Hybrid Verification by Exploiting the Environment*

Limor Fix** and Fred B. Schneider

Department of Computer Science, Cornell University, Ithaca, New York 14853

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.

* 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 Fullbright post-doctoral award.

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 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 Φ. Program P and the property Φ 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 Φ, denoted $\langle P, \Phi\rangle \in Sat$, exactly when all behaviors of P are permitted by Φ:

$$\langle P, \Phi\rangle \in Sat \quad \text{if and only if} \quad [[P]] \subseteq [[\Phi]]$$

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 $\mathcal{E}$ defined by an environment, the *feasible behaviors* of a program P under $\mathcal{E}$ are those behaviors of P that are also in $\mathcal{E}$: $[[P]] \cap [[\mathcal{E}]]$. A program P *satisfies* a property Φ *under* an environment $\mathcal{E}$, denoted $\langle P, \mathcal{E}, \Phi\rangle \in ESat$, if and only if every feasible behavior of P under $\mathcal{E}$ is in Φ:

$$\langle P, \mathcal{E}, \Phi\rangle \in ESat \quad \text{if and only if} \quad ([[P]] \cap [[\mathcal{E}]]) \subseteq [[\Phi]] \tag{1}$$

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

$$\langle P, \mathcal{E}, \Phi\rangle \in ESat \quad \text{if and only if} \quad [[P]] \subseteq ([[\Phi]] \cup \overline{[[\mathcal{E}]]}), \tag{2}$$

where $\overline{[[\mathcal{E}]]}$ denotes the set complement of $[[\mathcal{E}]]$.

3 Programs

Consider a simple programming language having an empty statement (**skip**), assignment (:=), sequential composition (;), iteration (**do**), and parallel composition (||). 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:

$$P :: S_1 \mid\mid S_2$$

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

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* $Gval_S$ to evaluate Boolean guards $G_1, \ldots, G_n$ and select a corresponding statement S_i whose guard is *true*. Then, execute S_i. Thus, once none of the guards evaluates to *true*, the **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 $S_1 \mid\mid S_2$ 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, **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 *control graph* is a tuple $(V, E, V_{entry}, E_{exit})$, 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., **skip**). When convenient, we denote such a labeled edge by the 4-tuple (v, v', g, op). We call v the *source* node of the edge and call v' the *destination* node. Destination node v' must be either an element of V or the distinguished node "?".
- V_{entry} is a set of *entry* nodes. $V_{entry} \subseteq V$.
- E_{exit} is a set of *exit* edges, those edges with "?" as their destination node. $E_{exit} \subseteq E$.

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

$$
\begin{array}{l}
S_1 : \textbf{do}\ ps = \mathit{off} \wedge WL \leq 50 \rightarrow ps := on \\
\quad [\!]\ \ ps = on \wedge WL \geq 95 \rightarrow ps := \mathit{off} \\
\quad [\!]\ \ \neg(ps = \mathit{off} \wedge WL \leq 50) \wedge \neg(ps = on \wedge WL \geq 95) \rightarrow \textbf{skip} \\
\quad \textbf{od} \\
|| \\
S_2 : \textbf{do}\ vs = close \wedge PRESSED \rightarrow vs := open \\
\quad [\!]\ \ vs = open \wedge PRESSED \rightarrow vs := close \\
\quad [\!]\ \ \neg PRESSED \rightarrow \textbf{skip} \\
\quad \textbf{od}
\end{array}
$$

Fig. 1. Program P

each edge is labeled with a Boolean expression and an assignment.[3] 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 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 $t_1 = \mathit{off} \wedge t_2 \leq 50$ to signify that assignment $ps := on$ is selected for execution only if the values read for ps and WL satisfy $ps = \mathit{off} \wedge WL \leq 50$. Nodes v_2 and v_3 model assignment $ps := on$; v_4 and v_5 model assignment $ps := \mathit{off}$; 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,

[3] When the guard is omitted, "*true*" is intended; when the assignment is omitted, "**skip**" is intended.

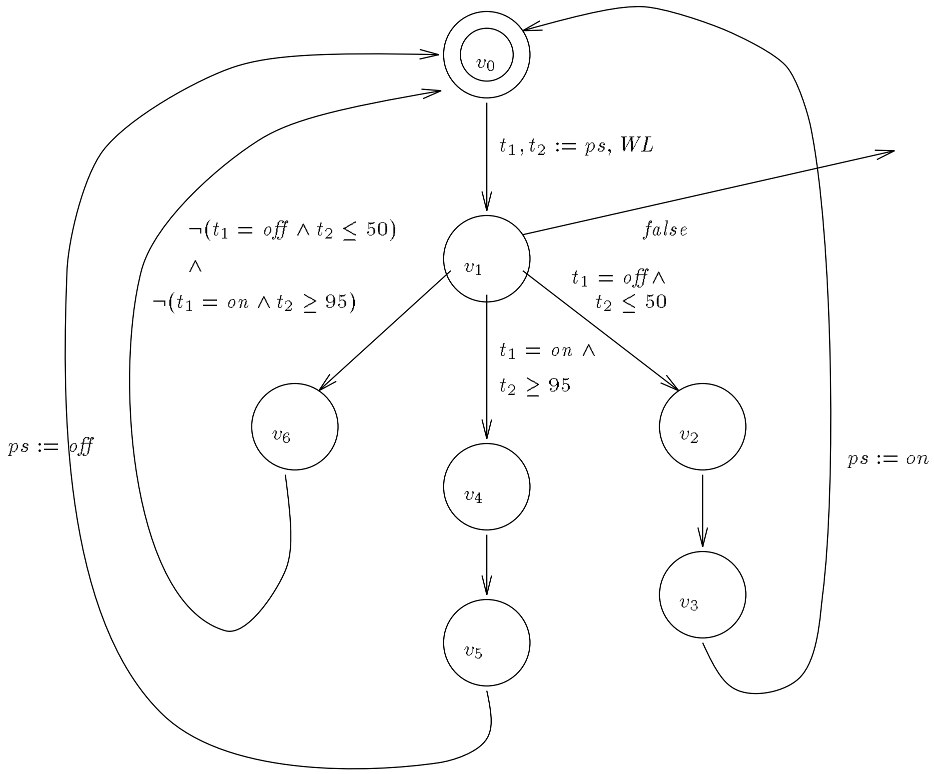

Fig. 2. Control graph of S_1

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 *true*, then we say that node v is *active*.

Finally, *clock variables* capture elapsed time since various control graph nodes were last active. Clock variable *now* records the elapsed time since the program started execution. Clock variable $\uparrow v$ contains the elapsed time since control variable v last changed from *false* to *true* and has value $\perp$ if v has never become *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 *true* to *false*; it has value $\perp$ if v has never been *true*.

Execution of a program is modeled as a sequence of phases [18, 12]. Each *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 *trace* τ is a possibly infinite sequence of phases

$$([r_1, r_1'], f_1), ([r_2, r_2'], f_2), \ldots \tag{3}$$

such that for all i, $r_i' = r_{i+1}$. The *length* $|\tau|$ of a trace τ 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 *prefix*, with $r_i < m \leq r_i'$, of τ, 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.[4] 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 τ 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 $([r_1, r_1'], f_1)$:

- $r_1 = 0$.
- $now = 0$ in state $f_1(r_1)$.
- Exactly the two control variables that correspond to the entry nodes of the control graph for P are *true* in state $f_1(r_1)$.
- If a control variable v is *true* in state $f_1(r_1)$ then clock variable $\uparrow v$ equals 0 in that state. Otherwise $\uparrow v$ equals $\perp$ in state $f_1(r_1)$.
- 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

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

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 $(v, v', g, \overline{x} := \overline{e})$ be in the control graph.[5] We formalize requirements (i) through (iii) by:

---

[4] 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.

[5] If $\overline{x}$ and $\overline{e}$ are empty, then the effect of execution is the same as **skip**.

- 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 $\overline{x}$ in state $f_{i+1}(r_{i+1})$ is equal to the value of the corresponding expression in $\overline{e}$ 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 $\overline{x} := \overline{e}$.
- Clock variable $\uparrow v'$ 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.

$$\begin{tikzcd}
f_1(0) \arrow[r, dashed] & f_1(r_1') \arrow[d] & & \\
& f_2(r_2) \arrow[r, dashed] & f_2(r_2') \arrow[d] & \\
& & f_3(r_3) \arrow[r, dashed] & \ldots
\end{tikzcd}$$

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 $Init \Rightarrow \Box I$ defines the property containing all traces τ such that (i) $Init$ does not hold initially on τ or (ii) I holds throughout τ. Thus, I implies that the "bad thing" being prescribed by the safety property has not happened.

In formula $Init \Rightarrow \Box I$, we call $Init$ 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.

$$\begin{aligned} A \;&:: T\ op_{rel}\ T' \mid \neg A \mid A \wedge A' \mid (\forall x.A : A') \\ T \;&:: \mathrm{C} \mid Var \mid \tfrac{d}{dt}(Var) \mid T\ op_{arith}\ T' \mid T\{T'\} \mid \textstyle\int_T^{T'} T'' \\ Var &:: x \mid X \mid v \mid \ \uparrow v \mid \ \downarrow v \mid now \end{aligned} \tag{4}$$

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 τ inductively.

$$\begin{aligned} \mathcal{M}_T(\mathrm{C})(\tau) &= \mathrm{C} \\ \mathcal{M}_T(Var)(\tau) &= f_i(r_i')(Var), \text{ where } ([r_i, r_i'], f_i) \text{ is the last phase in } \tau \\ \mathcal{M}_T(\tfrac{d}{dt}(Var))(\tau) &= lim_{\triangle r \to 0} \frac{\mathcal{M}_T(Var)(\tau) - \mathcal{M}_T(Var)(\tau_{..|\tau|-\triangle r})}{\triangle r} \\ \mathcal{M}_T(T\ op_{arith}\ T')(\tau) &= \mathcal{M}_T(T)(\tau)\ op_{arith}\ \mathcal{M}_T(T')(\tau) \\ \mathcal{M}_T(T\{T'\})(\tau) &= \mathcal{M}_T(T)(\tau_{..|\tau|-\mathcal{M}_T(T')(\tau)}) \\ \mathcal{M}_T(\textstyle\int_T^{T'} T'')(\tau) &= \textstyle\int_{\mathcal{M}_T(T)(\tau)}^{\mathcal{M}_T(T')(\tau)} \mathcal{M}_T(T'')(\tau_{..t})dt \end{aligned} \tag{5}$$

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

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

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

$$\begin{aligned} \tau \in [[Init \Rightarrow \Box I]] : \mathcal{M}_{\mathcal{A}}(Init)(\tau_{..0}) = false \ \text{ or} \\ (\forall j.0 \le j \le |\tau| :\ \mathcal{M}_{\mathcal{A}}(I)(\tau_{..j}) = true) \end{aligned} \tag{6}$$

5 Verifying Hybrid Systems

When verifying a hybrid system, we are interested in proving that executions of a program P satisfy $[[Init \Rightarrow \Box I]]$ 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 $[[Init \Rightarrow \Box I]]$ 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:

$$\tau \in [[\Box Env]] : \quad (\forall j. 0 \leq j \leq |\tau| : \mathcal{M}_{\mathcal{A}}(Env)(\tau_{..j}) = true) \tag{7}$$

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

$$[[P]] \subseteq [[\mathcal{H}]] \tag{8}$$

$$[[\mathcal{H}]] \subseteq ([[Init \Rightarrow \Box I]] \cup \overline{[[\Box Env]]}) \tag{9}$$

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, γ maps each node v among the nodes V of CG_P to an assertion γ_v satisfying R1 and R2, and $\Box Env$ is a hybrid-environment property. The assertion

$$I_{\mathcal{H}} : \quad \bigwedge_{v \in V} v \Rightarrow \gamma_v \tag{10}$$

is called the *invariant* of $\mathcal{H}$. Property $[[\mathcal{H}]]$ is defined to be the set of all finite and infinite traces τ such that (i) τ 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 $[[P]] \subseteq ([[Init \Rightarrow \Box I]] \cup \overline{[[\Box Env]]})$.

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 $(v, u, g, op) \in E$:*

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

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:*

$$(Env \wedge v \wedge \gamma_v \wedge g \wedge w \wedge \gamma_w \wedge \downarrow v = 0) \Rightarrow (w \wedge \gamma_w)[op, \uparrow u := 0, v := false, u := true]$$

is valid.

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

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

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

The proofs of both theorems are in Appendix B.

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 $\Box Env$ 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 ℓ/sec and a valve that passes 0.25 ℓ/sec. When the valve is closed and the pump is off the water-level does not change:

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

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

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

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

$$(ps = off \wedge vs = open) \Rightarrow \frac{d}{dt}(WL) = -0.25$$

Finally, when the valve is closed and the pump is on, the water-level increases at the rate of 0.5 ℓ/sec:

$$(ps = on \wedge vs = close) \Rightarrow \frac{d}{dt}(WL) = 0.5$$

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

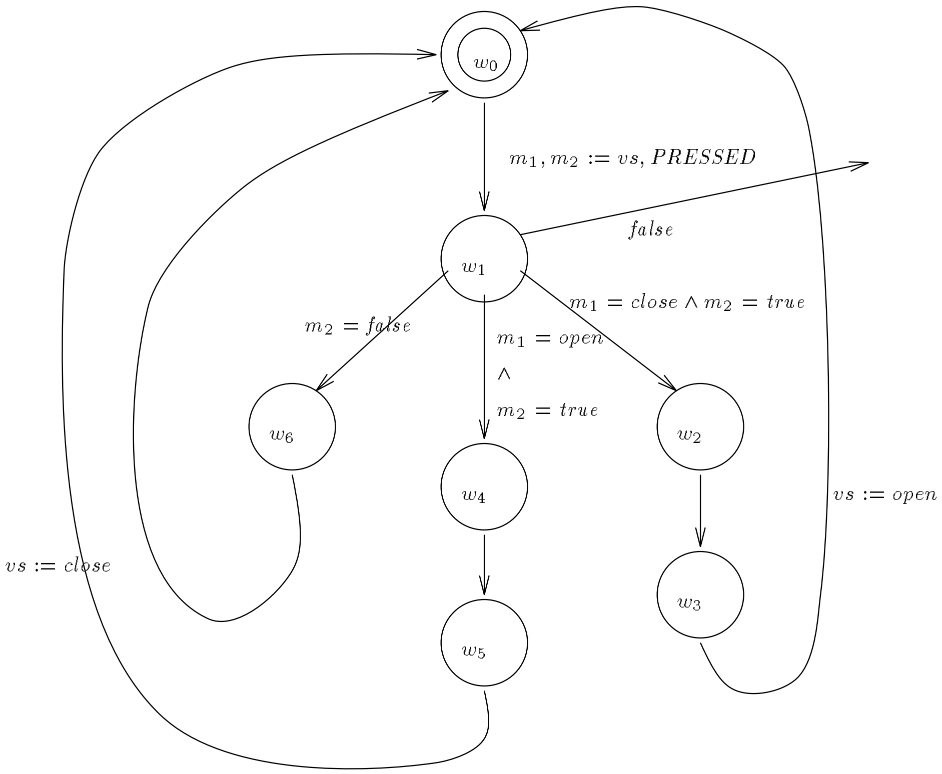

Fig. 3. The control graph of S_2

whatever it was when u first became true plus the change to the water-level since that time:

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

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

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

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 $Init \Rightarrow \Box I$, where:

$$\begin{array}{ll} Init: & ps = \mathit{off} \wedge vs = close \wedge v_0 \wedge w_0 \wedge \neg PRESSED \wedge WL\{\uparrow v_0\} = 90 \\ I: & 48 < WL < 98 \end{array}$$

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

$$\gamma_{w_i} : (v_0 \oplus \ldots \oplus v_6) \wedge (w_0 \oplus \ldots \oplus w_6) \;\; \mathit{for} \;\; i = 0..6$$

$$\begin{aligned} \gamma_{v_0} :& (vs = open \vee vs = close) \wedge \\ & (ps = on \vee ps = \mathit{off}) \wedge \\ & ps = on \Rightarrow 48.5 < WL\{\uparrow v_0\} < 96 \wedge \\ & ps = \mathit{off} \Rightarrow 49.5 < WL\{\uparrow v_0\} < 98 \end{aligned}$$

$$\begin{aligned} \gamma_{v_1} :& (vs = open \vee vs = close) \wedge \\ & (ps = on \vee ps = \mathit{off}) \wedge \\ & ps = on \Rightarrow 48.75 < WL\{\uparrow v_1\} < 96.5 \wedge \\ & ps = \mathit{off} \Rightarrow 49.25 < WL\{\uparrow v_1\} < 98 \wedge \\ & t_1 = ps \wedge t_2 = WL\{\uparrow v_1\} \end{aligned}$$

$$\begin{aligned} \gamma_{v_2} :& (vs = open \vee vs = close) \wedge ps = \mathit{off} \wedge \\ & 49 < WL\{\uparrow v_2\} \leq 50 \end{aligned}$$

$$\begin{aligned} \gamma_{v_3} :& (vs = open \vee vs = close) \wedge ps = \mathit{off} \wedge \\ & 48.75 < WL\{\uparrow v_3\} \leq 50 \end{aligned}$$

$$\begin{aligned} \gamma_{v_4} :& (vs = open \vee vs = close) \wedge ps = on \wedge \\ & 95.25 \leq WL\{\uparrow v_4\} < 97 \end{aligned}$$

$$\begin{aligned} \gamma_{v_5} :& (vs = open \vee vs = close) \wedge ps = on \wedge \\ & 95.5 \leq WL\{\uparrow v_5\} < 97.5 \end{aligned}$$

$$\begin{aligned} \gamma_{v_6} :& (vs = open \vee vs = close) \wedge \\ & (ps = on \vee ps = \mathit{off}) \wedge \\ & ps = on \Rightarrow 49 < WL\{\uparrow v_6\} < 95.5 \\ & ps = \mathit{off} \Rightarrow 49.75 < WL\{\uparrow v_6\} < 96.5 \end{aligned}$$

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

1. M. Abadi and L. Lamport. An old-fashioned recipe for real-time. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Real-time: Theory in Practice*, pages 1–17. Lecture Notes in Computer Science Vol. 600, Springer-Verlag, 1992.
2. R. Alur, C. Courcoubetis, T.A. Henzinger, and P-H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In R.L. Grossman, A. Nerode, A.P. Ravn, and H. Rischel, editors, *Hybrid Systems*, pages 209–229. Lecture Notes in Computer Science Vol. 736, Springer-Verlag, 1993.
3. H. Barringer, R. Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. In *16th Annual ACM Aymposium on Theory of Computing*, pages 51–63, 1984.
4. Z. Chaochen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.
5. Z. Chaochen, A.P. Ravn, and M.R. Hansen. An extended duration calculus for hybrid real-time systems. In R.L. Grossman, A. Nerode, A.P. Ravn, and H. Rischel, editors, *Hybrid Systems*, pages 36–59. Lecture Notes in Computer Science Vol. 736, Springer-Verlag, 1993.
6. E.M. Clarke, D.E. Long, and K.L. McMillan. Compositional model checking. In *Proceedings 4th IEEE Symposium on Logic in Computer Science*, pages 353–362, 1989.
7. J.W. de Bakker, C. Huizing, W.-P. de Roever, and G. Rozenberg (eds.). *Real-Time: Theory in Practice.* Lecture Notes in Computer Science Vol. 600, Springer-Verlag, 1992.
8. E.A. Emerson. Alternative semantics for temporal logics. *Theoretical computer science*, 26:121–130, 1983.
9. L. Fix, N. Francez, and O. Grumberg. Program composition via unification. In *19th International Colloquium On Automata, Languages and Programming*, pages 672–684. Lecture Notes in Computer Science Vol. 623, Springer-Verlag, 1992.
10. L. Fix and F.B. Schneider. Reasoning about programs by exploiting the environment. To appear, 21st International Colloquium On Automata, Languages and Programming.
11. O. Grumberg and D.E. Long. Model checking and modular verification. In *CONCUR'91, 2nd International Conference on Concurrency Theory*, pages 250–263. Lecture Notes in Computer Science Vol. 527, Springer-Verlag, 1991.
12. T.A. Henzinger, Z. Manna, and A. Pnueli. Towards refining temporal specifications into hybrid systems. In R.L. Grossman, A. Nerode, A.P. Ravn, and H. Rischel, editors, *Hybrid Systems*, pages 60–76. Lecture Notes in Computer Science Vol. 736, Springer-Verlag, 1993.
13. J. Hooman. A compositional approach to the design of hybrid systems. In R.L. Grossman, A. Nerode, A.P. Ravn, and H. Rischel, editors, *Hybrid Systems*, pages 121–148. Lecture Notes in Computer Science Vol. 736, Springer-Verlag, 1993.
14. C.B. Jones. Specification and design of (parallel) programs. In R.E.A. Mason, editor, *Information Processing'83*, pages 321–332. Elsevier Science Publishers, 1983.
15. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Engineering*, 3:125–143, 1977.
16. L. Lamport. The hoare logic of concurrent programs. *Acta Informatica*, 14, 1980.

17. L. Lamport. Hybrid systems in tla^+. In R.L. Grossman, A. Nerode, A.P. Ravn, and H. Rischel, editors, *Hybrid Systems*, pages 77–102. Lecture Notes in Computer Science Vol. 736, Springer-Verlag, 1993.
18. O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In J.W. de Bakker, C. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *REX Workshop, Real-time: Theory in practice*, pages 447–484. Lecture Notes in Computer Science Vol. 600, Springer-Verlag, 1992.
19. Z. Manna and A. Pnueli. Verifying hybrid systems. In R.L. Grossman, A. Nerode, A.P. Ravn, and H. Rischel, editors, *Hybrid Systems*, pages 4–35. Lecture Notes in Computer Science Vol. 736, Springer-Verlag, 1993.
20. K. Marzullo, F.B. Schneider, and N. Budhiraja. Derivation of sequential real-time, process-control programs. In A. Van Tilborg and G. Koob, editors, *Foundations of Real-Time Computing*, pages 39–54. Kluwer Academic Publishers, 1991.
21. J. Misra and M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, SE-7(4):417–426, 1981.
22. X. Nicollin, J. Sifakis, and S. Yovine. From atp to timed graphs and hybrid systems. In J.W. de Bakker, C. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *REX Workshop, Real-time: Theory in practice*, pages 549–572. Lecture Notes in Computer Science Vol. 600, Springer-Verlag, 1992.
23. A. Pnueli. In transition from global to modular temporal reasoning about program. In K.R. Apt, editor, *Logics and models of concurrent systems*, pages 123–144. NATO ASI Series, Vol. F13, Springer-Verlag, 1985.
24. F.B. Schneider. Designing real-time systems (that really work!). Invited lecture at Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, Warwick, England, Sept. 1988.

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 $e_0 = (v_0, ?, true, \mathbf{skip})$, and $E_{exit} = \{e_0\}$.
- For S an assignment $x := e(\overline{y})$:[6] Define $V = \{v_0, v_1\}$, $V_{entry} = \{v_0\}$, and $E = \{e_0, e_1\}$ where $e_0 = (v_0, v_1, true, \overline{t} := \overline{y})$, $e_1 = (v_1, ?, true, x := e(\overline{t}))$, and $E_{exit} = \{e_1\}$.
- For S a statement composition $S_1; S_2$: Let $(V^1, E^1, V^1_{entry}, E^1_{exit})$ be the control graph for S_1 and let $(V^2, E^2, V^2_{entry}, E^2_{exit})$ be the control graph for S_2. Define $V = V^1 \cup V^2$, $V_{entry} = V^1_{entry}$, $E_{exit} = E^2_{exit}$, and

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

- For S an iteration **do** $G_1 \rightarrow S_1 [] G_2 \rightarrow S_2$ **od**:[7] Let $\overline{y}$ be the list of variables mentioned by G_1 and G_2. Let $(V^1, E^1, V^1_{entry}, E^1_{exit})$ be the control graph

[6] $e(\overline{y})$ denotes that expression e refers to variables in list $\overline{y}$.
[7] To simplify the presentation, we assume only two alternatives.

for S_1 and let $(V^2, E^2, V^2_{entry}, E^2_{exit})$ be the control graph for S_2. Define $V = V^1 \cup V^2 \cup \{v_0, v_1\}$, $V_{entry} = \{v_0\}$,

$$
\begin{aligned}
E = & E^1 \cup E^2 - E^1_{exit} - E^2_{exit} \cup \\
& \{(v_0, v_1, true, \overline{t} := \overline{y})\} \cup \\
& \{v_1, v, G_1[\overline{t}/\overline{y}], \mathbf{skip}) \mid v \in V^1_{entry}\} \cup \{v_1, v, G_2[\overline{t}/\overline{y}], \mathbf{skip}) \mid v \in V^2_{entry}\} \cup \\
& \{(v_1, ?, \neg G_1[\overline{t}/\overline{y}] \wedge \neg G_2[\overline{t}/\overline{y}], \mathbf{skip})\} \cup \\
& \{(v, v_0, g, op) \mid \exists (v, ?, g, op) \in E^1_{exit} \cup E^2_{exit}\}
\end{aligned}
$$

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

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

B Soundness Proofs

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

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

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

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 τ and at $\tau'_{..j}$ 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 $(v, u, g, op) \in E$:

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

  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:

  $$(Env \wedge v \wedge \gamma_v \wedge g \wedge w \wedge \gamma_w \wedge \downarrow v = 0) \Rightarrow (w \wedge \gamma_w)[op, \uparrow u := 0, v := false, u := true]$$

  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 $[[\mathcal{H}]]$ we need to prove that either, (i) $\tau \notin [[\Box Env]]$, or (ii) $\mathcal{M}_{\mathcal{A}}(I_{\mathcal{H}})(\tau_{..0}) = \mathit{false}$, or (iii) $(\forall j . 0 \leq j \leq |\tau| : \mathcal{M}_{\mathcal{A}}(I_{\mathcal{H}})(\tau_{..j}) = \mathit{true})$. 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 $\mathcal{M}_{\mathcal{A}}(I_{\mathcal{H}})(\tau_{..0}) = \mathit{true}$.
4. By induction on the number i of the phases in τ, we next prove:

$$(\forall j . 0 \leq j \leq |\tau| : \mathcal{M}_{\mathcal{A}}(I_{\mathcal{H}})(\tau_{..j}) = \mathit{true})$$

Basis: $i = 1$. According to 3 we have that $\mathcal{M}_{\mathcal{A}}(I_{\mathcal{H}})(\tau_{..0}) = \mathit{true}$. According to 1 and Lemma 1 we conclude $(\forall j . 0 \leq j \leq r_1^{'} : \mathcal{M}_{\mathcal{A}}(I_{\mathcal{H}})(\tau_{..j}) = \mathit{true})$.
Step: Assume $(\forall j . 0 \leq j \leq r_{i-1}^{'} : \mathcal{M}_{\mathcal{A}}(I_{\mathcal{H}})(\tau_{..j}) = \mathit{true})$ holds for $i > 1$. We prove:

$$(\forall j . r_i < j \leq r_i^{'} : \mathcal{M}_{\mathcal{A}}(I_{\mathcal{H}})(\tau_{..j}) = \mathit{true})$$

(a) $\mathcal{M}_{\mathcal{A}}(Env \wedge I_{\mathcal{H}})(\tau_{..r_{i-1}^{'}}) = \mathit{true}$, by the step assumption and 3.

(b) Without lose of generality, assume $(v_{i-1}, w_{i-1}) \xrightarrow{g,op} (v_i, w_i)$ 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 $I_{\mathcal{H}}$:

$$\mathcal{M}_{\mathcal{A}}(Env \wedge v_{i-1} \wedge w_{i-1} \wedge \gamma_{v_{i-1}} \wedge \gamma_{w_{i-1}} \wedge g \wedge \downarrow v_{i-1} = 0)(\tau_{..r_{i-1}^{'}}) = \mathit{true}$$

(c) Let τ^* 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)

$$\mathcal{M}_{\mathcal{A}}(v_i \wedge w_i \wedge \gamma_{v_i} \wedge \gamma_{w_i})(\tau^*) = \mathit{true}$$

(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:

$$\mathcal{M}_{\mathcal{A}}(v_i \wedge w_i \wedge \gamma_{v_i} \wedge \gamma_{w_i})(\tau^*) = \mathcal{M}_{\mathcal{A}}(v_i \wedge w_i \wedge \gamma_{v_i} \wedge \gamma_{w_i})(\tau_{..j})$$

(e) According to 4(c), 4(d) and definition (10) of $I_{\mathcal{H}}$, $\mathcal{M}_{\mathcal{A}}(I_{\mathcal{H}})(\tau_{..j}) = \mathit{true}$.

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

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

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

Proof:

1. Let $\tau \in [[\mathcal{H}]]$.
2. According to definition of $[[\mathcal{H}]]$, either:
   (a) $\tau \in \overline{[[\Box Env]]}$. In this case, $\tau \in ([[Init \Rightarrow \Box I]] \cup \overline{[[\Box Env]]})$.
   (b) $\tau \in [[\Box Env]]$ and $\mathcal{M}_{\mathcal{A}}(I_{\mathcal{H}})(\tau_{..0}) = \mathit{false}$. In this case, since $(Env \wedge Init) \Rightarrow I_{\mathcal{H}}$ we know $\mathcal{M}_{\mathcal{A}}(Env \wedge Init)(\tau_{..0}) = \mathit{false}$. However, since $\mathcal{M}_{\mathcal{A}}(Env)(\tau_{..0}) = \mathit{true}$ we get $\mathcal{M}_{\mathcal{A}}(Init)(\tau_{..0}) = \mathit{false}$. Therefore according to definition (6), $\tau \in [[Init \Rightarrow \Box I]]$ and thus $\tau \in ([[Init \Rightarrow \Box I]] \cup \overline{[[\Box Env]]})$.

(c) $\tau \in [[\Box Env]]$ and $(\forall j.0 \leq j \leq |\tau| : \mathcal{M}_{\mathcal{A}}(Env \wedge I_{\mathcal{H}})(\tau_{..j}) = true)$. In this case, since by hypothesis $(Env \wedge I_{\mathcal{H}}) \Rightarrow I$ we get $(\forall j.0 \leq j \leq |\tau| : \mathcal{M}_{\mathcal{A}}(I)(\tau_{..j}) = true)$. Thus, $\tau \in [[Init \Rightarrow \Box I]]$ and this implies $\tau \in ([[Init \Rightarrow \Box I]] \cup \overline{[[\Box Env]]})$.

C The verification conditions

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

- $(Env \wedge Init) \Rightarrow I_{\mathcal{H}}$
- $(Env \wedge I_{\mathcal{H}}) \Rightarrow I$

For Theorem 1, we must prove:

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