space while forming the composition of automata (`-r1` for performing reachability analysis, `-r0` for none), and for performing more expensive computations in order to avoid the possibility of arithmetic overflow errors, (`-o0` for no avoidance, `-o2` for maximal effort to avoid overflow, and `-o1` for some avoidance attempted).

Examples Numerous sample input files and their output logs can be found in the subdirectory `examples`. Examine these to familiarize yourself with the input description language. Some of them are discussed in the user guide and [HHWT95a].

Bugs, comments, suggestions Please report any bugs or installation and maintenance problems to hytech@eecs.berkeley.edu. We do not have the resources to provide commercial-level support, but we can probably help you. We also welcome comments and suggestions, since the experience of HyTech's users will help to improve future versions of the software.

9 Hints for the Efficient Use of HyTech

This section describes hints on how to make the most of HyTech's computational power. If HyTech does not terminate on your input file, and you cannot figure out why, trying these heuristics may well help. Sometimes a slightly modified description will make a tremendous difference. As a general principle, keep your model of the system as simple as possible at first. Once HyTech has successfully analyzed the system, slowly add more detail to your model.

Keep the system description small. Generally, the smaller the better, *i.e.* try to minimize the number of components, locations, and variables. For example, try to model only a small number of the system's components first. Share locations wherever possible, *e.g.* error locations can often be combined into one. Some locations may be eliminated if they are "intermediate" locations not involved in direct synchronization with other components, and time spent in these locations can be transferred to the immediately adjacent locations.

Encode discrete variables into locations. For a bounded discrete variable, it is generally more efficient to split each location into several locations, one for each value of the variable, than to declare the variable as a real-valued variable. However, the increased efficiency often carries the disadvantage of a less compact description.

Manually compose tightly coupled components. When taking the product of two automata, many product locations are irrelevant since they are unreachable. If two components are tightly coupled with synchronized events, the reachable product automaton can be substantially smaller than the complete product. It may be beneficial to input the reachable product of such automata, instead of their components, since this version of HyTech constructs complete products only.

Keep constants simple. Generally, the lower the lcm:gcd ratio of the constants in the system, the faster the reachability analysis. Indeed, lowering the ratio may be necessary for reachability to terminate. To achieve low lcm:gcd ratios, it is often possible to verify an abstracted system where lower bounds are rounded down to smaller constants, and upper bounds are rounded up [AIKY92].

Model urgent events explicitly. If an event is urgent, model this fact directly where possible by using the Asap guard. This is more efficient than introducing an auxiliary clock.

Exploit "don't care" information. In many locations of an automaton, not all variable values are relevant. However, reachability analysis will record the exact values of such "don't care" variables. Thus to simplify the reachable region, it is helpful to make these variables completely unknown wherever they are irrelevant. This can be achieved by explicitly assigning them into the interval $(-\infty, \infty)$ on all transitions into the appropriate locations, using the syntax `x'=x'` for the variable x. A tempting option is to set them to a particular fixed value while control remains in a given location. However, this strategy is not as beneficial as assigning them into $(-\infty, \infty)$, since there is a nontrivial relationship between them and any other variables as time passes.

Use strong invariants. Sometimes it is helpful to restrict reachability analysis as much as possible through the use of strong invariants. For instance, enforcing implicit invariants can be advantageous, particularly when performing backward reachability analysis. In the gas burner example, backward reachability is required, since forward reachability does not terminate. It would be easy (and natural) to model the system without using the invariants $x \geq 0$, $y \geq 0$, and $t \geq 0$ for the clock and stopwatch variables. These invariants would play no role in forward analysis. However, backward analysis is nonterminating without these invariants, whereas adding them causes termination in six iterations.

Use the reachability facility provided. It is optimized for its task and faster than writing your own while loops. It also enables error traces to be generated.

Try forward and backward analysis. It is often not easy to predict which direction will terminate faster.

Free memory. Free any regions that will not be used again.

Acknowledgements We thank Grzegorz Czajkowski for implementing much of the parser, and Peter Kopke for implementing the fundamental routines for handling strict inequalities.

References

[ACH93] R. Alur, C. Courcoubetis, and T.A. Henzinger. Computing accumulated delays in real-time systems. In C. Courcoubetis, editor, *CAV 93: Computer-aided Verification*, Lecture Notes in Computer Science 697, pages 181–193. Springer-Verlag, 1993.

[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 R.L. Grossman, A. Nerode, A.P. Ravn, and H. Rischel, editors, *Hybrid Systems*, Lecture Notes in Computer Science 736, pages 209–229. Springer-Verlag, 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 *Proceedings of the 14th Annual Real-time Systems Symposium*, pages 2–11. IEEE Computer Society Press, 1993. Full version appears in *IEEE Transactions on Software Engineering*, 22(3):181–201, 1996.

[AIKY92] R. Alur, A. Itai, R.P. Kurshan, and M. Yannakakis. Timing verification by successive approximation. In G. von Bochmann and D.K. Probst, editors, *CAV 92: Computer-aided Verification*, Lecture Notes in Computer Science 663, pages 137–150. Springer-Verlag, 1992.

[BER94a] A. Bouajjani, R. Echahed, and R. Robbana. Verification of context-free timed systems using linear hybrid observers. In D.L. Dill, editor, *CAV 94: Computer-aided Verification*, Lecture Notes in Computer Science 818, pages 118–131. Springer-Verlag, 1994.

[BER94b] A. Bouajjani, R. Echahed, and R. Robbana. Verifying invariance properties of timed systems with duration variables. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *FTRTFT 94: Formal Techniques in Real-time and Fault-tolerant Systems*, Lecture Notes in Computer Science 863, pages 193–210. Springer-Verlag, 1994.

[BR95] A. Bouajjani and R. Robbana. Verifying ω-regular properties for subclasses of linear hybrid systems. In P. Wolper, editor, *CAV 95: Computer-aided Verification*, Lecture Notes in Computer Science 939, pages 437–450. Springer-Verlag, 1995.

[Cer92] K. Cerāns. Decidability of bisimulation equivalence for parallel timer processes. In G. von Bochmann and D.K. Probst, editors, *CAV 92: Computer-aided Verification*, Lecture Notes in Computer Science 663, pages 302–315. Springer-Verlag, 1992.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the Fifth Annual Symposium on Principles of Programming Languages*. ACM Press, 1978.

[CHR91] Z. Chaochen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.

[Cor94] J.C. Corbett. Modeling and analysis of real-time Ada tasking programs. In *Proceedings of the 15th Annual Real-time Systems Symposium*, pages 132–141. IEEE Computer Society Press, 1994.

[DOY94] C. Daws, A. Olivero, and S. Yovine. Verifying ET-LOTOS programs with KRONOS. In *Proceedings of Seventh International Conference on Formal Description Techniques*, pages 227–242. Chapman & Hall1994.

[DY95] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with kRONOS. In *Proceedings of the 16th Annual Real-time Systems Symposium*, pages 66–75. IEEE Computer Society Press, 1995.

[GL92] R. Gerber and I. Lee. A layered approach to automating the verification of real-time systems. *IEEE Transactions on Software Engineerging*, 9(18):768–784, September 1992.

[Hal93] N. Halbwachs. Delay analysis in synchronous programs. In C. Courcoubetis, editor, *CAV 93: Computer-aided Verification*, Lecture Notes in Computer Science 697, pages 333–346. Springer-Verlag, 1993.

[Hen92] T.A. Henzinger. Sooner is safer than later. *Information Processing Letters*, 43:135–141, 1992.

[Hen95] T.A. Henzinger. Hybrid automata with finite bisimulations. In Z. Fülöp and F. Gécseg, editors, *ICALP 95: Automata, Languages, and Programming*, Lecture Notes in Computer Science 944, pages 324–335. Springer-Verlag, 1995.

[HH95a] T.A. Henzinger and P.-H. Ho. Algorithmic analysis of nonlinear hybrid systems. In P. Wolper, editor, *CAV 95: Computer-aided Verification*, Lecture Notes in Computer Science 939, pages 225–238. Springer-Verlag, 1995.

[HH95b] T.A. Henzinger and P.-H. Ho. HYTECH: The Cornell Hybrid Technology Tool. In P. Antsaklis, A. Nerode, W. Kohn, and S. Sastry, editors, *Hybrid Systems II*, Lecture Notes in Computer Science 999, pages 265–293. Springer-Verlag, 1995.

[HH95c] T.A. Henzinger and P.-H. Ho. A note on abstract-interpretation strategies for hybrid automata. In P. Antsaklis, A. Nerode, W. Kohn, and S. Sastry, editors, *Hybrid Systems II*, Lecture Notes in Computer Science 999, pages 252–264. Springer-Verlag, 1995.

[HHK95] M.R. Henzinger, T.A. Henzinger, and P.W. Kopke. Computing simulations on finite and infinite graphs. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 453–462. IEEE Computer Society Press, 1995.

[HHWT95a] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: the next generation. In *Proceedings of the 16th Annual Real-time Systems Symposium*, pages 56–65. IEEE Computer Society Press, 1995.

[HHWT95b] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. A user guide to HYTECH. In E. Brinksma, W.R. Cleaveland, K.G. Larsen, T. Margaria, and B. Steffen, editors, *TACAS 95: Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science 1019, pages 41–71. Springer-Verlag, 1995.

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

[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.

[Ho95] P.-H. Ho. *Automatic Analysis of Hybrid Systems.* PhD thesis, Department of Computer Science, Cornell University, 1995.

[HRP94] N. Halbwachs, P. Raymond, and Y.-E. Proy. Verification of linear hybrid systems by means of convex approximation. In B. LeCharlier, editor, *SAS 94: Static Analysis Symposium*, Lecture Notes in Computer Science 864, pages 223–237. Springer-Verlag, 1994.

[HWT95a] T. A. Henzinger and H. Wong-Toi. Linear phase-portrait approximations for nonlinear hybrid systems. In R. Alur and T.A. Henzinger, editors, *Hybrid Systems III*, Lecture Notes in Computer Science. Springer-Verlag, 1995.

[HWT95b] P.-H. Ho and H. Wong-Toi. Automated analysis of an audio control protocol. In P. Wolper, editor, *CAV 95: Computer-aided Verification*, Lecture Notes in Computer Science 939, pages 381–394. Springer-Verlag, 1995.

[KPSY93] Y. Kesten, A. Pnueli, J. Sifakis, and S. Yovine. Integration graphs: a class of decidable hybrid systems. In R.L. Grossman, A. Nerode, A.P. Ravn, and H. Rischel, editors, *Hybrid Systems*, Lecture Notes in Computer Science 736, pages 179–208. Springer-Verlag, 1993.

[Lam87] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, 1987.

[LPY95] K. G. Larsen, P. Pettersson, and W. Yi. Compositional and symbolic model-checking of real-time systems. In *Proceedings of the 16th Annual Real-time Systems Symposium*, pages 76–87. IEEE Computer Society Press, 1995.

[LS85] N. Leveson and J. Stolzy. Analyzing safety and fault tolerance using timed Petri nets. In *Proceedings of International Joint Conference on Theory and Practice of Software Development*, Lecture Notes in Computer Science 186, pages 339–355. Springer-Verlag, 1985.

[MPS95] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In E.W. Mayr and C. Puech, editors, *STACS 95: Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science 900, pages 229–242. Springer-Verlag, 1995.

[MV94] J. McManis and P. Varaiya. Suspension automata: a decidable class of hybrid automata. In D.L. Dill, editor, *CAV 94: Computer-aided Verification*, Lecture Notes in Computer Science 818, pages 105–117. Springer-Verlag, 1994.

[NOSY93] X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. An approach to the description and analysis of hybrid systems. In R.L. Grossman, A. Nerode, A.P. Ravn, and H. Rischel, editors, *Hybrid Systems*, Lecture Notes in Computer Science 736, pages 149–178. Springer-Verlag, 1993.

[NSY92] X. Nicollin, J. Sifakis, and S. Yovine. Compiling real-time specifications into extended automata. *IEEE Transactions on Software Engineering*, SE-18(9):794–804, 1992.

[OSY94] A. Olivero, J. Sifakis, and S. Yovine. Using abstractions for the verification of linear hybrid systems. In D.L. Dill, editor, *CAV 94: Computer-aided Verification*, Lecture Notes in Computer Science 818, pages 81–94. Springer-Verlag, 1994.

[PV94] A. Puri and P. Varaiya. Decidability of hybrid systems with rectangular differential inclusions. In D.L. Dill, editor, *CAV 94: Computer-aided Verification*, Lecture Notes in Computer Science 818, pages 95–104. Springer-Verlag, 1994.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Annual Symposium on Logic in Computer Science*, pages 322–331. IEEE Computer Society Press, 1986.

Appendix A HyTech Input and Output Files

A.1 Train-gate controller

```
-- train-gate controller

var
        x,       -- distance
        g,       -- angle of gate
        t        -- dcontrollers timer
        : analog;
        alpha  -- cutoff point for controller to issue commands
        : parameter
        ;
 -- -------------------------------------------------------------- *)

automaton train
synclabs : app,         -- approach signal for train
           exit;        -- signal that train is leaving

initially far & x>=2000;

loc far: while x>=1000  wait {dx in [-50,-40]}
        when x=1000 sync app goto near;

loc near: while x>=0   wait {dx in [-50,-30]}
        when x=0 goto past;

loc past: while x<=100 wait {dx in [ 50, 30]}
        when x=100 do {x' >= 2000} sync exit goto far;

end -- train

automaton controller
synclabs: app,
          exit,
          lower,         -- lower command sent to the gate
          raise;         -- raise command sent to the gate
initially idle;

loc idle: while True wait {dt = 0} -- wait for a signal from train
        when True sync app do {t' = 0} goto about_to_lower;
        when True sync exit do {t' = 0} goto about_to_raise;

loc about_to_lower: while t<=alpha wait {dt = 1}
        when True sync app goto about_to_lower;
        when True sync exit do {t' = 0} goto about_to_raise;
        -- send lower signal any time before t<=alpha;
        when True sync lower goto idle;

loc about_to_raise: while t<=alpha wait {dt in [1,1]}
        when True sync app do {t' = 0} goto about_to_lower;
        when True sync exit  goto about_to_raise;
        -- send raise signal any time before t<=alpha
        when True sync raise  goto idle;

end -- controller
```

```
automaton gate
synclabs: raise, lower;
initially open & g=90;

loc raising: while g<=90 wait {dg = 9}    -- gate is being raised
        -- gate is fully raised
        when g=90 goto open;
        -- selfloops for input enabledness
        when True sync raise goto raising;
        when True sync lower goto lowering;

loc open: while True wait {dg = 0}        -- wait for command
        when True sync raise goto open;
        when True sync lower goto lowering;

loc lowering: while g>=0 wait {dg = -9}   -- gate is being lowered
        -- gate is fully lowered
        when g=0 goto closed;
        when True sync lower goto lowering;
        when True sync raise goto raising;

loc closed: while True wait {dg = 0}      -- wait for command
        when True sync raise goto raising;
        when True sync lower goto closed;

end -- gate

-- analysis commands

var
   final_reg, init_reg : region;

init_reg :=  loc[train] = far & x>=2000
           & loc[controller] = idle
           & loc[gate] = open & g=90;

final_reg := loc[gate] = raising & x<=10
          | loc[gate] = open & x<=10
          | loc[gate] = lowering & x<=10;


print omit all locations
       hide non_parameters in
            reach forward from init_reg endreach
            & final_reg
       endhide;
```

A.2 Fischer mutual exclusion protocol

A.2.1 Input file — parametric analysis

```
var
        x,              -- P1's clock
        y: analog;      -- P2's clock
        k: discrete;    -- whose turn it is (has values 0,1,2)
        a,              -- max delay time to register intent
        b: parameter;   -- min time to delay before rechecking
```

```
automaton p1
synclabs : ; initially loc_1;
loc loc_1: while True wait {dx in [4/5,1]}
        when k=0 do {x' = 0} goto loc_2;
loc loc_2: while x<=a wait {dx in [4/5,1]}
        when True do {k' = 1,x' = 0} goto loc_3;
loc loc_3: while True wait {dx in [4/5,1]}
        when x>=b & k=1 goto cs;
        when x>=b & k=0 goto loc_1;
        when x>=b & k=2 goto loc_1;
loc cs: while True wait {dx in [4/5,1]}
        when True do {k' = 0} goto loc_1;
end

automaton p2
synclabs : ; initially loc_1;
loc loc_1: while True wait {dy in [1,11/10]}
        when k=0 do {y' = 0} goto loc_2;
loc loc_2: while y<=a wait {dy in [1,11/10]}
        when True do {k' = 2, y' = 0} goto loc_3;
loc loc_3: while True wait {dy in [1,11/10]}
        when y>=b & k=2 goto cs;
        when y>=b & k=0 goto loc_1;
        when y>=b & k=1 goto loc_1;
loc cs: while True wait {dy in [1,11/10]}
        when True do {k' = 0} goto loc_1;
end

var init_reg, final_reg : region;

init_reg := loc[p1] = loc_1 & loc[p2] = loc_1 & k=0;
final_reg := loc[p1] = cs & loc[p2] = cs ;

prints "Condition for faulty system";
print omit all locations hide non_parameters in
      reach forward from init_reg endreach & final_reg endhide;
```

A.2.2 Output file — parametric analysis

```
Command: hytech ../examples/mutex/fish2
=================================================================
HyTech: symbolic model checker for embedded systems
Version 1.02b 3/21/96
For more info:
    email: hytech@eecs.berkeley.edu
    http://www.eecs.berkeley.edu/~tah/HyTech
Warning: Input has changed from version 1.00(a). Use -i for more info
=================================================================

Number of iterations required for reachability: 12
Condition for faulty system
     11a >= 8b   & a >= 0


=================================================================
Max memory used =   282 pages =  1155072 bytes =   1.10 MB
Time spent      =       1.42u +      0.04s =       1.46 sec total
=================================================================
```

A.2.3 Input file — error-trace generation

```
-- Fischer mutual exclusion example
-- error trace generation

define(delay_a,5)       -- max delay time to register intent
define(delay_b,6)       -- min time to delay before rechecking

var
        x,              -- P1's clock
        y               -- P2's clock
        : analog;
        k: discrete;    -- whose turn it is (has values 0,1,2)

 -- -------------------------------------------------------------- *)

automaton p1
synclabs : start_1, set_k_1, enter_cs_1, reset_1;

initially loc_1 & True;

loc loc_1: while True wait {dx in [4/5,1]}
        when k=0 do {x' = 0} sync start_1 goto loc_2;

loc loc_2: while x<=delay_a wait {dx in [4/5,1]}
        when True do {k' = 1,x' = 0} sync set_k_1 goto loc_3;

loc loc_3: while True wait {dx in [4/5,1]}
        when x>=delay_b & k=1 sync enter_cs_1 goto cs;
        -- two failed attempts
        when x>=delay_b & k=0 sync reset_1 goto loc_1;
        when x>=delay_b & k=2 sync reset_1 goto loc_1;

loc cs: while True wait {dx in [4/5,1]}
        when True do {k' = 0} sync reset_1 goto loc_1;
end


automaton p2
synclabs : start_2, set_k_2, enter_cs_2, reset_2;

initially loc_1 & True;

loc loc_1: while True wait {dy in [1,11/10]}
        when k=0 do {y' = 0} sync start_2 goto loc_2;

loc loc_2: while y<=delay_a wait {dy in [1,11/10]}
        when True do {k' = 2, y' = 0} sync set_k_2 goto loc_3;

loc loc_3: while True wait {dy in [1,11/10]}
        when y>=delay_b & k=2 sync enter_cs_2 goto cs;
        -- two failed attempts
        when y>=delay_b & k=0 sync reset_2 goto loc_1;
        when y>=delay_b & k=1 sync reset_2 goto loc_1;

loc cs: while True wait {dy in [1,11/10]}
        nwhen True do {k' = 0} sync reset_2 goto loc_1;
end
```

```
--

var
   init_reg, final_reg, reached, reached_viol : region;

init_reg := loc[p1] = loc_1 & loc[p2] = loc_1 & k=0;

final_reg := loc[p1] = cs & loc[p2] = cs;

reached := reach forward from init_reg endreach;

reached_viol := reached & final_reg;

if empty(reached_viol)
  then prints "Mutual exclusion requirement holds";
  else prints "Mutual exclusion violated";
       print trace to final_reg using reached;
endif;
```

A.2.4 Output file — error-trace generation

```
Command: hytech ../examples/mutex/fish2-e
=================================================================
HyTech: symbolic model checker for embedded systems
Version 1.02b 3/21/96
For more info:
    email: hytech@eecs.berkeley.edu
    http://www.eecs.berkeley.edu/~tah/HyTech
Warning: Input has changed from version 1.00(a). Use -i for more info
=================================================================

Number of iterations required for reachability: 16
Mutual exclusion violated
 ====== Generating trace to specified target region ========
Time: 0.00
Location: loc_1.loc_1
     x + 5 = 0   & y = 0   & k = 0
-------------------------------
  VIA: start_2
-------------------------------
Time: 0.00
Location: loc_1.loc_2
     x + 5 = 0   & y = 0   & k = 0
---------------
 VIA 5.00 time units
---------------
Time: 5.00
Location: loc_1.loc_2
     x = 0   & y = 5   & k = 0
-------------------------------
  VIA: start_1
-------------------------------
Time: 5.00
Location: loc_2.loc_2
```

```
     x = 0   & y = 5   & k = 0
-------------------------------
  VIA: set_k_2
-------------------------------
Time: 5.00
Location: loc_2.loc_3
     x = 0   & y = 0   & k = 2
---------------
 VIA 5.45 time units
---------------
Time: 10.45
Location: loc_2.loc_3
     11x = 48   & y = 6   & k = 2
-------------------------------
  VIA: enter_cs_2
-------------------------------
Time: 10.45
Location: loc_2.cs
     11x = 48   & y = 6   & k = 2
-------------------------------
  VIA: set_k_1
-------------------------------
Time: 10.45
Location: loc_3.cs
     x = 0   & y = 6   & k = 1
---------------
 VIA 6.00 time units
---------------
Time: 16.45
Location: loc_3.cs
     x = 6   & y = 12   & k = 1
-------------------------------
  VIA: enter_cs_1
-------------------------------
Time: 16.45
Location: cs.cs
     x = 6   & y = 12   & k = 1
 ============ End of trace generation ============


=================================================================
Max memory used =   278 pages =  1138688 bytes =   1.09 MB
Time spent      =       2.11u +      0.07s =       2.18 sec total
=================================================================
```

A.3 Corbett's distributed controller

```
-- dCorbetts distributed control system
--
-- Computes maximal delay between output control signals

var
        y1,              -- stopwatch for sensor_1
        y2,              -- stopwatch for sensor_2
        x1,              -- dschedulers stopwatch for delay on sensor 1
        x2,              -- dschedulers stopwatch for delay on sensor 2
        z                -- dcontrollers stopwatch
        : stopwatch;
        c                -- to measure maximal time delay
        : clock;
        alpha
        : parameter;

 -- -------------------------------------------------------------- *)

automaton sensor_1
synclabs : request_1, read_1, send_1, ack_1;

initially idle & y1=6;

loc idle: while y1<=6 wait {dy1 = 1}
        when y1>=6 sync request_1 goto read;

loc read: while True wait {dy1 = 0}
        when True sync read_1 do {y1'=0} goto wait;

loc wait: while y1<=4 wait {dy1=1}
        when y1>=4 sync request_1 goto read;
        when asap sync send_1 goto send;

loc send: while True wait {dy1=0}
        when True do {y1' = 0} sync ack_1 goto idle;
end

-- -------------------------------------------------

automaton sensor_2
synclabs : request_2, read_2, send_2, ack_2;

initially idle & y2=6;

loc idle: while y2<=6 wait {dy2 = 1}
        when y2>=6 sync request_2 goto read;

loc read: while True wait {dy2 = 0}
        when True do {y2'=0} sync read_2 goto wait;

loc wait: while y2<=8 wait {dy2=1}
        when y2>=8 sync request_2 goto read;
        when asap sync send_2 goto send;

loc send: while True wait {dy2=0}
        when True do {y2'= 0} sync ack_2 goto idle;
end

-- -------------------------------------------------

automaton scheduler
synclabs : request_1, read_1, request_2, read_2;

initially idle;

loc idle: while True wait {dx1=0,dx2=0}
        when True  sync request_1 do {x1' = 0} goto sensor1;
        when True  sync request_2 do {x2' = 0} goto sensor2;
```

```
loc sensor1: while x1<=11/10 wait {dx1=1,dx2=0}
        when x1>=1/2 sync read_1 goto idle;
        when True sync request_2 do {x2'=0} goto sensor2_wait1;

loc sensor2: while x2<=2 wait {dx2=1,dx1=0}
        when x2>=3/2 sync read_2 goto idle;
        when True sync request_1 do {x1'=0} goto sensor2_wait1;

loc sensor2_wait1: while x2<=2 wait {dx2=1,dx1=0}
        when x2>=3/2 sync read_2 goto sensor1;

end

 -- -------------------------------------------------------------

automaton controller
synclabs :      send_1, expire_1, ack_1,
                send_2, expire_2, ack_2,
                signal;

initially rest & c=0;

loc rest: while True wait {dz=0}
        when True sync send_1 do {z'=0} goto rec_1;
        when True sync send_2 do {z'=0} goto rec_2;

loc rec_1: while z<=1 wait {dz=1}
        when z>=9/10 sync ack_1 goto wait_2;

loc wait_2: while z<=10 wait {dz=1}
        when z>=10 sync expire_2 goto rest;
        when True sync send_2 do {z'=0} goto rec_12;

loc rec_12: while z<=1 wait {dz=1}
        when z>=9/10 sync ack_2 do {z'=0} goto compute;

loc rec_2: while z<=1 wait {dz=1}
        when z>=9/10 sync ack_2 goto wait_1;

loc wait_1: while z<=10 wait {dz=1}
        when z>=10 sync expire_1 goto rest;
        when True sync send_1 do {z'=0} goto rec_21;

loc rec_21: while z<=1 wait {dz=1}
        when z>=9/10 sync ack_1 do {z'=0} goto compute;

loc compute: while z<=56/10 wait {dz=1}
        when z>=36/10 sync signal do {c'=0} goto rest;

end

 -- -------------------------------------------------------------

print omit all locations
        hide non_parameters in
                reach forward from
                        loc[sensor_1] = idle & y1=6 &
                loc[sensor_2] = idle & y2=6 &
                loc[scheduler] = idle &
                loc[controller] = rest & c=0
        endreach
        & c >= alpha
endhide;
```

Appendix B Grammar

HyTech input is described by the following grammar. Non-terminals appear within angled parentheses. A non-terminal followed by two colons is defined by the list of immediately following non-blank lines, each of which represents a legal expansion. The metasymbol | is used at the beginning of lines to introduce alternative expansions. Input characters of terminals appear in typewriter font. The metasymbol λ denotes the empty string.

```
⟨hytech_input⟩::
 ⟨automata_descriptions⟩ ⟨commands⟩
```

We define each of these two components in the next two subsections.

B.1 Automata descriptions

B.1.1 Main definitions

```
⟨automata_descriptions⟩::
 ⟨declarations⟩ ⟨automata⟩

⟨declarations⟩::
 var ⟨var_lists⟩

⟨var_lists⟩::
 ⟨var_list⟩ : ⟨var_type⟩ ; ⟨var_lists⟩
| λ

⟨var_list⟩::
 ⟨name⟩
| ⟨name⟩ , ⟨var_list⟩

⟨var_type⟩::
 integrator
| stopwatch
| clock
| analog
| parameter
| discrete

⟨automata⟩::
 ⟨automaton⟩ ⟨automata⟩
| ⟨automaton⟩

⟨automaton⟩::
 automaton ⟨automaton_name⟩ ⟨prolog⟩ ⟨locations⟩ end

⟨prolog⟩::
 ⟨initialization⟩ ⟨sync_labels⟩
| ⟨sync_labels⟩ ⟨initialization⟩

⟨initialization⟩::
 initially ⟨name⟩ ⟨state_initialization⟩ ;
```

```
⟨state_initialization⟩::
 & ⟨convex_predicate⟩
| λ

⟨sync_labels⟩::
 synclabs : ⟨sync_var_list⟩ ;

⟨sync_var_list⟩::
 ⟨sync_var_nonempty_list⟩
| λ

⟨sync_var_nonempty_list⟩::
 ⟨name⟩ , ⟨sync_var_nonempty_list⟩
| ⟨name⟩

⟨locations⟩::
 ⟨location⟩ ⟨locations⟩

⟨location⟩::
 loc ⟨location_name⟩ :  while ⟨convex_predicate⟩ wait { ⟨rate_info_list⟩ } ⟨transitions⟩

⟨rate_info_list⟩::
 ⟨rate_info_nonempty_list⟩
| λ

⟨rate_info_nonempty_list⟩::
 ⟨rate_info⟩ , ⟨rate_info_nonempty_list⟩
| ⟨rate_info⟩

⟨rate_info⟩::
| ⟨rate_linear_expression⟩ in [ ⟨rational⟩ , ⟨rational⟩ ]
 ⟨rate_linear_expression⟩ ⟨rate_relop⟩ ⟨rate_linear_expression⟩

⟨transitions⟩::
 ⟨transition⟩ ⟨transitions⟩
| λ

⟨transition⟩::
 when ⟨convex_predicate⟩ ⟨update_synchronization⟩ goto ⟨location_name⟩ ;

⟨update_synchronization⟩::
 ⟨updates⟩ ⟨syn_label⟩
| ⟨syn_label⟩ ⟨updates⟩

⟨updates⟩::
 do { ⟨update_list⟩ }
| λ

⟨update_list⟩::
 ⟨update_nonempty_list⟩
| λ
```

```
⟨update_nonempty_list⟩::
  ⟨update⟩ , ⟨update_nonempty_list⟩
| ⟨update⟩

⟨update⟩::
  ⟨update_linear_expression⟩ ⟨relop⟩ ⟨update_linear_expression⟩

⟨syn_label⟩::
  sync ⟨name⟩
| λ
```

B.1.2 Rationals, linear terms, linear constraints and convex predicates

```
⟨convex_predicate⟩::
  ⟨linear_constraint⟩ & ⟨convex_predicate⟩
| ⟨linear_constraint⟩

⟨linear_constraint⟩::
  ⟨linear_expression⟩ ⟨relop⟩ ⟨linear_expression⟩
| True
| False
| asap

⟨relop⟩::
  <
| <=
| =
| >=
| >

⟨rate_relop⟩::
  <=
| =
| >=

⟨linear_expression⟩::
  ⟨linear_term⟩
| ⟨linear_expression⟩ + ⟨linear_term⟩
| ⟨linear_expression⟩ - ⟨linear_term⟩

⟨linear_term⟩::
  ⟨rational⟩
| ⟨rational⟩ ⟨name⟩
| ⟨name⟩

⟨update_linear_expression⟩::
  ⟨update_linear_term⟩
| ⟨update_linear_expression⟩ + ⟨update_linear_term⟩
| ⟨update_linear_expression⟩ - ⟨update_linear_term⟩
```

```
⟨update_linear_term⟩::
  ⟨rational⟩
| ⟨rational⟩ ⟨name⟩
| ⟨rational⟩ ⟨name⟩'
| ⟨name⟩
| ⟨name⟩'

⟨rate_linear_expression⟩::
  ⟨rate_linear_term⟩
| ⟨rate_linear_expression⟩ + ⟨rate_linear_term⟩
| ⟨rate_linear_expression⟩ - ⟨rate_linear_term⟩

⟨rate_linear_term⟩::
  ⟨rational⟩
| ⟨rational⟩ d⟨name⟩
| d⟨name⟩

⟨rational⟩::
  ⟨integer⟩
| ⟨integer⟩ / ⟨pos_integer⟩

⟨integer⟩::
  ⟨pos_integer⟩
| - ⟨pos_integer⟩
```

Notes

1. All names are case-sensitive strings beginning with a letter, and containing letters, digits, and the underscore character.
2. Positive integers (⟨*pos_integer*⟩) are sequences of digits.

B.2 Analysis commands

```
⟨commands⟩::
  ⟨region_declaration⟩ ⟨statements⟩

⟨region_declaration⟩ ::
  var ⟨region_declaration_list⟩ :  region;
| λ

⟨region_declaration_list⟩::
  ⟨region_name⟩ ⟨rest_of_decl_list⟩

⟨rest_of_decl_list⟩::
  , ⟨region_declaration_list⟩
| λ

⟨statements⟩::
  ⟨statement⟩ ; ⟨statements⟩
| λ
```

```
⟨statement⟩::
 free ⟨region_name⟩
| print ⟨region_expression⟩
| prints ⟨string⟩
| printsize ⟨region_name⟩
| print omit ⟨hide_loc_info⟩ ⟨region_expression⟩
| print trace to ⟨region_expression⟩ using ⟨region_name⟩
| ⟨region_name⟩ := ⟨region_expression⟩
| if ⟨bool_expression⟩ then ⟨statements⟩ endif
| if ⟨bool_expression⟩ then ⟨statements⟩ else ⟨statements⟩ endif
| while ⟨bool_expression⟩ do ⟨statements⟩ endwhile

⟨region_expression⟩::
 ⟨region_name⟩
| ⟨state_predicate⟩
| ( ⟨region_expression⟩ )
| ~⟨region_expression⟩
| ⟨region_expression⟩ | ⟨region_expression⟩
| ⟨region_expression⟩ & ⟨region_expression⟩
| diff ( ⟨region_expression⟩ , ⟨region_expression⟩)
| pre ( ⟨region_expression⟩ )
| post ( ⟨region_expression⟩ )
| hull ( ⟨region_expression⟩ )
| reach forward from ⟨region_expression⟩ endreach
| reach backward from ⟨region_expression⟩ endreach
| hide ⟨hide_var_info⟩ in ⟨region_expression⟩ endhide
| iterate ⟨region_name⟩ from ⟨region_expression⟩ using { statements }
| weakdiff ( ⟨region_expression⟩ , ⟨region_expression⟩)

⟨bool_expression⟩::
 empty ( ⟨region_expression⟩ )
| not ⟨bool_expression⟩
| ⟨bool_expression⟩ and ⟨bool_expression⟩
| ⟨bool_expression⟩ or ⟨bool_expression⟩
| ⟨region_expression⟩ < ⟨region_expression⟩
| ⟨region_expression⟩ <= ⟨region_expression⟩
| ⟨region_expression⟩ = ⟨region_expression⟩
| ⟨region_expression⟩ >= ⟨region_expression⟩
| ⟨region_expression⟩ > ⟨region_expression⟩
| ⟨region_expression⟩ weakle ⟨region_expression⟩
| ⟨region_expression⟩ weakeq ⟨region_expression⟩
| ⟨region_expression⟩ weakge ⟨region_expression⟩

⟨hide_loc_info⟩::
 all locations
| ⟨component_list⟩ locations

⟨hide_var_info⟩::
 all
| non_parameters
| ⟨variable_list⟩
```

```
⟨component_list⟩::
  ⟨component_name⟩ , ⟨component_list⟩
| ⟨component_name⟩

⟨variable_list⟩::
  ⟨variable_name⟩ , ⟨variable_list⟩
| ⟨variable_name⟩

⟨state_predicate⟩::
  ⟨state_predicate⟩ | ⟨state_predicate⟩
| ⟨state_predicate⟩ & ⟨state_predicate⟩
| loc [ ⟨automaton_name⟩ ] = ⟨location_name⟩
| ⟨convex_predicate⟩
| True
| False
```

In boolean expressions, the unary operator `not` has highest priority, followed by the infix binary operator `and`, then `or`. In region expressions, the operator `~` has highest priority, followed by `&`, and then `|`. Expressions are evaluated bottom-up from the left.

B.3 Reserved words

The following words are keywords and cannot be used as names for automata, synchronization labels, locations, or regions. Some of these are not described above, but are retained from earlier versions of the input language for backward compatibility.

`all`, `analog`, `and`, `asap`, `automaton`, `backward`, `clock`, `diff`, `direction`, `discrete`, `do`, `eliminate_non_parameters`, `eliminate_variables`, `eliminate_all_locations`, `eliminate_locations`, `else`, `end`, `endhide`, `endif`, `endreach`, `endwhile`, `empty`, `forward`, `False`, `final`, `free`, `from`, `goto`, `hide`, `hull`, `if`, `in`, `inf`, `initially`, `integrator`, `iterate`, `stopwatch`, `loc`, `locations`, `non_parameters`, `not`, `omit`, `or`, `parameter`, `post`, `pre`, `print`, `prints`, `printsize`, `reach`, `region`, `stopwatch`, `sync`, `synclabs`, `then`, `to`, `trace`, `True`, `using`, `var`, `vars`, `wait`, `weakdiff`, `weakeq`, `weakge`, `weakle`, `when`, `while`