Hybrid Verification by Exploiting the Environment*

Limor Fix†
Fred B. Schneider

TR 94-1436
July 1994

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*Work supported in part by the Office of Naval Research under contract N00014-91-J-1219, The National Science Foundation under Grant No. CCR-9003440, DARPA/NSF Grant No. CCR-9014363, NASA/DARPA grant NAG-2-893, and AFOSR grant F49620-94-1-0198. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not reflect the views of these agencies.

†Limor Fix is also supported, in part, by a Fulbright post-doctoral award.
Hybrid Verification by Exploiting the Environment*

Limor Fix† Fred B. Schneider
Department of Computer Science
Cornell University
Ithaca, New York 14853

June 30, 1994

Abstract. A method for verifying hybrid systems is given. Such systems involve state components whose values are changed by continuous (physical) processes. The verification method is based on proving that only those executions that satisfy constraints imposed by an environment also satisfy the property of interest. A suitably expressive logic then allows the environment to model state components that are changed by physical processes.

1 Introduction

What executions of a concurrent program are possible and what properties are satisfied by that program may depend on the environment. Consider a system to maintain a given water level in a tank. Under computer control, a pump causes water to be added and a valve causes water to be drained. Correctness of the control program depends on the environment—in particular, on the rate at which the pump adds water and the rate at which the valve drains water. In fact, correctness of the control program is defined in terms of permissible states of the environment, because correctness is based on the water-level. One simply cannot specify or reason about such a control program without saying something about its environment.

In [10] we introduced two principles for verifying programs whose executions are affected by an environment. The state of the environments considered in [10] change discretely along with each atomic action of the program. Nevertheless, our principles were shown to be usable for verifying real-time behavior of concurrent programs, because schedulers and resource limitations that affect execution time can be regarded as part of the environment. In this paper, we extend those results to environments having variables that change value continuously, as time passes. The result is a new verification method for hybrid systems.

The remainder of this paper is structured as follows. In Section 2, we review the principles introduced in [10]. Section 3 presents a simple concurrent programming language, giving a plausible semantics for programs that will control physical processes. Our specification language is discussed in Section 4. Section 5 explains how invariance-based proof methods for verifying safety properties

---

*Work supported in part by the Office of Naval Research under contract N00014-91-J-1219, the National Science Foundation under Grant No. CCR-9003440, DARPA/NSF Grant No. CCR-9014363, NASA/DARPA grant NAG-2-893, and AFOSR grant F49620-94-1-0198. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not reflect the views of these agencies.

†Limor Fix is also supported, in part, by a Fulbright post-doctoral award.
can be used for verifying hybrid systems as well. Section 6 contains an example. And, Section 7 puts our work in context. A soundness proof of our verification method appears in an appendix.

2 Formalizing and Exploiting the Environment

Any method for program verification comprises: a programming language, a property language, and a way to prove that a program $P$ satisfies a property $\Phi$. Program $P$ and the property $\Phi$ define sets $[[P]]$ and $[[\Phi]]$ of behaviors, where a behavior is a mathematical object that describes an execution of the program. A program $P$ satisfies a property $\Phi$, denoted $(P, \Phi) \in Sat$, exactly when all behaviors of $P$ are permitted by $\Phi$:

$$ (P, \Phi) \in Sat \text{ if and only if } [[P]] \subseteq [[\Phi]] $$

The environment in which a program executes defines a property too. This property contains behaviors that are not precluded by one or another aspect of the environment. For example, with the water tank discussed above, the environment defines a property containing those behaviors where the water level changes continuously and only by amounts consistent with the pump’s rate and the valve’s rate. Behaviors in which the water level changes abruptly are not in this property.

For a property $E$ defined by an environment, the feasible behaviors of a program $P$ under $E$ are those behaviors of $P$ that are also in $E$: $[[P]] \cap [[E]]$. A program $P$ satisfies a property $\Phi$ under an environment $E$, denoted $(P, E, \Phi) \in ESat$, if and only if every feasible behavior of $P$ under $E$ is in $\Phi$:

$$ (P, E, \Phi) \in ESat \text{ if and only if } ([[P]] \cap [[E]]) \subseteq [[\Phi]] $$  \hspace{1cm} (1)

Based on simple set theory and (1), we also have

$$ (P, E, \Phi) \in ESat \text{ if and only if } [[P]] \subseteq [[\Phi]] \cup [[E]], $$ \hspace{1cm} (2)

where $[[E]]$ denotes the set complement of $[[E]]$.

3 Programs

Consider a simple programming language having an empty statement (skip), assignment ($:=)$, sequential composition (;), iteration (do), and parallel composition ($\parallel$). To simplify the exposition, we assume that every program is a parallel composition of exactly two sequential processes. The syntax of a program $P$ is given by the following grammar:

$$ P :: S_1 \parallel S_2 $$

$$ S :: \textbf{skip} \mid x := e \mid S_1 ; S_2 \mid \textbf{do} \; G_1 \rightarrow S_1 \parallel \ldots \parallel G_n \rightarrow S_n \; \textbf{od} $$

The skip statement does not change any program variable; some non-zero time elapses.

Assignment $x := e$ changes variable $x$ to be the same value as expression $e$. The value of $e$ is computed at some instant after execution of the assignment is started; $x$ is changed instantaneously after some additional time elapses. Thus, execution of our assignment involves performing two atomic actions.

Sequential composition $S_1 ; S_2$ is executed by first executing $S_1$ and if and when $S_1$ terminates, $S_2$ is executed.

Execution of a do statement $S$ involves repeating the following until no longer possible: use guard evaluation action $Gvals$ to evaluate Boolean guards $G_1, \ldots, G_n$ and select a corresponding
\[ S_1 : \text{do } ps = \text{off} \land WL \leq 50 \rightarrow ps := \text{on} \\
\text{do } \begin{array}{l}
ps = \text{on} \land WL \geq 95 \rightarrow ps := \text{off} \\
- (ps = \text{off} \land WL \leq 50) \land -(ps = \text{on} \land WL \geq 95) \rightarrow \text{skip} 
\end{array} \text{ od} \\
\]

\[ S_2 : \text{do } vs = \text{close} \land \text{PRESSED} \rightarrow vs := \text{open} \\
\text{do } \begin{array}{l}
vs = \text{open} \land \text{PRESSED} \rightarrow vs := \text{close} \\
-\text{PRESSED} \rightarrow \text{skip} 
\end{array} \text{ od} \\
\]

Figure 1: Program P

statement \( S_1 \) whose guard is \text{true}. Then, execute \( S_1 \). Thus, once none of the guards evaluates to \text{true}, the \text{do} terminates. Execution of \( Gval_S \) is not instantaneous but uses values of variables that are all read together some time after \( Gval_S \) starts; the statement selection occurs some time after these values have been read.

Finally, execution of a parallel composition \( S_1 \parallel S_2 \) results in the simultaneous execution of \( S_1 \) and \( S_2 \). It terminates once both \( S_1 \) and \( S_2 \) have terminated.

**Program Semantics using Control-Graphs**

We represent a program using a control graph—a collection of nodes and edges, not unlike a flowchart. Each node models a delay prior to executing an atomic action; each edge models execution of an atomic action and describes a state change that occurs (instantaneously). Thus, \text{skip} gives rise to a single node followed by a single edge, whereas an assignment \( x := e \) gives rise to a sequence of two nodes—one whose outgoing edge computes the value of \( e \) and a successor whose outgoing edge updates \( x \).

Formally, a \text{control graph} is a tuple \((V, E, V_{\text{entry}}, E_{\text{exit}})\), where:

- \( V \) is a set of nodes.
- \( E \) is a set of edges. Each edge \((v, v')\) is labeled with a Boolean expression \( g \) and a multiple assignment \( op \) (possibly empty, i.e., \text{skip}). When convenient, we denote such a labeled edge by the 4-tuple \( (v, v', g, op) \). We call \( v \) the \text{source} node of the edge and call \( v' \) the \text{destination} node. Destination node \( v' \) must be either an element of \( V \) or the distinguished node “?”.
- \( V_{\text{entry}} \) is a set of \text{entry} nodes. \( V_{\text{entry}} \subseteq V \).
- \( E_{\text{exit}} \) is a set of \text{exit} edges, those edges with “?” as their destination node. \( E_{\text{exit}} \subseteq E \).

As an example, consider sequential subprogram \( S_1 \) of Figure 1. The control graph of \( S_1 \) is given in Figure 2. We use double circles to indicate entry nodes, and each edge is labeled with a Boolean expression and an assignment.\(^1\) The Boolean expression labeling an edge must hold in order for that edge to be traversed; the assignment is executed whenever the edge is traversed. Thus, in Figure 2, node \( v_0 \) is the sole entry node and allows the passage of time before \( Gval_{S_1} \) reads variables \( ps \) and \( WL \). The edge from \( v_0 \) to \( v_1 \), labeled with no guard and assignment \( t_1, t_2 := ps, WL \), models

\(^1\)When the guard is omitted, “true” is intended; when the assignment is omitted, “\text{skip}” is intended.
Figure 2: Control graph of $S_1$

that read. Edges from $v_1$ model the actual selection (and exit from the loop). The edge from $v_1$ to $v_2$ is labeled with Boolean expression $t_1 = \text{off} \land t_2 \leq 50$ to signify that assignment $ps := \text{on}$ is selected for execution only if the values read for $ps$ and $WL$ satisfy $ps = \text{off} \land WL \leq 50$. Nodes $v_2$ and $v_3$ model assignment $ps := \text{on}$; $v_4$ and $v_5$ model assignment $ps := \text{off}$; and $v_6$ models the skip.

Appendix A gives a procedure for translating a program into a control graph. When that procedure is used, the control graph $CG_P$ for any program $P$: $S_1 || S_2$ contains exactly two disconnected subgraphs, each with a single entry node: one entry node is for subprogram $S_1$ and the other is for subprogram $S_2$.

States, Phases, and Traces

A state is a mapping from variables to values. The variables are partitioned into program variables, environment variables, control variables, and clock variables. Program variables (which are typeset using lower-case identifiers) appear in assignments, as targets and/or expressions. Execution is the only way to change program variables. In the program of Figure 1, $ps$ and $vs$ are examples of program variables.

Environment variables may appear in guards and the expressions of assignments but may not appear as targets of assignments. We typeset environment variables using upper-case identifiers, to distinguish them from program variables. Environment variables are presumed to be changed by the environment, perhaps based on physical or chemical processes governed by scientific laws. In Figure 1, $WL$ and $PRESSED$ are environment variables.

For verification, it is useful to associate with each node $v$ of the control graph a Boolean control variable $v$. The value of control variable $v$ is true if and only if an atomic action modeled by an
edge from node \( v \) can next be executed. If control variable \( v \) is \textit{true}, then we say that node \( v \) is \textit{active}.

Finally, \textit{clock variables} capture elapsed time since various control graph nodes were last active. Clock variable \( \uparrow v \) records the elapsed time since the program started execution. Clock variable \( \downarrow v \) contains the elapsed time since control variable \( v \) last changed from \textit{false} to \textit{true} and has value \( \perp \) if \( v \) has never become \textit{true}. Thus, \( \uparrow v \) contains the elapsed time since node \( v \) last became active. And, clock variable \( \downarrow v \) contains the elapsed time since the start of control variable’s \( v \) last change from \textit{true} to \textit{false}; it has value \( \perp \) if \( v \) has never been \textit{true}.

Execution of a program is modeled as a sequence of phases \([18, 12]\). Each \textit{phase} gives values to the variables over some period of time. We denote a phase as a pair \(([r, r'], f)\), where \([r, r']\) is a closed interval of the reals and \( f \) is a mapping from \([r, r']\) to states. Phase \(([r, r'], f)\) associates state \( f(t) \) with any time \( t \) such that \( r \leq t \leq r' \).

A \textit{trace} \( \tau \) is a possibly infinite sequence of phases

\[
([r_1, r'_1], f_1), ([r_2, r'_2], f_2), \ldots
\]

such that for all \( i, r'_i = r_{i+1} \). The \textit{length} \( |\tau| \) of a trace \( \tau \) is defined to be infinity if there are infinite number of phases in the trace and otherwise is \( r'_n \) of its last phase \(([r_n, r'_n], f_n)\). A length \( m \) \textit{prefix}, with \( r_i < m \leq r'_i \), of \( \tau \), denoted by \( \tau_{-m} \), is a finite trace

\[
([r_1, r'_1], f_1), ([r_2, r'_2], f_2), \ldots, ([r_i, m], f_i).
\]

Notice that a trace associates two states with the endpoints of each phase.\( ^2 \) This is because we intend execution of an atomic action to delimit adjacent phases. State \( f_i(r'_i) \) occurs just prior to executing the atomic action that terminates phase \(([r_i, r'_i], f_i)\); state \( f_{i+1}(r_{i+1}) \) is the one produced by executing that atomic action.

Trace \( \tau \) of (3) is a behavior of a program \( P \), hence an element of \([[P]]\), provided all state changes are consistent with execution of \( P \). For this to be so, first we require of initial phase \(([r_1, r'_1], f_1)\):

1. \( r_1 = 0 \).
2. \( \textit{now} = 0 \) in state \( f_1(r_1) \).
3. Exactly the two control variables that correspond to the entry nodes of the control graph for \( P \) are \textit{true} in state \( f_1(r_1) \).
4. If a control variable \( v \) is \textit{true} in state \( f_1(r_1) \) then clock variable \( \uparrow v \) equals 0 in that state. Otherwise \( \uparrow v \) equals \( \perp \) in state \( f_1(r_1) \).
5. For all control variables \( v \), \( \downarrow v \) equals \( \perp \) in state \( f_1(r_1) \).

Second, we require that no program variable or control variable \( x \) changes value during a phase. (Environment variables and clock variables are not so constrained.)

\[
(\forall j, r_i \leq j \leq r'_i: f_i(j)(x) = f_i(r_i)(x))
\]

Third, we require that for any adjacent phases

\[
\ldots, ([r_i, r'_i], f_i), ([r_{i+1}, r'_{i+1}], f_{i+1}), \ldots
\]

\( ^2 \)There are two exceptions. Only a single state is associated with the very beginning of the trace; and for finite traces, only a single state is associated with the very end of the trace.
differences between states $f_i(r'_i)$ and $f_{i+1}(r_{i+1})$ are the result of executing a single atomic action. That is, the state change can be attributed to traversing an edge $e$ in the control graph where (i) the source node is active, (ii) the guard evaluates to true in state $f_i(r'_i)$, and (iii) any changed program variables in state $f_{i+1}(r_{i+1})$ are the result of executing the multiple assignment labeling edge $e$. Our control graphs are for 2-process programs, so without loss of generality, let control variables $v$ and $w$ be true in phase $([r_i, r'_i], f_i)$, control variables $v'$ and $w$ be true in phase $([r_{i+1}, r'_{i+1}], f_{i+1})$, and edge $(v, v', g, \bar{e} := \bar{e})$ be in the control graph.\footnote{If $\bar{x}$ and $\bar{e}$ are empty, then the effect of execution is the same as skip.} We formalize requirements (i) through (iii) by:

- Exactly two control variables are true in each of states $f_i(r'_i)$ and $f_{i+1}(r_{i+1})$, and one of those control variables is true in both $f_i(r'_i)$ and $f_{i+1}(r_{i+1})$. This corresponds to the restriction that only a single process executes a single atomic action between adjacent phases.

- Guard $g$ is true in state $f_i(r'_i)$. This means that edge $(v, v', g, op)$ can be traversed.

- The value of every program variable of $\bar{x}$ in state $f_{i+1}(r_{i+1})$ is equal to the value of the corresponding expression in $\bar{e}$ in state $f_i(r'_i)$; the value of no other program variable and no environment variables changes between $f_i(r'_i)$ and $f_{i+1}(r_{i+1})$. Thus, state changes are due to executing assignment $\bar{x} := \bar{e}$.

- Clock variable $\downarrow v'$ equals 0 in state $f_{i+1}(r_{i+1})$; clock variable and $\downarrow v$ equals 0 in state $f_i(r'_i)$. This causes the clock variables to have their intended meanings.

Finally, all clock variables change value within a phase $([r_i, r'_i], f_i)$ in the expected way. The value of a clock variable $c$ at time $t$, where $t$ satisfies $r_i \leq t < r'_i$, is given by

$$f_i(t)(c) = f_i(r_i)(c) + (t - r_i).$$

A clock variable $c$ that is not reset in state $f_i(r'_i)$ also satisfies $f_i(r'_i)(c) = f_i(r_i)(c) + (r'_i - r_i)$.

The following diagram summarizes how the starting and ending states of adjacent phases in a trace are related. A dashed arrow indicates changes to environment and clock variables; a solid arrow denotes changes to program variables, control variables, and clock variables that occur by traversing a control graph edge. The trace begins at state $f_1(0)$ and the state changes continuously according to the function $f_1$, until time $r'_1$. At time $r'_1$, an instantaneous state change occurs corresponding to execution of some atomic action. This causes the state to change from $f_1(r'_1)$ to $f_2(r_2)$, based on the assignment labeling the edge of the control graph that is traversed and the resetting of certain clock variables.

\[
\begin{align*}
  f_1(0) \quad &\rightarrow \quad f_1(r'_1) \\
  &\downarrow \\
  f_2(r_2) \quad &\rightarrow \quad f_2(r'_2) \\
  &\downarrow \\
  f_3(r_3) \quad &\rightarrow \quad \ldots
\end{align*}
\]
4 Properties

We now introduce a language for expressing properties. We restrict consideration to safety properties [15], properties that assert some "bad thing" does not happen during execution. Informally, formula $\text{Init} \Rightarrow \Box I$ defines the property containing all traces $\tau$ such that (i) $\text{Init}$ does not hold initially on $\tau$ or (ii) $I$ holds throughout $\tau$. Thus, $I$ implies that the "bad thing" being prescribed by the safety property has not happened.

In formula $\text{Init} \Rightarrow \Box I$, we call $\text{Init}$ and $I$ assertions and assume that they are defined by the grammar below. There, we assume $x$ is a program variable, $X$ is an environment variable, and $v$ is a control variable. We also assume a set $C$ of real constants. Finally, $op_{rel}$ denotes a relational operator and $op_{arith}$ an arithmetic operator.

$$A ::= T \ op_{rel} \ T' \mid \neg A \mid A \land A' \mid (\forall x. A : A')$$

$$T ::= C \mid Var \mid \frac{d}{dt}(Var) \mid T \ op_{arith} \ T' \mid T\{T'\} \mid \int_{T}^{T'} T''$$

$$Var ::= x \mid X \mid v \mid \uparrow v \mid \downarrow v \mid \text{now}$$

(4)

For any variable $a$, term $\frac{d}{dt} (a)$ is the first derivative of $a$ with respect to time. The past term $T\{T'\}$ equals the value of $T$ at the (past) state existing $T'$ units ago. And, $\int_{T}^{T'} T''$ is the value of the definite integral of term $T''$ between $T$ and $T'$.

We formalize the value $\mathcal{M}_{T}(T)$ of a term $T$ in a finite trace $\tau$ inductively.

$$\mathcal{M}_{\tau}(C)(\tau) = C$$

$$\mathcal{M}_{\tau}(Var)(\tau) = f_{i}(\tau_{i})(Var), \text{ where } (\tau_{i}, \tau_{i}', f_{i}) \text{ is the last phase in } \tau$$

$$\mathcal{M}_{\tau}(\frac{d}{dt}(Var))(\tau) = \lim_{\Delta \tau \to 0} \frac{\mathcal{M}_{\tau}(Var)(\tau) - \mathcal{M}_{\tau}(Var)(\tau_{-\Delta \tau})}{\Delta \tau}$$

(5)

$$\mathcal{M}_{\tau}(T \ op_{arith} T')(\tau) = \mathcal{M}_{\tau}(T)(\tau) \ op_{arith} \mathcal{M}_{\tau}(T')(\tau)$$

$$\mathcal{M}_{\tau}(T\{T'\})(\tau) = \mathcal{M}_{\tau}(T)(\tau_{-[\tau]}) - \mathcal{M}_{\tau}(T')(\tau)$$

$$\mathcal{M}_{\tau}(\int_{T}^{T'} T'')(\tau) = \int_{\mathcal{M}_{\tau}(T)(\tau)}^{\mathcal{M}_{\tau}(T'')(\tau_{-\tau})} \mathcal{M}_{\tau}(T'')(\tau_{-\tau}) d\tau$$

The value $\mathcal{M}_{A}(A)$ of an assertion $A$ in finite trace $\tau$ is a Boolean function defined in the usual way using $\mathcal{M}_{\tau}$.

An assertion $A$ is defined to be valid iff for every finite trace $\tau$, $\mathcal{M}_{A}(A)(\tau) = true$. We assume a deductive system is available for proving validity of assertions.

Finally, we formalize the set of traces in $[[\text{Init} \Rightarrow \Box I]]$. It is just those finite and infinite traces $\tau$ for which $\mathcal{M}_{A}(\text{Init})(\tau,0) = false$ or, for all $j$, $\mathcal{M}_{A}(I)(\tau,j) = true$:

$$\tau \in [[\text{Init} \Rightarrow \Box I]] : \mathcal{M}_{A}(\text{Init})(\tau,0) = false \text{ or } (\forall j.0 \leq j \leq |\tau| : \mathcal{M}_{A}(I)(\tau,j) = true)$$

(6)

5 Verifying Hybrid Systems

When verifying a hybrid system, we are interested in proving that executions of a program $P$ satisfy $[[\text{Init} \Rightarrow \Box I]]$ if environment variables change values according to some given constraints (presumably dictated by scientific laws). In the parlance of Section 2, this is an instance of proving that $P$ satisfies $[[\text{Init} \Rightarrow \Box I]]$ under an environment $\mathcal{E}$, where $\mathcal{E}$ is the property asserting that environment variables only change values according to the given constraints.
Define the hybrid-environment property $\Box Env$, for $Env$ an assertion, to be the set of all finite and infinite traces where $Env$ holds throughout:

$$\tau \in [\Box Env] : (\forall j.0 \leq j \leq |\tau| : \mathcal{M}_A(Env)(\tau_j) = \text{true})$$

(7)

Hybrid-system verification is thus equivalent to establishing $\langle P, \Box Env, \text{Init} \Rightarrow \Box I \rangle \in ESat$. According to (2), it suffices to prove $[\langle P \rangle] \subseteq ([\text{Init} \Rightarrow \Box I] \cup [\Box Env])$. We accomplish this by introducing another property $[[\mathcal{H}]]$, called a hybrid proof outline, satisfying:

$$[\langle P \rangle] \subseteq [[\mathcal{H}]]$$

(8)

$$[[\mathcal{H}]] \subseteq ([\text{Init} \Rightarrow \Box I] \cup [\Box Env])$$

(9)

A hybrid proof outline, like an ordinary proof outline, associates assertions with control points. In particular, a hybrid proof outline associates an assertion with each node in a control graph. Assertions are of a restricted form so they cannot be invalidated by changes to environment variables (which might occur while a given node of a control graph remains active).

R1: Clock variables $c$ and environment variables $X$ appear only in past terms having the form $X\{c\}$. Such terms do not change value during a phase, even as clock variables advance and the environment variables are updated.

R2: Terms using derivatives are not permitted.

Formally, a hybrid proof outline is a triple $\mathcal{H} = (CG_P, \gamma, Env)$ where $CG_P$ is a control graph for a program $P$, $\gamma$ maps each node $v$ among the nodes $V$ of $CG_P$ to an assertion $\gamma_v$ satisfying R1 and R2, and $\Box Env$ is a hybrid-environment property. The assertion

$$I_\mathcal{H} : \bigwedge_{v \in V} v \Rightarrow \gamma_v$$

(10)

is called the invariant of $\mathcal{H}$. Property $[[\mathcal{H}]]$ is defined to be the set of all finite and infinite traces $\tau$ such that (i) $\tau$ does not satisfy hybrid-environment property $\Box Env$ or (ii) $\tau \in [[I_\mathcal{H} \Rightarrow \Box I_\mathcal{H}]]$.

For an assertion $A$, we write $A[x := e]$ to denote the textual substitution of every free occurrence of $x$ in $A$ by $e$. The following theorems give conditions for verifying (8) and (9) above, and therefore they give a method for establishing $[\langle P \rangle] \subseteq ([\text{Init} \Rightarrow \Box I] \cup [\Box Env])$.

**Theorem 1** Given a hybrid proof outline $\mathcal{H} = ((V, E, V_{entry}, V_{exit}), \gamma, Env)$ for program $P$, then $[\langle P \rangle] \subseteq [[\mathcal{H}]]$ if the following conditions hold:

- For every $(v, u, g, op) \in E$:
  
  $$(\text{Env} \land v \land \gamma_v \land g \land v = 0) \Rightarrow (u \land \gamma_u)[op, \uparrow u := 0, v := \text{false}, u := \text{true}]$$

  is valid.

- For every $(v, u, g, op) \in E$ and every $w \in V$ such that $v$ and $w$ are nodes of different processes:
  
  $$(\text{Env} \land v \land \gamma_v \land g \land w \land \gamma_w \land v = 0) \Rightarrow (w \land \gamma_w)[op, \uparrow u := 0, v := \text{false}, u := \text{true}]$$

  is valid.

**Theorem 2** Given a hybrid proof outline $\mathcal{H} = ((V, E, V_{entry}, V_{exit}), \gamma, Env)$, if

$$(\text{Env} \land \text{Init}) \Rightarrow I_\mathcal{H} \text{ and } (\text{Env} \land I_\mathcal{H}) \Rightarrow I$$

are valid, then $[[\mathcal{H}]] \subseteq ([\text{Init} \Rightarrow \Box I] \cup [\Box Env])$.

The proofs of both theorems are in Appendix B.
Figure 3: The control graph of $S_2$

6 Example

To illustrate the approach, we return to the control program in Figure 1. Sub-program $S_1$ reads the water level ($WL$) in a tank. If the level is too low—less than or equal to 50—then a pump is activated, causing water to be added, until the level reaches 95. Sub-program $S_2$ monitors a control button. When the button is pressed, $S_2$ toggles the valve state. The two components for the control graph of this program appear in Figures 2 and 3.

A hybrid-environment property $\square Env$ for this system asserts that changes to the water level are based on the pump rate and valve state. We assume a pump with throughput $0.5 \, \ell/sec$ and a valve that passes $0.25 \, \ell/sec$. When the valve is closed and the pump is off the water-level does not change:

$$(ps = off \land vs = close) \Rightarrow \frac{d}{dt}(WL) = 0$$

When the valve is open and the pump is on, the water-level increases at the rate of $0.25 \, \ell/sec$, reflecting the relative capacities of the valve and pump:

$$(ps = on \land vs = open) \Rightarrow \frac{d}{dt}(WL) = 0.25$$

When the valve is open but the the pump is off, the water-level decreases at the rate $-0.25 \, \ell/sec$:

$$(ps = off \land vs = open) \Rightarrow \frac{d}{dt}(WL) = -0.25$$
Finally, when the valve is closed and the pump is on, the water-level increases at the rate of 0.5 \( \ell/sec \):

\[(ps = on \land vs = close) \Rightarrow \frac{d}{dt}(WL) = 0.5\]

The water level changes over time. This is reflected by the following assertion, which states that while any control variable \( u \) holds, the water-level equals whatever it was when \( u \) first became true plus the change to the water-level since that time:

\[
\bigwedge_{u \in \{v_0, \ldots, v_6, w_0, \ldots, w_6\}} \left( u \Rightarrow WL = WL\{\uparrow u\} + \int_{\uparrow u - 1}^{\text{now}} \frac{d}{dt}(WL) \right)
\]

We assume that execution of each program step takes at least 0.5 units of time and at most 1 unit of time:

\[
\bigwedge_{u \in \{v_0, \ldots, v_6, w_0, \ldots, w_6\}} (u \Rightarrow (\uparrow u \leq 1)) \land ((\downarrow u = 0) \Rightarrow (\downarrow u \geq 0.5))
\]

Notice that real-time execution bounds are defined using an assertion about the environment. This seems natural, since there is nothing intrinsic about the program text that supplies such bounds. Rather, the bounds are an artifact of the particular processor executing the program. Moreover, associating the bounds with the environment makes it possible to use our verification framework for different real-time behaviors.

The property that we wish to establish is that our control program ensures that the water-level remains between 48 and 98. We formalize this property as \( I \land \Box I \), where:

\[
\begin{align*}
\text{Init} : & \quad ps = \text{off} \land vs = \text{close} \land v_0 \land w_0 \land \neg \text{PRESSED} \land WL\{\uparrow v_0\} = 90 \\
\text{I} : & \quad 48 < WL < 98
\end{align*}
\]

To prove that this property holds, we construct a hybrid proof outline \( \mathcal{H} \), with the following mapping \( \gamma \) that assigns an assertion to every node in the control graph. Let \( \oplus \) denote the xor logic operation.

\[
\begin{align*}
\gamma_{v_i} : & \quad (v_0 \oplus \ldots \oplus v_6) \land (w_0 \oplus \ldots \oplus w_6) \quad \text{for} \quad i = 0..6 \\
\gamma_{v_0} : & \quad (vs = \text{open} \lor vs = \text{close}) \land
\begin{align*}
(ps = on \lor ps = \text{off}) \land
\begin{align*}
ps = on & \Rightarrow 48.5 < WL\{\uparrow v_0\} < 96 \land \\
ps = off & \Rightarrow 49.5 < WL\{\uparrow v_0\} < 98
\end{align*}
\end{align*}
\]
\[
\gamma_{v_1} : (vs = \text{open} \lor vs = \text{close}) \land
\begin{align*}
(ps = on \lor ps = \text{off}) \land
\begin{align*}
ps = on & \Rightarrow 48.75 < WL\{\uparrow v_1\} < 96.5 \land \\
ps = off & \Rightarrow 49.25 < WL\{\uparrow v_1\} < 98 \land \\
t_1 = ps \land t_2 = WL\{\uparrow v_1\}
\end{align*}
\end{align*}
\]
\[
\gamma_{v_2} : (vs = \text{open} \lor vs = \text{close}) \land ps = \text{off} \land
\begin{align*}
49 < WL\{\uparrow v_2\} \leq 50
\end{align*}
\]
\[
\gamma_{v_3} : (vs = \text{open} \lor vs = \text{close}) \land ps = \text{off} \land
\begin{align*}
48.75 < WL\{\uparrow v_3\} \leq 50
\end{align*}
\]
\[
\gamma_{v_4} : (vs = \text{open} \lor vs = \text{close}) \land ps = \text{on} \land
\begin{align*}
95.25 < WL\{\uparrow v_4\} < 97
\end{align*}
\]

\[ \gamma_v : (v = \text{open} \lor v = \text{close}) \land p = \text{on} \land \\
95.5 \leq WL\{v_5\} < 97.5 \\
\gamma_w : (v = \text{open} \lor v = \text{close}) \land \\
(p = \text{on} \lor p = \text{off}) \land \\
p = \text{on} \implies 49 < WL\{v_6\} < 95.5 \\
p = \text{off} \implies 49.75 < WL\{v_6\} < 96.5 \]

According to Theorems 1 and 2, we must then check the set of verification conditions listed in Appendix C.

7 Discussion

Our work is perhaps closest in spirit to the various approaches for reasoning about open systems. An open system is one that interacts with its environment through shared memory or communication. The execution of such a system is commonly modeled as an interleaving of steps by the system and steps by the environment. Since an open system is not expected to function properly in an arbitrary environment, its specification typically will contain explicit assumptions about the environment. Such specifications are called assume-guarantee specifications, because they guarantee behavior when the environment satisfies some assumptions. Logics for verifying safety properties of assume-guarantee specifications are discussed in [9, 14, 21]; liveness properties are treated in [1, 3, 23]; and model-checking techniques based on assume-guarantee specifications are introduced in [6, 11].

Our approach differs from this open systems work both in the role played by the environment and in how state changes are made by the environment. We use the environment to represent aspects of the computation model and the scientific laws governing the behavior of environment variables—not as an abstraction of the behaviors for other agents that will run concurrently with the system. This generalizes what is advocated in [8] for reasoning about fair computations in temporal logic. Second, in our approach, every state change obeys constraints defined by the environment, while in the open systems view only state changes that are attributed to the environment must obey those constraints.

Interest in verification of hybrid systems is an outgrowth of work in verifying real-time bounds for concurrent programs. A rather substantial literature exists on the subject; see [7] for a collection of surveys. The problem of reasoning about arbitrary continuous-valued state components was first discussed in [24], in connection with process control program for railroad control. That work was ultimately published in [20].

Our underlying semantic model—traces—is similar to the hybrid traces of [18]. A hybrid trace consists of continuous and discrete moments. A continuous moment is mapped to a single state, and a discrete moment may be mapped to several states. With our notion of traces, every intermediate discrete moment is mapped to exactly two states.

Our computation model—control graphs and hybrid-environment properties—share features with phase transition systems [18, 12], hybrid statecharts [19], and hybrid automata [2]. Our computation model differs in its separation of program execution from changes to the environment. The control graph models program execution and the hybrid-environment property models state changes to the continuous-valued variables. One advantage of this separation is that changes to the computation model and to the physical laws can be easily accommodated. A second advantage is that assertions associated with control points in a program (i.e., nodes in the control graph) can be simpler because they need not explicitly mention environment state components.
Our specification language contains constructs for derivatives and integrals. Such constructs also appear in the specification languages of [4, 12, 17, 5, 22]. Our verification methodology extends the Hoare-logic methodology of [16] to hybrid system. Deductive-systems for proving safety properties of hybrid system are also presented in [19, 13]. Our work differs mainly in its independence from a particular computation model.

References


A Constructing a Control Graph

The control graph $CG_S$ that corresponds to a sub-program $S$ is defined inductively, as follows:

* For $S$ a skip: Define $V = \{v_0\}$, $V_{entry} = \{v_0\}$, $E = \{e_0\}$ where $e_0 = (v_0, ?, true, skip)$, and $E_{exit} = \{e_0\}$. 

13
• For $S$ an assignment $x := e(y)$: Define $V = \{v_0, v_1\}$, $V_{entry} = \{v_0\}$, and $E = \{e_0, e_1\}$ where $e_0 = (v_0, v_1, true, t := y)$, $e_1 = (v_1, ?, true, x := e(t))$, and $E_{exit} = \{e_1\}$.

• For $S$ a statement composition $S_1; S_2$: Let $(V^1, E^1, V_{entry}^1, E_{exit}^1)$ be the control graph for $S_1$ and let $(V^2, E^2, V_{entry}^2, E_{exit}^2)$ be the control graph for $S_2$. Define $V = V^1 \cup V^2$, $V_{entry} = V_{entry}^1$, $E_{exit} = E_{exit}^2$, and

$$E = E^1 \cup E^2 - E_{exit}^1 \cup \{(v, v', g, op) | \exists (v, ?, g, op) \in E_{exit}^1 \text{ and } v' \in V_{entry}^2\}$$

• For $S$ an iteration $do \ G_1 \rightarrow S_1 [\mid G_2 \rightarrow S_2 od$: Let $\bar{y}$ be the list of variables mentioned by $G_1$ and $G_2$. Let $(V^1, E^1, V_{entry}^1, E_{exit}^1)$ be the control graph for $S_1$ and let $(V^2, E^2, V_{entry}^2, E_{exit}^2)$ be the control graph for $S_2$. Define $V = V^1 \cup V^2 \cup \{v_0, v_1\}$, $V_{entry} = \{v_0\}$,

$$E = E^1 \cup E^2 - E_{exit}^1 \cup E_{exit}^2 \cup \{(v_0, v_1, true, t := \bar{y})\} \cup \{(v_1, v, G_1[t/\bar{y}], skip) | v \in V_{entry}^1\} \cup \{(v_1, ?, G_2[t/\bar{y}], skip) | v \in V_{entry}^2\} \cup \{(v_1, ?, \neg G_1[t/\bar{y}] \land \neg G_2[t/\bar{y}], skip)\} \cup \{(v, v_0, g, op) | \exists (v, ?, g, op) \in E_{exit}^1 \cup E_{exit}^2\}$$

and $E_{exit} = \{(v_0, ?, \neg G_1[t/\bar{y}] \land \neg G_2[t/\bar{y}], skip)\}$

• For $S$ a parallel composition $S_1 || S_2$: Let $(V^1, E^1, V_{entry}^1, E_{exit}^1)$ be the control graph for $S_1$ and let $(V^2, E^2, V_{entry}^2, E_{exit}^2)$ be the control graph for $S_2$. Define $V = V^1 \cup V^2$, $V_{entry} = V_{entry}^1 \cup V_{entry}^2$, $E_{exit} = E_{exit}^1 \cup E_{exit}^2$, and $E = E^1 \cup E^2$.

B Soundness Proofs

Lemma 1 Let $A$ be an assertion satisfying R1 and R2, and let $\tau$ be a finite trace. If $\tau'$ is a finite trace, $\tau$ is a prefix of $\tau'$, and

$$(\forall j. |\tau| < j \leq |\tau'| : \tau'_{\cdot j}(x) = \tau(x) \text{ for every program variable } x, \text{ and } \tau'_{\cdot j}(y) = \tau(y) + j - |\tau| \text{ for every clock variable } y),$$

then $(\forall j. |\tau| < j \leq |\tau'| : M_A(A)(\tau) = M_A(A)(\tau'_{\cdot j}))$.

Proof: Since $A$ satisfies R1 and R2, it refers only to program variables and to past terms of the form $z\{y\}$, such that $z$ is an environment variable and $y$ is a clock variable. Therefore, the evaluations of the terms of $A$ at $\tau$ and at $\tau'_{\cdot j}$ return that same values.

Theorem 1 Given a hybrid proof outline $\mathcal{H} = ((V, E, V_{entry}, E_{exit}), \gamma, Env)$ for program $P$, then $[[P]] \subseteq [[\mathcal{H}]]$ if the following conditions hold:

• For every $(v, u, g, op) \in E$:

$$(Env \land v \land \gamma_v \land g \land v = 0) \Rightarrow (u \land \gamma_u)[op, [u := 0, v := false, u := true]$$

is valid.

$^4$\(e(\bar{y})\) denotes that expression $e$ refers to variables in list $\bar{y}$.

$^5$To simplify the presentation, we assume only two alternatives.
• For every \((v, u, g, op) \in E\) and every \(w \in V\) such that \(v\) and \(w\) are nodes of different processes:

\[(Env \land v \land \gamma_v \land g \land w \land \gamma_w \land v = 0) \Rightarrow (w \land \gamma_w)[op, \lceil u := 0, v := false, u := true]\]

is valid.

**Proof:**

1. Let \(\tau \in [[P]], \) where \(\tau = ([r_1, r'_1], f_1), ([r_2, r'_2], f_2), \ldots\)

2. According to definition of \([[H]]\) we need to prove that either, (i) \(\tau \notin [[\Box Env]]\), or (ii) \(M_A(I_H)(\tau, 0) = false\), or (iii) \((\forall j, 0 \leq j \leq |\tau| : M_A(I_H)(\tau, j) = true)\). It therefore suffices to prove that if (i) and (ii) do not hold then (iii) holds.

3. Assume (i) and (ii) of 2 do not hold, so \(\tau \in [[\Box Env]]\) and \(M_A(I_H)(\tau, 0) = true\).

4. By induction on the number \(i\) of the phases in \(\tau\), we next prove:

\[(\forall j, 0 \leq j \leq |\tau| : M_A(I_H)(\tau, j) = true)\]

**Basis:** \(i = 1\). According to 3 we have that \(M_A(I_H)(\tau, 0) = true\). According to 1 and Lemma 1 we conclude \(\forall j, 0 \leq j \leq r'_1 : M_A(I_H)(\tau, j) = true\).

**Step:** Assume \((\forall j, 0 \leq j \leq r'_{i-1} : M_A(I_H)(\tau, j) = true)\) holds for \(i > 1\). We prove:

\[(\forall j, r_i < j \leq r'_i : M_A(I_H)(\tau, j) = true)\]

(a) \(M_A(Env \land I_H)(\tau, r'_i) = true\), by the step assumption and 3.

(b) Without lose of generality, assume \((v_{i-1}, w_{i-1}) \xrightarrow{op} (v_i, w_i)\) such that \(v_{i-1} \neq v_i, w_{i-1} = w_i, \) and \(v_{i-1}\) and \(w_{i-1}\) are true in \(\tau, r'_{i-1}\). According to 1, 4(a), and the definition of \(I_H\):

\[M_A(Env \land v_{i-1} \land w_{i-1} \land \gamma_{v_{i-1}} \land \gamma_{w_{i-1}} \land g \land v_{i-1} = 0)(\tau, r'_{i-1}) = true\]

(c) Let \(\tau^*\) be obtained by extending \(\tau, r'_{i-1}\) with the single phase \(([r_i, r_i], f_i)\). Then, according to the hypotheses of the theorem and 4(b)

\[M_A(v_i \land w_i \land \gamma_{v_i} \land \gamma_{w_i})(\tau^*) = true\]

(d) Let \(j\) be such that \(r_i < j \leq r'_i\). Then since \(\tau \in [[P]]\), according to 1 and due to Lemma 1:

\[M_A(v_i \land w_i \land \gamma_{v_i} \land \gamma_{w_i})(\tau^*) = M_A(v_i \land w_i \land \gamma_{v_i} \land \gamma_{w_i})(\tau, j)\]

(e) According to 4(c), 4(d) and definition (10) of \(I_H\), \(M_A(I_H)(\tau, j) = true\).

**Theorem 2** Given a hybrid proof outline \(H = ((V, E, V_{entry}, E_{exit}, \gamma, Env), I)\), if

\[(Env \land Init) \Rightarrow I_H \quad \text{and} \quad (Env \land I_H) \Rightarrow I\]

are valid, then \([[H]] \subseteq \left(\left[\left[Init \Rightarrow \Box I\right]\right] \cup \left[\left[\Box Env\right]\right]\right)]\).

**Proof:**
1. Let $\tau \in \llbracket \mathcal{H} \rrbracket$.

2. According to definition of $\llbracket \mathcal{H} \rrbracket$, either:

   (a) $\tau \in \llbracket \mathcal{D} \mathcal{E} \mathcal{N} \mathcal{V} \rrbracket$. In this case, $\tau \in (\llbracket \text{Init} \Rightarrow \Box I \rrbracket \cup \llbracket \Box \mathcal{D} \mathcal{E} \mathcal{N} \mathcal{V} \rrbracket)$.

   (b) $\tau \in \llbracket \mathcal{D} \mathcal{E} \mathcal{N} \mathcal{V} \rrbracket$ and $\mathcal{M}_A(I_H)(\tau, 0) = \text{false}$. In this case, since $(\mathcal{E} \mathcal{N} \mathcal{V} \wedge \text{Init}) \Rightarrow I_H$ we know $\mathcal{M}_A(\mathcal{E} \mathcal{N} \mathcal{V} \wedge \text{Init})(\tau, 0) = \text{false}$. However, since $\mathcal{M}_A(\mathcal{E} \mathcal{N} \mathcal{V})(\tau, 0) = \text{true}$ we get $\mathcal{M}_A(\text{Init})(\tau, 0) = \text{false}$. Therefore according to definition (6), $\tau \in (\llbracket \text{Init} \Rightarrow \Box I \rrbracket \cup \llbracket \Box \mathcal{D} \mathcal{E} \mathcal{N} \mathcal{V} \rrbracket)$.

   (c) $\tau \in \llbracket \mathcal{D} \mathcal{E} \mathcal{N} \mathcal{V} \rrbracket$ and $(\forall j. 0 \leq j \leq |\tau| : \mathcal{M}_A(\mathcal{E} \mathcal{N} \mathcal{V} \wedge I_H)(\tau, j) = \text{true})$. In this case, since by hypothesis $(\mathcal{E} \mathcal{N} \mathcal{V} \wedge I_H) \Rightarrow I$ we get $(\forall j. 0 \leq j \leq |\tau| : \mathcal{M}_A(I)(\tau, j) = \text{true})$. Thus, $\tau \in (\llbracket \text{Init} \Rightarrow \Box I \rrbracket \cup \llbracket \Box \mathcal{D} \mathcal{E} \mathcal{N} \mathcal{V} \rrbracket)$.

C The verification conditions

The following conditions must be proved valid in order to complete the verification. To use Theorem 2, we must prove:

- $(\mathcal{E} \mathcal{N} \mathcal{V} \wedge \text{Init}) \Rightarrow I_H$
- $(\mathcal{E} \mathcal{N} \mathcal{V} \wedge I_H) \Rightarrow I$

For Theorem 1, we must prove:

- $(\mathcal{E} \mathcal{N} \mathcal{V} \wedge v_0 \wedge \gamma_{v_0} \wedge \downarrow v_0 = 0) \Rightarrow (v_1 \wedge \gamma_{v_1})[t_1, t_2 := ps, WL, \downarrow v_1 := 0, v_0 := \text{false}, v_1 := \text{true}]
- (\mathcal{E} \mathcal{N} \mathcal{V} \wedge v_1 \wedge \gamma_{v_1} \wedge t_1 = \text{off} \wedge t_2 \leq 50 \wedge \downarrow v_1 = 0) \Rightarrow (v_2 \wedge \gamma_{v_2})[\downarrow v_2 := 0, v_1 := \text{false}, v_2 := \text{true}]
- (\mathcal{E} \mathcal{N} \mathcal{V} \wedge v_1 \wedge \gamma_{v_1} \wedge t_1 = \text{on} \wedge t_2 \geq 95 \wedge \downarrow v_1 = 0) \Rightarrow (v_4 \wedge \gamma_{v_4})[\downarrow v_4 := 0, v_1 := \text{false}, v_4 := \text{true}]
- (\mathcal{E} \mathcal{N} \mathcal{V} \wedge v_1 \wedge \gamma_{v_1} \wedge (\neg (t_1 = \text{off} \wedge t_2 \leq 50) \wedge \neg (t_1 = \text{on} \wedge t_2 \geq 95)) \wedge \downarrow v_1 = 0) \Rightarrow (v_6 \wedge \gamma_{v_6})[\downarrow v_6 := 0, v_1 := \text{false}, v_6 := \text{true}]
- (\mathcal{E} \mathcal{N} \mathcal{V} \wedge v_2 \wedge \gamma_{v_2} \wedge \downarrow v_2 = 0) \Rightarrow (v_3 \wedge \gamma_{v_3})[\downarrow v_3 := 0, v_2 := \text{false}, v_3 := \text{true}]
- (\mathcal{E} \mathcal{N} \mathcal{V} \wedge v_4 \wedge \gamma_{v_4} \wedge \downarrow v_4 = 0) \Rightarrow (v_5 \wedge \gamma_{v_5})[\downarrow v_5 := 0, v_4 := \text{false}, v_5 := \text{true}]
- (\mathcal{E} \mathcal{N} \mathcal{V} \wedge v_6 \wedge \gamma_{v_6} \wedge \downarrow v_6 = 0) \Rightarrow (v_0 \wedge \gamma_{v_0})[\downarrow v_0 := 0, v_6 := \text{false}, v_0 := \text{true}]
- (\mathcal{E} \mathcal{N} \mathcal{V} \wedge v_3 \wedge \gamma_{v_3} \wedge \downarrow v_3 = 0) \Rightarrow (v_0 \wedge \gamma_{v_0})[ps := on, \downarrow v_0 := 0, v_3 := \text{false}, v_0 := \text{true}]
- (\mathcal{E} \mathcal{N} \mathcal{V} \wedge v_5 \wedge \gamma_{v_5} \wedge \downarrow v_5 = 0) \Rightarrow (v_0 \wedge \gamma_{v_0})[ps := off, \downarrow v_0 := 0, v_5 := \text{false}, v_0 := \text{true}]
- (\mathcal{E} \mathcal{N} \mathcal{V} \wedge w_1 \wedge g \wedge \downarrow w_1 = 0) \Rightarrow (w_2)[op, \downarrow w_2 := 0, w_1 := \text{false}, w_2 := \text{true}]$ for every $(w_i, w_j, g, op) \in E$.
- $(\mathcal{E} \mathcal{N} \mathcal{V} \wedge v \wedge \gamma_v \wedge g \wedge \downarrow v = 0) \Rightarrow (w)[op, \downarrow u := 0, v := \text{false}, u := \text{true}]$ for $w$ in $CG_{S_2}$ and $(v, u, g, op)$ in $CG_{S_1}$.
- $(\mathcal{E} \mathcal{N} \mathcal{V} \wedge w \wedge g \wedge v \wedge \gamma_v \wedge \downarrow w = 0) \Rightarrow (v \wedge \gamma_v)[op, \downarrow u := 0, w := \text{false}, u := \text{true}]$ for $v$ in $CG_{S_1}$ and $(w, u, g, op)$ in $CG_{S_2}$.  

16