Approaches to Verification of Hybrid Systems: A Survey

CS2108H Final Project

B. Devereux

March 15, 2000

Abstract

Hybrid systems are characterized by a mix of analog and digital inputs and outputs. We define such systems formally as hybrid automata, and discuss how to specify interesting properties of them. A few approaches to verifying hybrid systems are outlined: reachability analysis, model-checking on finite-state approximating automata, theorem proving, and linear programming. These approaches work with hybrid automata, or some other formalism that can be converted into hybrid automata. The verification methods are compared on a number of practical grounds, including what sort of information they provide from failed verification runs and how they distinguish between system and environment variables in the system.

1 Introduction

Informally, a *hybrid system* is characterized by a mixture of digital and analog inputs and outputs. Most often, hybrid systems have analog inputs, usually from various kinds of sensors; internal digital control; and analog outputs. Common real-world systems that are modelled as hybrid systems are thermostats, water-level controllers, and steam-boiler controllers; other applications, less useful to industry but still interesting, include descriptions of simple physical systems (a billiard ball on a table) and of ecosystems.

The study of hybrid systems is an area of overlap between disciplines, mostly computer science and control systems. Computer scientists tend to model hybrid systems as finite automata, plus real-valued variables. Conversely, control-systems experts tend to view them as dynamical systems, with boolean variables as well as real ones. A background in dynamical systems is not assumed for the purpose of this survey, so the computer-science approach will be taken.

Beginning with an extremely brief description of dynamical systems, we will proceed to the syntax and semantics of hybrid automata, probably the most widely-used model for hybrid systems. A few tractable subclasses are pointed out, and one operation – parallel composition – is defined. Some of the major verification techniques are outlined in brief: to wit, reachability analysis (symbolic model-checking) as used in HyTech, approximating automata, theorem proving, and mathematical programming. At the end are a few very short pointers to other techniques.

2 Definitions

In this section, we briefly define dynamical systems, and then introduce the syntax and semantics of hybrid automata. We distinguish between linear and nonlinear hybrid automata, and define the operation of parallel composition.

2.1 Dynamical Systems

The field of dynamical systems is a huge one; a formal presentation can be found in [Str94], merely one of many.

A dynamical system consists of a set of variables V (normally real or complex-valued), and differential equations expressing the instantaneous rate of change of each variable at each point, in terms of the values of some or all of the other variables at that point. At any point in time, the values of the variables in V are a vector of dimension $|V|$; if they are real-valued, they form a vector ν in $\mathbb{R}^{|V|}$. A dynamical system is often, informally, referred to as a *flow*, since, given a point in $\mathbb{R}^{|V|}$ at time t, the constraints on the first derivative force the values to change, as time passes, in a continuous manner. A dynamical system is *linear* if every differential equation is linear in the system's variables.

A *trajectory* of a dynamical system is a curve in phase space which is everywhere tangential to the slope of the system. Distance along a trajectory represents evolution of the system from a given point on that trajectory as time passes.

For the purposes of hybrid automata, a dynamical system is often expressed as a conjunction of the differential equations: for instance $F = (\dot{x_1} = -Kx_1 \wedge \dot{x_2} = 1)$, where $\dot{x}$ is the first derivative of x. We say that a pair of vectors of real variables $(\nu, \dot{\nu})$ in $\mathbb{R}^{|V|} \times \mathbb{R}^{|V|}$ *satisfies* the flow F, if the first conjunct of F is satisfied by the substitution of ν_1 for x_1 and of $\dot{\nu}_1$ for $\dot{x_1}$, and so on, up to n. Expressing the flow as a conjunction also allows us to conjoin two flows, which will be needed in Section 2.4 to perform parallel composition of two hybrid automata. For convenience, though, we also assume that we can pull out a single conjunct expressing the flow of a single variable: using F above, $F(x_2) = 1$.

Most of the flows dealt with in verification of hybrid systems are linear. Exact analysis of nonlinear dynamical systems is extremely difficult at best, and usually impossible; this is the domain of *chaos theory.* Chaotic hybrid systems have been examined little or not at all in the literature.

2.2 Hybrid Automata

2.2.1 Example

A *hybrid automaton* is a finite-state automaton with a set of real-valued variables associated with it. In each state, the variables evolve continuously according to rules for that state. When some condition on the variables which allows an instantaneous state transition to take place is met, than a transition *may* occur; if this happens in conjunction with the violation of one of the conditions (called an *invariant*) that must be met while the system is in a given state, then a transition *must* occur.

A typical, simple hybrid automaton is the thermostat from [ACH+95], shown in Figure 1.

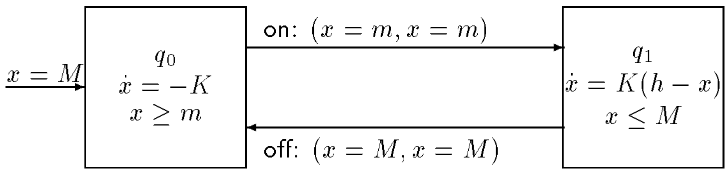

Figure 1: Thermostat automaton

This is a model for a simple thermostat in an indoor environment. In state q_0 the heater is off, and the temperature x decreases linearly, proportionally to the empirically-determined constant K. The heater stays off so long as the temperature exceeds the minimum m. When the temperature drops to m, the invariant for staying off ($x \geq m$) is violated, and the condition $x = m$ on the state transition labelled on is met; so that transition occurs, to q_1 where the heater is on.

In state q_1, the evolution of the temperature is also influenced by the heater constant h. The heater stays on as long as the temperature does not exceed the maximum M; as soon as it does, the thermostat switches the heater off again, returning to state q_0.

Next, the elements of a hybrid automaton are described formally piece by piece: at the end of the next section, each element is described by reference back to this thermostat example.

2.2.2 Syntax

Formalizations vary widely from paper to paper, even when the set of authors intersects. Sometimes an extremely general formalization is presented, as in [ACH+95] and [BM98], followed by the admission that such a general model is useless for practical purposes, and the presentation of a tractable specialization. Other papers, such as [HHWT98], [AH97], begin by presenting a formalization particular enough for their purposes. The formalization of this section is closely based on that of [HHWT98] and [HHWT97], with some minor changes to emphasize similiarities to, and differences from, classical finite automata.

A hybrid automaton is a tuple $H = (V, Q, \mathit{flow}, \mathit{switch}, \mathit{lab}, J, \Delta, \mathit{Inv}, \mathit{Init})$. To expand on the components:

Variables A finite ordered set $V = \{v_1, v_2, \ldots, v_n\}$ of variable symbols. The set of variable symbols $\dot{V}$, of the same cardinality, is also used, for the first derivatives of the variables. By convention, ν is used to denote a *valuation* of the variables; a vector $(\nu_1, \nu_2, \ldots, \nu_n)$ over the real numbers where $v_1 = \nu_1$ and so forth; likewise $\dot{\nu}$ is a valuation of $\dot{V}$.

Control modes A finite set Q of automaton states. In a classical automaton, they would simply be called states; however, that term is defined differently for hybrid automata.

States A *state* of a hybrid automaton is a triple (q, ν, ν') consisting of a control mode $q \in Q$, and valuations ν, ν' assigning the symbols in V to real values. ν is interpreted as the vector of current values of the variables, and ν' as the vector of current values of their first derivatives.

Flows Each control mode has an associated dynamical system over the variables in V. We will use $flow(q)$ to denote the entire dynamical system, and $flow(q, v)$ to denote the equation for $\dot{v}$ in control mode q.

Control switches The control switches $(q, q') \in switch$ corresponded to possible state transitions in classical automata; each is a possible change (under certain conditions) from one control mode q to another, q'. The set of control switches E is a multiset, so there may be several transitions possible from one mode to the same successor mode, governed by different conditions.

Transition labels Each possible transition is labelled by a symbol L from a set lab. Every transition (q, q) is labelled by the special stutter symbol τ.

Jump conditions A jump condition is a pair of predicates $(G, R) \in J$ with G over $V \cup \dot{V}$, and R over $V' \cup \dot{V'}$; intuitively, G expresses the conditions under which a control switch may be used to change states, and R the restrictions on the values of the variables after the change. $R(v)$ is the restriction on the value of a single variable v after the change.

Transitions Each transition $\delta \in \Delta$ is a triple (e, j, L), with e the control switch (start and end states of the transition), L the transition label, and j the jump condition which enables the control switch.

Invariants The invariant function $Inv(q)$ assigns to each control mode q a predicate over $V \cup \dot{V}$. The invariant must be true while the system is in the control mode; if it becomes untrue, then some transition must be taken. The choice of transition is nondeterministic, although the invariant in the successor control mode must hold for a transition to be taken.

Initial conditions The initial condition function $Init(q)$ assigns to each control mode q a predicate over $V \cup \dot{V}$, which expresses the possible valuations when the system begins in mode q. We assume that wherever $Init(q)$ holds, so does $Inv(q)$.

Final conditions Final conditions are *not* always used, since most hybrid systems in practice are intended to run indefinitely. They can be defined in the same manner as initial conditions, if needed. However, sometimes the violation of a safety condition is made a final condition of each control mode: that is, the system terminates if safety is violated [HHWT98].

In the thermostat from Figure 1, the set of variables $V = \{x\}$, the control modes $Q = \{q_0, q_1\}$, $flow(q_0) = (\dot{x} = -Kx)$, and $flow(q_1, x) = K(h - x)$. (q_0, q_1) is a control switch, on and off are transition labels, $Inv(q_1) = (x \leq m)$, $Init(q_0) = (x = M)$, and the jump condition associated with (q_0, q_1) is $(x = m, x = m)$. Note that this jump condition does not change the values of the system! The initial condition of q_0 is $x = M$, and that of q_1 is $false$, since the system never begins in q_1.

A hybrid automaton may begin its run in any state (q, ν) so long as ν satisfies $Init(q)$. The thermostat is designed to always begin in the state (q_0, M).

2.2.3 Semantics

The state $(q, \nu, \dot{\nu})$ of a hybrid automaton may change in two ways: *discretely*, involving a change of all three elements of the state to the new state $(q', \nu', \dot{\nu}')$ by means of some transition $((q, q'), (G, R), L)$ where $\nu, \dot{\nu}$ satisfy G and $\nu', \dot{\nu}'$ satisfy R; and *continuously*, in a single control mode, by means of a finite time-delay t, with the requirement that ν evolves into ν' by the passage of time t, and $\dot{\nu}$ and $\dot{\nu}'$ both satisfy $flow(q)$.

For a discrete state transition with label σ, we will write $s_1 \xrightarrow{\sigma} s_2$, and for a continuous transition by time t, $s_1 \xrightarrow{t} s_2$. The reflexive transitive closure of the transition relation will be written using $\Rightarrow$: $s \Rightarrow s'$ if $s \rightarrow s_2 \rightarrow \cdots \rightarrow s'$.

2.3 Linear and Nonlinear Hybrid Automata

The analysis methods described here work precisely only with *linear* hybrid automata: that is, those where the flow at each control mode is linear, and the invariants, initial conditions, and jump conditions are all *linear convex predicates* [HHWT97].

A linear convex predicate is a finite disjunction of linear inequalities; that is, inequalities between a rational constant and a linear combination of variables with rational coefficients. For instance, $(x \leq 3) \vee (y \leq 60)$ is a linear convex predicate.

Even linear hybrid automata are fairly intractable: the reachability problem is undecidable in the general case [ACH+95], even though most cases of interest do in fact yield to analysis. There are a few more tractable subclasses that are still useful, including the following:

Discrete automata If, for every control mode q and variable v, $flow(q, v) = 0$, then the automaton is discrete: variables only change to meet the constraint R of a jump condition. (Recall that this is the constraint on the values of the system *after* the transition.)

Timed automata A variable $v \in V$ is called a *clock* if, at every control mode q, $flow(q, v)$=1, and for all jump conditions (G, R), $R(v) = 0$ or x. It is called a *proposition* if $R(v) = 0$ or 1. A *timed automaton* is a linear hybrid automaton in which all the variables are either propositions or clocks, and the predicates (jump, initial, and invariant conditions) are boolean combinations of inequalities with the form xRc or $x-yRc$ where x and y are variables of the automaton, c is a nonnegative integer and R is a relational operator from the set $\{<, \leq, =, >, \geq\}$. For instance, reachability for timed automata is always decidable [ACH+95]. The tool KRONOS is used to verify timed automata [NSY92].

An example of a timed automaton is the time-delay automaton of Figure 2. This automaton has the clock variable t. In state q_0, the clock will increase until T time units have elapsed, when it will move to q_1 with the transition label done. In q_1, the automaton may, at any time, move back to the initial state with the transition label reset. Such a timed automaton can be composed with another automaton, synchronizing on the label done, to enforce a wait of T time units before a transition.

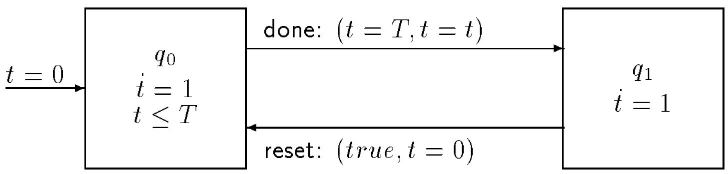

Figure 2: Time-delay automaton

2.4 Parallel Composition of Hybrid Automata

The parallel composition of two hybrid automata, H_1 and H_2, which share the same set of variables V, is a new automaton $H = H_1 \times H_2$ where

$$H = (Q_1 \times Q_2, V, flow, switch, lab_1 \cup lab_2, J, \Delta, Inv, Init)$$

where

Flows The flow at each product control mode (q_1, q_2) is $flow(q_1) \wedge flow(q_2)$. Note that in the arbitrary case, this will come out to *false*: for instance, if $\dot{x} = 3$ in q_1 and $\dot{x} = 4$ in q_2, then their conjunction will be *false*.

Transitions There is a transition $(((q_1, q_2), (q_1', q_2')), (G, R), L)$ in Δ if and only if three conditions hold:

1. $((q_1, q_1'), (G_1, R_1), L_1) \in \Delta_1$ and $((q_2, q_2'), (G_2, R_2), L_2) \in \Delta_2$
2. either $L_1 = L_2 = L$ (both transitions share the same label), or either $L_1 = L \in (L_1 \cup L_2)$ and $L_2 = \tau$ (the transition is labelled by a label from the first automaton which is not used by the second, and the second automaton has a stutter transition) or $L_2 = L \in (L_1 \cup L_2)$ and $L_1 = \tau$ (the transition is labelled by a label from the second automaton which is not used by the first, and the first automaton has a stutter transition).
3. $(G, R) = (G_1 \wedge G_2, R_1 \wedge R_2)$: the start and end conditions for the product transition are the conjunction (or intersection) of the jump conditions for each transition in the original automata.

The control switches, jump conditions, and transition labels are as determined by the product transitions.

Initial conditions $Init((q_1, q_2)) = Init(q_1) \wedge Init(q_2)$.

3 Verification

First we discuss what properties need to be verified for hybrid systems, and the means used to specify them. Following that, we describe in brief four methods of verification, and mention a few more.

3.1 What Is To Be Verified?

3.1.1 Safety Requirements

In almost all cases, what needs to be verified is a *safety requirement*: an assertion that some predicate over V always holds. Actually, the formulation of the property may originally include statements about the state of the system, as well; but the automaton can be augmented with extra variables so that the predicate need only be over V.

This is useful for analysis, because it allows us to consider the safety property as a subspace of the phase space, and apply techniques for proving that, from any permissible starting point, the system will always stay within that subspace (for a positive safety requirement) or never enter it (for a negative requirement).

Safety requirements can be expressed in a temporal logic such as Timed CTL (TCTL). TCTL has the normal syntax of CTL, as well as the reset quantifier, which introduces a clock variable (one that is *not* in V) and sets it to zero, with the following syntax:

$$z.\phi$$

where ϕ is a TCTL formula and z is the clock variable. The semantics is that $z.\phi$ is satisfied if ϕ is satisfied whenever $z = 0$ as the root of the computation tree, and $\dot{z} = 1$ throughout the computation.

3.1.2 Safety Requirements as Polytopes

A *polytope* is a finite region of n-dimensional space which is bounded by $(n-1)$-dimensional surfaces. A polygon (2-polytope) is a region of 2-space bounded by lines, and a polyhedron (3-polytope) is a region of 3-space bounded by planes.

If the safety requirements are a linear convex predicate, then they define a union of polytopes in the system's state space.

Timed CTL properties are also expressible as unions of polytopes.

3.1.3 Monitor Automata

Sometimes it is preferable to express the desired property as a "monitor automaton" which is composed with the system. The monitor automaton enters an "unsafe" state when the safety condition is violated. Model-checking is used to ensure that the composed system never enters the unsafe state.

3.1.4 Examples

For instance, with the thermostat, we may wish to verify the property that the heater is active for less than 2/3 of the first 60 minutes. This demands the augmentation of the automaton with two more variables: y, measuring the *total* elapsed time, and z, measuring time

spent in control mode q_0, in which the heater is on. The property to be verified is then:

$$y = 60 \wedge z \geq 2y/3$$

We can augment the automaton with y and z as in Figure 3, or compose it with the monitor automaton of Figure 4.

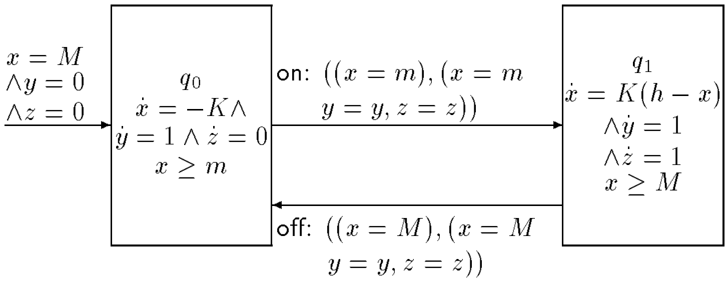

Figure 3: Thermostat automaton, augmented for verification

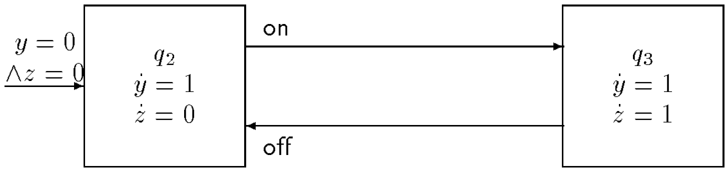

Figure 4: Monitor automaton

In this case, composition with the monitor will yield the same augmented automaton. However, we can create monitors for any number of properties without having to rewrite the thermostat automaton itself. Perhaps it is an out-of-the-box component that we don't wish to change, or cannot.

3.2 Reachability Analysis: Forward and Backward

In *forward reachability analysis* [ACH+95], we begin with the set of possible initial states of the hybrid automaton, and a finite union of convex polytopes, C, corresponding to the safety requirement. In the case of a positive safety requirement, we endeavour to show that every possible trajectory starting from the initial state space remains inside C; in the case of a negative requirement, we wish to prove that every possible trajectory remains outside C. In either case, the intersection of the set of reachable states with C is computed.

With *backward reachability analysis*, we begin with a region of state space, and compute backwards to find the initial state set it is reachable from. The intersection of this initial state set with that of the system being verified is computed.

Since the method is a state-space search, where the states consist of variable value assignments, reachability analysis is a form of symbolic model-checking.

In general, as stated above, the reachability problem is undecidable, but in practical cases it is often decidable. The model checker HyTech [HHWT97] performs such analysis. Actually, it does considerably more: HyTech incorporates a number of methods for approximating nonlinear hybrid automata by linear ones. It also produces error traces, and performs parametric analysis to determine appropriate values for unspecified parameters, such as h in the thermostat automaton. (K comes from an empirical measure of how fast heat is lost by the environment; we are not allowed to tweak it in the system, although of course K can be changed by adding insulation or opening a window.)

3.2.1 Approximating Automata

Highly intractable hybrid automata – very nonlinear ones which do not yield to any of the approximation techniques of the previous section – may be approximated by linear hybrid automata that can be shown to be in bisimulation with them.

Bisimulation for hybrid automata is defined in different ways: a *bisimulation relation* $\mathcal{R}$ is a subset of $Q \times Q$ with the properties that for all states p, q, if $q\mathcal{R}q'$, then if $p \xrightarrow{\sigma} p'$ (resp. $q \xrightarrow{\sigma}$), then there exists q' (resp. p') such that $q \xrightarrow{\sigma}$ (resp. $p \xrightarrow{\sigma} p'$) and $p'\mathcal{R}q'$. This is an equivalence relation [Bro99].

Another way of looking at a bisimulation [ACH+95] is as a *partition of the state space*; a set π of regions, whose union is equal to $\mathbb{R}^{|V|}$, with the property that for any two regions R, R' with $R \Rightarrow R'$, for every state $q \in R$, $\{q\} \Rightarrow R'$. This is the set of equivalence classes of a bisimulation relation.

Two methods of dealing with nonlinear automata, described in detail in [HHWT98], are *clock translation* and *linear phase-portrait approximation*. Both methods involving state-splitting (in this case, splitting elements of the finite set Q of control modes, not the infinite state space), in order to simplify flows and invariants to the point where techniques for solving nonlinear differential equations can be applied. In clock translation, variables that are solvable after state-splitting can be replaced by clock variables; in phase-portrait approximation, jump conditions, initial conditions, and flows are weakened (replaced by conditions implying the original condition), with the weakened automaton being linear.

Also, linear hybrid automata can be automatically converted into fully discrete finite automata that conservatively approximate the hybrid automaton. This approximation is simply a bisimulation that allows the invariants and jump conditions to be converted to a finite set of discrete threshold events. This is the approach of [CK99]. Conditions that allow a change of control modes to occur correspond to hyperplanes in the state space: taken together, all of these hyperplanes partition the space, and each threshold plane bounding a partition is a possible threshold event. Also, subsets of the state space where it is possible to continue indefinitely without leaving are identified: these are *null event sets* and correspond to sinks in the finite automaton.

The approximating automaton can be verified using any standard model-checking system. The principal benefit would seem to be the greater maturity, at the present time, of finite-automaton model checking technology. In [CK99], Chutinan and Krogh assert that it may be possible to use model checking on approximating automata to verify properties that are not decidable on linear hybrid automata.

Any approximating method, of course, comes with the caveat that a true safety property of the original system may be rendered false by the approximation; all a good method guarantees is that no false safety property will be made true.

3.3 Theorem Proving

In [LL96], a hybrid system was expressed as a timed automaton, and the safety properties expressed as theorems and proven by hand. Shortly afterward, Archer and Heitmeyer developed a set of tools for the theorem prover PVS [AH97] – theories about timed automata and templates for describing them within PVS – and used these tools to derive the results of [LL96] with a moderate amount of guidance.

In this approach, properties are encoded in first-order logic, rather than in a temporal logic; quantification is over reachable states of the automaton, rather than time steps or branches of a computation tree. Reasoning about functions of real numbers, which is necessary here, is difficult in general, but it was found that a subset of the full theory of real numbers could be used. Nondeterministic transitions were modelled using the choice operator on predicates, ϵ, which is a function from a predicate P to the range of the predicate. If the subset of the domain satisfying the predicate, $P^{-1}(true)$, is nonempty, then $\epsilon(P) \in P^{-1}(true)$. Otherwise $\epsilon(P) \in Dom(P)$, the domain of P. Although this restricts the range of valid conclusions that can be drawn more than using a transition relation, it turned out to be sufficient in the described case.

3.4 Linear Programming

The reachability analysis approach seen above takes advantage of the geometric interpretation of safety requirements as convex polytopes in the system's phase space. Some other approaches to verification are also essentially geometric. One such approach is the use of *linear programming*, a technique used in general to find value assignments to sets of variables under constraints. The theoretical basis of this approach is described in [BM98], and a case study is given in [BM99].

The method rests on a trick of converting logical propositions into constraints on variables. Each proposition is replaced by a *logical variable* $\delta_i \in \{0, 1\}$, and logical connective are replaced by constraints such as:

$$x_1 \vee x_2 \equiv \delta_1 + \delta_2 \geq 1$$

The informal argument for the validity of such a conversion proceeds by noting that $\delta_1 + \delta_2 \geq 1$ if and only if either is equal to 1.

We also wish some of these propositions to be inequalities on functions, such as $(f(x) \leq 0)$ over a given finite domain D. If M is defined as the local maximum of $f(x)$ on D, and m as the local minimum, a proposition like $(f(x) \leq 0) \wedge x_2$ becomes the inequality $f(x) - \delta_1 \leq -1 + m(1 - \delta_1)$.

In a hybrid automaton, there is a dynamical system associated with each control mode. Another way of viewing the situation is to view the automaton as a *single* dynamical system, where each differential equation governing the value of v_i is a case statement.

In the case of the thermostat:

$$\dot{x} = \begin{cases} -Kx & \text{if in } q_0 \\ K(h - x) & \text{if in } q_1 \end{cases}$$

and if a variable δ, which is 0 in q_0 and 1 in q_1, is introduced:

$$\dot{x} = (-Kx)(1 - \delta) + \delta(K(h - x))$$

Of course, to keep the semantics the same, $\dot{\delta}$ would also be needed. The variable δ is a *proposition*, in the terminology of timed automata [ACH+95]. The system obtained after this constraint is called a *mixed logical-dynamical system*.

This approach is, in a sense, the exact opposite of generating approximating finite automata: beginning with the hybrid automaton, transform it to a dynamical system with added variables to represent the control logic, and then reason about it using analysis methods for dynamical systems. Contrast this with generating an approximating automaton, which removes the dynamical considerations entirely and allows the verifier to use nothing but automata theory.

3.5 Other Approaches

There are a number of other approaches to verifying hybrid systems, including:

Duration calculus The duration calculus is a language for specifying real-time systems. In [YC99], the authors augmented the calculus with discrete control, and used a theorem-proving approach to prove properties about hybrid systems specified in the calculus.

Action systems Hybrid systems are specified in a version of Dijkstra's guarded command language which includes continuous evolution of variables in [RR99]. The specifications can be translated into HyTech.

Topology A purely topological and geometric approach is taken in [Bro99].

4 Comparisons

4.1 Distinction Between System And Environment

In the thermostat example, the constant K is derived from the environment, and the constant h is an operating parameter of the system. HyTech distinguishes between these two types of constant, but it seems as though the distinction is one that could easily be missed, and no simple, obvious decomposition into system and environment is possible.

The only approach where the desirability of modelling system and environment separately is [LL96], where an automaton represents the system and another represents the environment, and they are composed before verification. In nonlinear systems like the thermostat, though, it is far from clear how that would be done.

A tool based on linear programming could perform the same distinction between system and environment constants as HyTech, as indeed could any automata-based tool, but the problem of specifying, or deriving, precisely how the system has its desired effect on the environment, and what it would be in a different environment, does not seem to be addressed thus far by any method for describing and verifying hybrid systems.

It should be noted, though, that techniques for *synthesizing* hybrid systems to control a set of environment variables in a desired way, *do* address this issue; for instance, [TLS99] defines hybrid automata with state and input variables, something not done in papers only dealing with verification.

4.2 Choosing The Right Tool For The System

The appropriate model and tool to be used for verifying a hybrid system will depend on the nature of that system. Somewhat more hands-on experience would aid immeasurably in generating heuristics for this choice, but some brief observations follow.

What is to be verified might exist at several levels of abstraction: there could be a *circuit*, as in [CK99], a *real-time program*, or an *informal or semi-formal specification* of the system. Converting circuits to hybrid systems, which are then approximated by finite automata, is dealt with in [CK99]; certainly such a transformation ought to be theoretically possible in any formalism, but this is the only explicit treatment, so there is a good case for choosing that method.

Real-time programs can be quite simply – and probably automatically – converted to the action systems of [RR99], which can then be rendered into hybrid automata. However, it should not be too much more complicated to convert a real-time program directly into a hybrid automaton for verification, given some domain knowledge. Given the close relationship between flow of program control and state transitions in an automaton, this is most likely a better choice than a model such as the mixed logical-dynamical systems of [BM98], which emphasizes the dynamics.

The capacity for verifying an automaton with associated integer and real variables also suggests that a tool like HyTech could be used for abstract interpretation, even in cases with no real-time considerations.

In the case of a specification, which in general will be somewhat more abstract than a program, any of the approaches examined should work equally well; a formalism can be chosen based on the easiest way to express the specification formally.

4.3 Information From Failed Verification

An important issue in practical verification is the information that the tool provides when the verification fails. The quality of this information will have a strong impact on the time of the development process, since accurate diagnostics will allow the specification to be refined much more quickly.

Like model-checkers for finite automata, HyTech provides error traces, showing how a safety property can be violated. In the literature surveyed, this is the only example of such information being provided. Theorem proving methods ought, in principle, to provide not only specific counter-examples, but the precise proof obligation needed to make the property hold, which would then guide the next iteration of the specification. There is a brief mention of this in [AH97], but no detail.

Bisimulation checkers such as CCS [CPS93] are also capable of finding formulae that hold in one model and not in another; such a facility would be of considerable use for hybrid automata as well, but there is no evidence that it is implemented anywhere.

4.4 Adaptability to Existing Toolkit

Where cost of installation and learning curve are of considerable importance, there is some benefit to a verification tool that will make use of existing tools. An approximating-automata generator can be used with a finite-state model checker such as SMV or SPIN, and a mixed logical-dynamical system can be verified, where verification is possible, with any existing package or tool for linear programming. Similarly, the theorem-proving approach described makes use of experience with a standard theorem-prover; PVS, in this case.

Balanced against these advantages, a stand-alone hybrid system verifier such as HyTech or Kronos can offer more information from failed verifications, as discussed in the previous section, and in the future, potentially faster algorithms based on properties specific to hybrid automata.

5 Conclusions

The concept of hybrid systems was introduced informally, and then one specific formalism – hybrid automata – was described in detail. This formalism received the most attention because it has the most mature tool currently available, HyTech, associated with it, and more than one verification strategy. Mixed logical-dynamical systems were also described briefly.

Several verification methods were briefly described: reachability analysis on hybrid automata, generation of approximating automata, theorem proving, and linear programming on mixed logical-dynamic systems. A few others were described briefly.

The specification and verification methods were compared and criticized using a set of criteria: how they distinguish between system and environment, what sort of system (or system description) they are appropriate to, what kind of information is recoverable from a failed verification, and adaptability to an existing verification toolkit.

This is a relatively new area, and much of the extant work seems to be proofs-of-concept, with few case studies of applications in industry to be found. The most advanced tool currently available is HyTech, which seems set to become a *de facto* standard; several researchers, such as the author of [RR99], have created front ends to it independently.

Interesting avenues for development would be the development of hybrid programming methodology, with stepwise refinement, or the further application of techniques developed for hybrid systems to real-world systems of a hybrid nature, such as economies and ecologies. Also, it would be interesting to see the theorem-proving toolkit extended, since that may offer avenues for verification in the nonlinear case that a model-checker cannot pursue. A beginning of investigation into refinement of hybrid systems appears in [RS99].

References

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. Algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 3(34), 1995.

[AH97] Myla Archer and Constance Heitmeyer. Verifying hybrid systems modeled as timed automata: a case study. In *HART '97, Grenoble, France, March 26-28, 1997*, 1997.

[BM98] Alberto Bemporad and Manfred Morari. Control of systems integrating logic, dynamics, and constraints. Technical Report AUT-98-04, ETH - Swiss Federal Institute of Technology, 1998.

[BM99] A. Bemporad and M. Morari. Verification of hybrid systems via mathematical programming. In Frits W. Vaandrager and Jan H. van Schuppen, editors, *Hybrid Systems: Computation and Control*, number 1569 in Lecture Notes in Computer Science. Springer-Verlag, 1999.

[Bro99] Mireille Broucke. A geometric approach to bisimulation and verification of hybrid systems. In Frits W. Vaandrager and Jan H. van Schuppen, editors, *Hybrid Systems: Computation and Control*, number 1569 in Lecture Notes in Computer Science. Springer-Verlag, 1999.

[CK99] Alongkrit Chutinan and Bruce H. Krogh. Computing approximating automata for a class of linear hybrid systems. In Panos Antsaklis, Wolf Kohn, Michael Lemmon, Anil Nerode, and Shankar Sastry, editors, *Hybrid Systems V*, number 1567 in Lecture Notes in Computer Science. Springer-Verlag, 1999.

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

[HHWT97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HyTech: a model checker for hybrid systems. *Software Tools for Technology Transfer*, (1):pp. 110–122, 1997. Special issue on Timed and Hybrid Systems.

[HHWT98] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. Algorithmic analysis of nonlinear hybrid systems. *IEEE Transactions on Automatic Control*, (43):540–554, 1998. Special issue on Hybrid Control Systems.

[LL96] Gunter Leeb and Nancy Lynch. Proving safety properties of the steam boiler controller. In Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, editors, *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, number 1165 in Lecture Notes in Computer Science. Springer-Verlag, 1996.

[NSY92] X. Nicollin, J. Sifakis, and S. Yovine. Compiling real-time specifications into extended automata. *IEEE TSE Special Issue on Real-Time Systems*, 18(9):pp. 794–804, September 1992.

[RR99] M. Rönkkö and A. P. Ravn. Action systems with continous behaviour. In Panos Antsaklis, Wolf Kohn, Michael Lemmon, Anil Nerode, and Shankar Sastry, editors, *Hybrid Systems V*, number 1567 in Lecture Notes in Computer Science. Springer-Verlag, 1999.

[RS99] M. Rönkkö and K. Sere. Refinement and continous behaviour. In Frits W. Vaandrager and Jan H. van Schuppen, editors, *Hybrid Systems: Computation and Control*, number 1569 in Lecture Notes in Computer Science. Springer-Verlag, 1999.

[Str94] Steven H. Strogatz. *Nonlinear Dynamics and Chaos*. Addison-Wesley, 1994.

[TLS99] Claire Tomlin, John Lygeros, and Shankar Sastry. Computing controllers for nonlinear hybrid systems. In Frits W. Vaandrager and Jan H. van Schuppen, editors, *Hybrid Systems: Computation and Control*, number 1569 in Lecture Notes in Computer Science. Springer-Verlag, 1999.

[YC99] Xinyao Yu and Zongji Chen. Control of interval temporal systems. In Panos Antsaklis, Wolf Kohn, Michael Lemmon, Anil Nerode, and Shankar Sastry, editors, *Hybrid Systems V*, number 1567 in Lecture Notes in Computer Science. Springer-Verlag, 1999.