Hybrid cc, Hybrid Automata and Program Verification

(Extended Abstract)

Vineet Gupta * Radha Jagadeesan ** Vijay Saraswat *

1 Introduction

Synchronous programming. Discrete event driven systems [HP85,Ber89,Hal93] are systems that react with their environment at a rate controlled by the environment. Such systems can be quite complex, so for modular development and re-use considerations, a model of a composite system should be built up from models of the components compositionally. From a programming language standpoint, this modularity concern is addressed by the analysis underlying synchronous languages [BB91,Hal93,BG92,HCP91,GBGM91,Har87,CLM91,SJG95], (adapted to dense discrete domains in [BBG93]):

- Logical concurrency/parallelism plays a role in determinate reactive system programming analogous to the role of procedural abstraction in sequential programming — the role of matching program structure to the structure of the solution to the problem at hand.
- Preemption — the ability to stop a process in its tracks — is a fundamental programming tool for such systems [Ber93]. Examples of preemption include process suspension ("cntrl-Z") and process abortion("cntrl-C").
- The language should allow the expression of multiple notions of *logical time*, *i.e.* any signal can serve as a notion of time.

The design of synchronous programming languages has two distinct pieces: 1) A notion of defaults analyzed at the level of the basic (untimed) concurrent language. 2) The discrete timed synchronous language obtained by extending the untimed language uniformly over discrete time. Defaults allow the expression of the preemption constructs that rely on instantaneous detection of negative information. Furthermore, there are expressiveness advantages to be gained from building the programming language on top of a logic that allows expression of defaults — recent results [GKPS95] suggest that such programs can be exponentially more succinct than programs that do not admit expression of defaults.

* Xerox PARC, 3333 Coyote Hill Road, Palo Alto Ca 94304; `{vgupta,saraswat}@parc.xerox.com`

** Dept. of Mathematical Sciences,Loyola University-Lake Shore Campus, Chicago, Il 60626; `radha@math.luc.edu`

Hybrid cc. In earlier work[GJSB,GJSB95], we extended the analysis underlying synchronous programming to build an executable modeling and programming language for hybrid systems, Hybrid cc. Hybrid cc integrates conceptual frameworks for continuous and discrete change, as exemplified by the theory of differential equations and real analysis on the one hand, and the theory of programming languages on the other. As before, the language is built on top of a notion of defaults analyzed at the level of the basic (untimed) concurrent logic language. The hybrid programming synchronous language is obtained by extending the untimed language uniformly over continuous (real) time.

As in synchronous programming languages, various patterns of temporal activity can be defined in Hybrid cc — for example, the instantaneous preemption combinators of synchronous programming. In [GJSB], we provided precise operational semantics and described an interpreter that implements the operational semantics. In [GJSB95], we showed how to build and execute compositional models for various problems described in the literature. We also demonstrated that the process of building models was facilitated by a denotational semantics that allowed a more abstract view of programs as (temporal) constraints.

This paper. In this paper, we demonstrate the relationship of Hybrid cc to the methodology and tools developed in the research on verification of hybrid systems — for example, see earlier proceedings of this conference for a variety of approaches and tools [GNRR93,hyb95].

We aim to establish Hybrid cc as a high-level programming notation for hybrid automata — much as synchronous programming languages are high level notation for discrete automata. Concretely, we establish the expressiveness of Hybrid cc in two ways:

- For any given hybrid automaton, we describe a Hybrid cc program whose only traces are valid runs of the system.
- For any given safety property expressed in (real-time) temporal logic, we show how to write a Hybrid cc program that "detects" if the property is violated.

Furthermore, we aim to make programs written in Hybrid cc amenable to the tools developed for the verification of hybrid systems. For any Hybrid cc program, we build an automaton whose valid runs are precisely execution traces of the program.

2 Hybrid cc— The underlying computational intuition.

Hybrid cc is a language in the concurrent constraint programming framework, augmented with a notion of continuous time and defaults. In this section, we present an intuitive sketch of Hybrid cc — for a detailed formal development, we refer readers to [GJSB].

The (concurrent) constraint (cc) programming paradigm [Sar89] replaces the traditional notion of a store as a valuation of variables with the notion of a store as a constraint on the possible values of variables. Computation progresses by accumulating constraints in the store, and by checking whether the store entails constraints.

A salient aspect of the cc computation model is that programs may be thought of as imposing constraints on the evolution of the system. Default cc [SJG95] provides five basic constructs: (tell) a (for a a primitive constraint), parallel composition (A, B), positive ask (**if** a **then** A), negative ask (**if** a **else** A), and hiding (**new** X **in** A). The program a imposes the constraint a. The program (A, B) imposes the constraints of both A and B — logically, this is the conjunction of A and B. **new** X **in** A imposes the constraints of A, but hides the variable X from the other programs — logically, this can be thought of as a form of existential quantification. The program **if** a **then** A imposes the constraints of A provided that the rest of the system imposes the constraint a — logically, this can be thought of as intuitionist implication. The program **if** a **else** A imposes the constraints of A unless the rest of the system imposes the constraint a — logically, this can be thought of as a form of *defaults* [Rei80].

This declarative way of looking at programs is complemented by an operational view. The basic idea in the operational view is that of a network of programs interacting with a shared store of primitive constraints. The program a is viewed as adding a to the store instantaneously. The program (A, B) behaves like the simultaneous execution of both A and B. **new** X **in** A starts A but creates a new local variable X, so no information can be communicated on it outside. The program **if** a **then** A behaves like A if the current store entails a. The program **if** a **else** A behaves like A if the current store on quiescence does *not* entail a.

Continuous evolution over time. Hybrid cc is obtained by uniformly extending Default cc across real (continuous) time. This is accomplished by two technical developments.

First, we enrich the underlying notion of a constraint system to make it possible to describe the continuous evolution of state. Intuitively, we allow constraints expressing initial value (integration) problem, using tokens of the form **hence** d or d, where d is of the form $dot(X, m) = r$, for X a variable, m a non-negative integer and r a real number. The token $dot(X, m) = r$ states that the mth derivative of X is r and **hence** $dot(X, m) = r$ states that for all time $t > 0$ the mth derivative of X is r. From $X = 0,$ **hence** $dot(X, 0) = 1$, we can infer at time t that $X = t$. *Integration* operators $\int^r$, for every $r \in \mathbb{R}$ are generated by the entailment relation: intuitively, $\int^r c$ is the constraint entailed by c after evolving for r units of time. In this example, $\int^r c$ does correspond with to the usual integral. The technical innovation here is the presentation of a *generic* notion of continuous constraint system (ccs), which builds into the very general notion of constraint sytems just the extra structure needed to enable the definition of continuous control constructs (without committing to a *particular* choice of vocabulary for constraints involving continuous time). As a result subsequent development is *parametric* on the underlying constraint language: for each choice of a ccs we get a hybrid programming language.

Secondly, we add to the untimed Default cc a single temporal control construct: **hence** A. Logically, **hence** A imposes the constraints of A at every time instant after the current one. Operationally, if **hence** A is invoked at time t, a new copy of A is invoked at each instant in (t, ∞).

Agents	Propositions
a	a holds now
if a **then** A	if a holds now, then A holds now
if a **else** A	if a will not hold now, then A holds now
new X **in** A	exists an instance $A[t/X]$ that holds now
A, B	both A and B hold now
hence A	A holds at every instant after now

Intuitively, **hence** might appear to be a very specialized construct, since it requires repetition of the *same* program at every subsequent time instant. However, **hence** can combine in very powerful ways with positive and negative ask operations to yield rich patterns of temporal evolution. The key idea is that negative asks allow the instantaneous preemption of a program — hence, a program **hence if** a **else** A will in fact not execute A at all those time instants at which a is true.

Let us consider some concrete examples. First, clearly, one can program **always** A (which executes A at every time instant) by $(A, \textbf{hence } A)$. Now suppose that we require that a program A be executed at every time instant beyond the current one until the first time at which a is true. This can be expressed as **new** X **in** (**hence** (**if** X **else** A, **if** a **then always** X)). Intuitively, at every instant beyond the current one, the condition X is checked. Unless it holds, A is executed. X is local — the only way it can be generated is by the other program (**if** a **then always** X), which, in fact generates X continuously upon detecting a. Thus, a copy of A is executed at each time point beyond the current one upto (and excluding) the first time point at which a is detected. Similarly, to execute A precisely at the first time instant (assuming there is one) at which a holds, execute: **new** X **in hence** (**if** X **else if** a **then** A, **if** a **then hence** X). Analogously, one can define the following combinators that are characteristic of synchronous programming:

- Process abortion — **do** A **watching** a — execute the process A until the event a happens.
- Multiform time — **time** A **on** a — the process A runs only during the times a holds.

While conceptually simple to understand, **hence** A requires the execution of A at every subsequent real time instant. Such a powerful combinator may seem impossible to implement computationally. For example, it may be possible to express programs of the form **new** T **in** $(T = 0, \textbf{hence } dot(T) = 1, \textbf{hence if } rational(T) \textbf{ then } A)$ which require the execution of A at every rational $q > 0$. Such programs are not implementable — because rationals and irrationals are everywhere dense as a subset of the reals. We show that in fact

Hybrid cc is computationally realizable. The basic intuition we exploit is that, in general, physical systems change "slowly", with points of discontinuous change, followed by periods of continuous evolution. Technically, we introduce a *stability* condition for continuous constraint system that guarantees that for every constraint a and b there is a neighborhood around 0 in which a either entails or disentails b at every point. This rules out constraints such as `rational(T)` as inadmissible.

With this restriction, computation in Hybrid cc may be thought of as progressing in alternating phases of computation at a time point, and in an open interval. Computation at the time point establishes the constraint in effect at that instant, and sets up the program to execute subsequently. Computation in the succeeding open interval determines the length of the interval r and the constraint whose continuous evolution over $(0, r)$ describes the state of the system over $(0, r)$. At time r the program set up in the interval is executed to determine the point constraint, and so on.

3 Compiling Hybrid cc to Hybrid automata

We now present an algorithm to compile Hybrid cc programs into Hybrid cc automata. The Hybrid cc automata we compile to are variations of the hybrid automata presented in [ACH+95]. An automaton accepts traces from the environment, and determines if the traces are in the denotation of the Hybrid cc program that the automaton was constructed from.

Given a constraint system $\mathcal{C} = (D, \vdash)$, define `cont` $\mathcal{C} = (\{$`cont` $d \mid d \in D\}, \vdash'$) — a copy of $\mathcal{C}$ with all tokens prefixed by a *cont* to distinguish them from the tokens of $\mathcal{C}$. The inference relation on `cont` $\mathcal{C}$ is induced by $\vdash$; an integration relation on `cont` $\mathcal{C}$ is induced by the integration relation of the continuous constraint system built on $\mathcal{C}$. Now, define the constraint system $\mathcal{C}'$ as the cartesian product $\mathcal{C} \times$ `cont` $\mathcal{C}$ in the category of constraint systems.

A Hybrid cc automaton consists of:

- A set of states St. States are of two kinds, point states and interval states. Each state is labeled with a Default cc program on the constraint system $\mathcal{C}$, and each interval state is labeled with a constraint in $\mathcal{C}$. One of the point states is designated a start state.
- A set of transitions between states. Each transition is labeled by a constraint in $\mathcal{C}'$. Transitions can go from point states to interval states and vice versa, and also from an interval state to itself. The set of labels on the outgoing transitions from any state is closed under least upper bounds.

The input trace supplied to the automaton is a pair of functions (Q, C), where $Q : \mathbb{R}^{\geq 0} \rightarrow St, C : \mathbb{R}^{\geq 0} \rightarrow \mathcal{C}$ satisfying:

- Q, C are partial functions with the same domain; the domain is a prefix of the real line.

- Q is finitely variable — Q changes only finitely many times in any bounded interval. Thus for all t in the domain of Q, Q is constant over some succeeding open interval — we refer to this value of Q as $Q(t+)$. Also for all $t > 0$ in the domain of Q, Q is constant over some preceding open interval — we refer to this value of Q as $Q(t-)$. Both $Q(t-)$ and $Q(t+)$ are required to be interval states. Finally, if $Q(t-) \neq Q(t)$, we demand that $Q(t)$ be a point state.
- For each $t \geq 0.\exists \epsilon_t > 0$, C is smooth over $(t, t+\epsilon_t)$, *i.e.* there is a constraint $c \in \mathcal{C}'$ such that for all $t' \in (t, t+\epsilon_t).C(t') = \int^{t'-t} c$. The constraint c that establishes the smoothness of C beyond t is always of the form $(a_t, \texttt{cont}\ b_t)$, and we define $C(t+) = \texttt{cont}\ b_t$.

Let (C, Q) be a trace. The automaton accepts this trace if:

- $Q(0)$ is the start state of the automaton.
- For all t, $C(t)$ is a fixed point of the program in state $Q(t)$.
- For all t, the automaton takes the transition determined by $C(t), C(t+)$ — the transition with the greatest label below $C(t), C(t+)$ — to the interval state $Q(t+)$ (which may be the same as $Q(t)$, in which case nothing is done).
- For all $t > 0$, if $Q(t-) \neq Q(t)$, the automaton takes the transition out of $Q(t-)$ determined by $C(t), C(t+)$ — the transition with the greatest label below $C(t), C(t+)$ — from $Q(t-)$ to the point state $Q(t)$.

Examples of Hybrid cc automata are given below. In the third automaton, for example, the trace $C(0) = b, Q(0) = 1, C((0,3)) = (a,b,c), Q((0,3)) = 4$ is accepted, while the trace $C([0,3)) = (a,b), Q(0) = 1, Q((0-3)) = 4$ is not accepted, as (a,b) is not a fixed point of the program (b,c). (Note that no other values for Q would have worked either — Q is essentially unique for any C.)

Compilation algorithm. We now show how to compile a program P into an automaton which accepts only those traces which satisfy the constraints imposed by P. In order to compile a program P, we first put it in a normal form, and then compile the normal form.

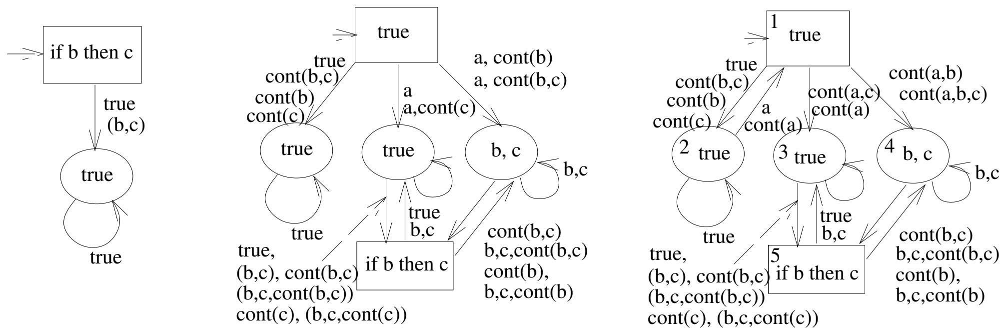
Normal form for Hybrid cc *programs.* A Hybrid cc program is in normal form if it can be written as the parallel composition of fragments generated as follows:

$$N ::= \textbf{if } a \textbf{ then if } b_1 \textbf{ else } \ldots \textbf{if } b_n \textbf{ else } b,\ n \geq 0$$
$$\mid \textbf{if } a \textbf{ then if } b_1 \textbf{ else } \ldots \textbf{if } b_n \textbf{ else hence } N,\ n \geq 0$$

The number of **hence** 's in a fragment is called the depth of the fragment. The program may be converted into such a normal form by repeated application of the following rewrite rules:

$$\begin{aligned}
\textbf{hence hence } A &\rightarrow \textbf{hence } A \\
\textbf{if } a \textbf{ else if } b \textbf{ then } A &\rightarrow \textbf{if } b \textbf{ then if } a \textbf{ else } A \\
\textbf{if } a \textbf{ then if } b \textbf{ then } A &\rightarrow \textbf{if } (a \sqcup b) \textbf{ then } A \\
\textbf{if } \mathit{true} \textbf{ then } A &\leftrightarrow A \\
\textbf{hence } (A, B) &\rightarrow \textbf{hence } A, \textbf{hence } B \\
\textbf{if } a \textbf{ then } (A, B) &\rightarrow \textbf{if } a \textbf{ then } A, \textbf{if } a \textbf{ then } B \\
\textbf{if } a \textbf{ else } (A, B) &\rightarrow \textbf{if } a \textbf{ else } A, \textbf{if } a \textbf{ else } B
\end{aligned}$$

Compilation of the Normal form. We first show how to compile each of the fragments, and a product construction will provide the final automaton. Compiling a fragment F is done inductively, based on the depth of the fragment. A dead state is defined as an interval state with the program `true` and a single transition labeled `true` going back to it. Examples of automata for programs of depth less than or equal to 1 are shown here — the general case follows immediately.



Program: if b then c — Program : if a then hence if b then c — Program : hence if a then hence if b then c

Some labels have been omitted to avoid cluttering. In the third automaton, the start state has been merged with the original start state.

If the depth is greater than one, then $F =$ **if** a **then if** b_1 **else** ...**if** b_n **else hence** F'. Let $F' =$ **if** a' **then if** b'_1 **else** ...**if** b'_m **else hence** .... We first construct the automaton for F' inductively. Now we convert this into the automaton for F as follows.

- Add arcs from the start state of F' with labels consisting of subsets of $\{$`cont` a', `cont` $b'_1, \ldots$ `cont` $b'_m\}$ (and their conjunctions with the other labels) — each going to the same state where the corresponding arc without `cont` went.
- If q is the dead state reached from the start state (on the transition say `true`), add from it arcs with labels from subsets of $\{a', b'_1, \ldots b'_m,$ `cont` a', `cont` b'_1, ..., `cont` $b'_m\}$. The arcs a', `cont` a' and $(a',$ `cont` $a')$ go to the start state, the others back to q.
- Add a new start state labeled `true`. The arcs from it are all the arcs from the (modified) start state of F', with their labels conjoined with a. All other arcs go to a new dead state.

We now combine the fragments to get the automaton for the entire program by performing a standard product construction. States are pairs of states in the automata for A and B, and are labeled by programs which are parallel compositions of the programs in the component states. A state is an interval state if an only if both its components are interval states, otherwise it is a point state. The start state is the joint start state of the component automata. Transitions are induced by pairs of transitions from the individual automata.

Hiding. Consider the program **wait** 5 **do** A = **new** X **in** ($X = 0$, **hence** ($dX/dt = 1$, **if** $X = 5$ **then** A)). This program starts A exactly 5 seconds after being started. This program has uncountably many "control points" (more precisely derivatives in the parlance of synchronous programming), one for each real in the interval $[0, 5]$. This makes a general compilation of the hiding combinator impossible. However, the above compilation algorithm can be extended to hiding of variables whose values do not evolve autonomously. The hidden variables used in the definition of the defined combinators and the hidden variables in the programs in this paper are of this kind. A more systematic semantic study remains to be undertaken.

Correctness of Automata. The correctness of our compilation is proved via the denotational semantics. Details are omitted for lack of space.

4 Testers for Hybrid Automata

In this section, we illustrate the expressive power of Hybrid cc relative to hybrid automata — we show that given a hybrid automaton, we can construct a Hybrid cc program accepting exactly the same traces as the automaton. For concreteness, we consider the hybrid automata presented in [ACH+95].

We recall the definition of hybrid automata from [ACH+95] for reference. A hybrid automaton is described by a 6-tuple:

- A finite set of locations, Loc.
- A finite set of real-valued variables Var. A valuation v is an assignment of real values to all variables in Var.
- A set of labels Lab.
- A finite set of transitions. Each transition consists of (l, a, μ, l'), where l, l' are the source and target locations, a is a label, and μ relates source valuations to target valuations.
- A function which labels each location with a set of *activities* — an activity is a function from $\mathbb{R}^{\geq 0}$ to valuations.
- A function assigning each location l_i with an invariant $Inv(l_i)$, which is a set of valuations.

Now a run of the automaton is defined as a finite or infinite sequence of 4-tuples: $\langle (l_0, v_0, f_0, t_0), (l_1, v_1, f_1, t_1), \ldots \rangle$ is a run if for all $i \geq 0$, f_i is an activity in l_i, $f_i(0) = v_i$, $\forall t, 0 \leq t < t_i \rightarrow f_i(t) \in Inv(l_i)$ and $(l_i, \_, \mu_i, l_{i+1})$ is a transition with $(f_i(t_i), v_{i+1}) \in \mu_i$.

Translation. Given a hybrid automaton, we show how to build an "equivalent" Hybrid cc program — more precisely, we build a Hybrid cc program whose observations are exactly the runs of the given hybrid system. It is helpful to view the Hybrid cc program that we build as an acceptor of the valid runs of the given hybrid automaton.

In order for us to be able to write the Hybrid cc program, we will need to be able to express as constraints various aspects of the given hybrid automaton. This will mean that our constraint system should be expressive enough. Also, if we desire a finite program, these aspects will have to be finitely representable. We summarize these representability conditions below:

- For each location l_i, the constraint $v \in Inv(l_i)$ must be expressible as a constraint in our constraint system.
- The set of activities in any location must be representable by the evolution of a set of constraints. For example, we could have the constraint $dx/dt \geq 3$ representing the set of all activities with the derivative of x greater than 3. What we require is a set of such constraints such that they exactly capture the activities at a location.
- In each transition, the relation μ must be representable as a constraint.

These conditions are not too strict — the examples of hybrid systems given in [ACH+95], these requirements are met by the simple constraint system described in [GJSB].

The program will be given as a set of Hybrid cc agents, each of which represents a location l_i. When the system enters a location l_i, the agent st_i is activated. The agent examines the initial valuations, and starts up a number of testers to check the run. Each tester keeps reporting `ok` as long as the run is in the allowed set of activities of that tester. Since it is initially not known which activity in the set is being taken, several testers are started in parallel, and as long as any one of them says `ok`, the run is allowed. Simultaneously, another tester keeps checking that the run does not violate the invariant of the location. The construct *abort* accepts nothing — if no tester reports `ok`, *abort* rejects the run.

The testers are written using the construct **while** a **do** A, which keeps on executing A while the constraint a is true. Thus for example, if we wanted to test for the set of all activities in which $dx/dt \geq 3$, then we would write **while** $(dx/dt \geq 3)$ **do always** *ok*, this would keep generating an `ok` while the constraint holds. If it stopped holding at some time, then no further `ok`'s would be generated.

These testers are kept running while the program counter `pc` indicates that the location is still l_i. As soon as it changes, the testers are killed. Now the second part of the agent is activated — **first** a **do** A starts A at the first instant (if any) that a holds. This part consists of one tester for each transition whose source is l_i. These check that the last valuation of the current location and the first valuation of the next location are in the relation μ for some transition, which also has the right target and the right label. (Labels may be omitted if desired.) If any check succeeds, then an `ok` is generated, and the agent for the next location is activated. The current location agent does nothing more.

```
st_i ::
new ok in [
while (pc = l_i) do [
```

```
     if (v in v_1) then while c_1 do always ok,
     if (v in v_2) then while c_2 do always ok,
     ...,
     always if ok else abort,
     always if (v in Inv(l_i)) else abort],
first (pc =/= l_i) do
     [if ((prev(v), v) in mu_1 and pc = l_k and label = a_1)
           then (ok, st_k),
      if ((prev(v), v) in mu_2 and pc = l_k' and label = a_2)
           then (ok, st_k'),
      ...,
      if ok else abort]].
```

The program for the system is now as follows. It just starts the appropriate location agent at the beginning of the run.

```
system::
    if pc = l_1 then st_1,
    if pc = l_2 then st_2,
        ... .
```

An important property of this translation is that the parallel composition of two hybrid automata is recognized by the parallel composition of the two testers for these automata — in particular, the synchronization of transitions is done as expected.

Now the various cases of hybrid automata can be described by testers over specific constraint systems. For example, a hybrid automaton is *linear* iff it can be tested by a tester over the constraint system in which for any variable dx/dt is a constant, and linear terms are allowed.

5 Testers for safety properties

We will now show how to write testers for real-time safety properties in Hybrid cc. The basic idea is that a tester for a safety property raises a flag whenever the safety property is violated. This section shows that Hybrid cc can serve as a language for description of properties in addition to serving as a language for the description of systems.

The particular logic that we consider to write safety properties is (the past properties of) integrator computation tree logic(ICTL) [AHH93]. The encoding itself is essentially a real-time extension of extant encodings of safety properties of discrete linear time temporal logics [MP91] in synchronous languages [JPVO95]. We will exploit the structuring facilities of the programming language to achieve modularity in the encoding; thus, the following translation is *compositional.*

The tester for a basic constraint c just checks if c is true now, and raises a flag if it is not. Note that the flag is not cumulative — so if c was false at some time in the past, it does not mean that the flag will remain raised now. However, also note that the tester for c detects violation of the safety property c at all points of time — it actually functions as a tester for $\Box c$.

To test $\neg A$, we build a tester for A. If A raises a flag in the current state, then we do nothing, otherwise a flag is raised. $A \vee B$ is tested by making sure that either one of A or B holds, if both raise flags, then the tester for $A \vee B$ raises a flag. The tester for A since B makes sure that whenever B becomes true (determined by a flag raised by the tester for $\neg B$), a process is started to check for A, and as soon as A becomes false, A since B becomes false and remains false forever. Finally $(z : p)A$ is tested by testing A, we simply use the **time** A **on** c construct of Hybrid cc to yield a compositional translation of the integrator variables of ICTL.

Property ϕ	Tester $T(\phi)$
c	**always if** c **else** $err(c)$
$\neg A$	$T(A)$, **always if** $err(A)$ **else** $err(\neg A)$
$A \vee B$	$T(A), T(B)$, **always if** $(err(A) \wedge err(B))$ **then** $err(A \vee B)$
A since B	$T(\neg B), T(A)$, **first** $err(\neg B)$ **do** **always if** $err(A)$ **then always** $err(A \text{ since } B)$
$(z : p)A$	$T(A), z = 0$, **time** (**always** $dz/dt = 1$) **on** p, **time** (**always** $dz/dt = 0$) **on** $not(p)$, **always if** $err(A)$ **then** $err((z : p)A)$.

Acknowledgements. Work on this paper has been supported in part by ONR through grants to Vijay Saraswat and to Radha Jagadeesan, and by NASA. Radha Jagadeesan has also been supported by a grant from NSF.

References

[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.

[AHH93] Rajeev Alur, T.A. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. In *Proceedings of the 14th Annual IEEE Real-time Systems Symposium*. IEEE, 1993.

[BB91] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. In *Special issue on Another Look at Real-time Systems*, Proceedings of the IEEE, September 1991.

[BBG93] A. Benveniste, M. Le Borgne, and Paul Le Guernic. *Hybrid Systems: The SIGNAL approach*. Number 736 in LNCS. Springer Verlag, 1993.

[Ber89] G. Berry. Real-time programming: General purpose or special-purpose languages. In G. Ritter, editor, *Information Processing 89*, pages 11 – 17. Elsevier Science Publishers B.V. (North Holland), 1989.

[Ber93] G. Berry. Preemption in concurrent systems. In *Proc. of FSTTCS*. Springer-Verlag, 1993. LNCS 781.

[BG92] G. Berry and G. Gonthier. The ESTEREL programming language: Design, semantics and implementation. *Science of Computer Programming*, 19(2):87 – 152, November 1992.

[CLM91] E. M. Clarke, D. E. Long, and K. L. McMillan. A language for compositional specification and verification of finite state hardware controllers. *Proceedings of the IEEE*, 79(9), September 1991.

[GBGM91] P. Le Guernic, M. Le Borgne, T. Gauthier, and C. Le Maire. Programming real time applications with SIGNAL. In *Special issue on Another Look at Real-time Systems*, Proceedings of the IEEE, September 1991.

[GJSB] Vineet Gupta, Radha Jagadeesan, Vijay Saraswat, and Daniel Bobrow. Computing with continuous change. Submitted to *Science of Computer Programming*.

[GJSB95] Vineet Gupta, Radha Jagadeesan, Vijay Saraswat, and Daniel Bobrow. Programming in hybrid constraint languages. In *Hybrid Systems II*, volume 999 of *Lecture notes in computer science*. Springer Verlag, November 1995.

[GKPS95] G. Gogic, H. Kautz, C. Papadimitriou, and B. Selman. The comparitive linguistics of knowledge representations. In *14th International joint conference on Artificial intelligence*, 1995.

[GNRR93] Robert Grossman, Anil Nerode, Anders Ravn, and Hans Rischel, editors. *Hybrid Systems*. Springer Verlag, 1993. LNCS 736.

[Hal93] N. Halbwachs. *Synchronous programming of reactive systems*. The Kluwer international series in Engineering and Computer Science. Kluwer Academic publishers, 1993.

[Har87] D. Harel. Statecharts: A visual approach to complex systems. *Science of Computer Programming*, 8:231 – 274, 1987.

[HCP91] N. Halbwachs, P. Caspi, and D. Pilaud. The synchronous programming language LUSTRE. In *Special issue on Another Look at Real-time Systems*, Proceedings of the IEEE, September 1991.

[HP85] David Harel and Amir Pnueli. *Logics and Models of Concurrent Systems*, volume 13, chapter On the development of reactive systems, pages 471–498. NATO Advanced Study Institute, 1985.

[hyb95] *Hybrid Systems II*. Springer Verlag, 1995. LNCS 999.

[JPVO95] L.J. Jagadeesan, C. Puchol, and J.E. Von Olnhausen. Safety property verification of ESTEREL programs and applications to telecommunications software. In *Proceedings of CAV*, LNCS 939, 1995.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1991. 427 pp.

[Rei80] Ray Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81 – 132, 1980.

[Sar89] Vijay A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, January 1989. To appear, Doctoral Dissertation Award and Logic Programming Series, MIT Press, 1992.

[SJG95] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Default Timed Concurrent Constraint Programming. In *Proceedings of Twenty Second ACM Symposium on Principles of Programming Languages, San Francisco*, January 1995.

This article was processed using the LaTeX macro package with LLNCS style