AEROSPACE REPORT NO.
ATR-92(2778)-9

Timelines and Proofs of Safety Properties in the State Delta Verification System (SDVS)

30 September 1992

Prepared by

L. G. MARCUS and T. K. MENAS
Computer Systsems Division

Prepared for

NATIONAL SECURITY AGENCY
Ft. George G. Meade, MD 20755-6000

Engineering and Technology Group

THE AEROSPACE CORPORATION
El Segundo, California

Aerospace Report No.
ATR-92(2778)-9

TIMELINES AND PROOFS OF SAFETY PROPERTIES IN THE STATE DELTA VERIFICATION SYSTEM (SDVS)

Prepared by

L. G. Marcus and T. K. Menas
Computer Systems Division

30 September 1992

Engineering and Technology Group
THE AEROSPACE CORPORATION
El Segundo, CA 90245-4691

Prepared for

NATIONAL SECURITY AGENCY
Ft. George G. Meade, MD 20755-6000

Report No.
ATR-92(2778)-9

TIMELINES AND PROOFS OF SAFETY PROPERTIES
IN THE STATE DELTA VERIFICATION SYSTEM (SDVS)

Prepared

L. G. Marcus

T. K. Menas

Approved

B. H. Levy, Manager
Computer Assurance Section

D. B. Baker, Director
Trusted Computer Systems Department

C. A. Sunshine, Principal Director
Computer Science and Technology
Subdivision

Abstract

Proofs of typical safety properties of programs in temporal-logic-based systems can be facilitated by the use of two proof rules: the Rule of Negation and the ω-Induction Rule. We show that each of these rules is valid only on timelines of certain order types; the joint use of these two rules is valid only on timelines that are finite, or ordered like the natural numbers.

We demonstrate the use of these rules by giving proofs of safety properties of a simple concurrent program in the State Delta Verification System (SDVS).

Contents

1 Introduction

We consider automated theorem provers and proof checkers based on temporal logic with a linear timeline, in particular in their application to computer verification. The variations in possible temporal systems are manifold. One issue is the choice of the order type of the timeline or timeframe. Most temporal systems assume a timeline that is ordered like the natural numbers (a standard reference is [1] or [2]), though there are occasions where more general timelines have been suggested and used.

The State Delta Verification System [3] (SDVS) is a system whose temporal operators are based on the *weak* versions of the standard temporal operators of $\Box$, $\Diamond$, and $\mathcal{U}$, i.e., all future states may include the present.[1] Most proof commands do not impose any restriction on the underlying timeline. However, we have found that in order for certain safety properties to be valid, we had to restrict the timelines admitted by our logic.

The constraints that certain proof rules impose on the timelines of temporal structures arise in two situations.

First, if the proof of a safety property of a program contains the negation of a temporal formula with "until," then it is often convenient to simplify that formula by using the Negation Rule:

$$\neg(p\ \mathcal{U}\ q) \equiv [\Box\neg q \vee \neg p \vee (\neg q\ \mathcal{U}\ (\neg p \wedge \neg q))]$$

Since this rule is valid on precisely well-ordered timelines (Theorem 3), the proof of such a safety property could very well be valid only for temporal structures with well-ordered timelines. Such a property will be given in Section 5 (Theorem 6). The importance of this formula is that it "pushes" the negation of an *until* formula inside another *until* formula. Successive applications of this rule result in an equivalent formula in which no *until* operator is in the scope of a negation. In an important sense, this reduction is impossible for linear timelines that are not well-ordered.

Second, the proof of even a simple safety property of a nonterminating program may require an ω-Induction Rule of the form

$$[(\alpha \wedge \beta) \wedge \Box(\alpha \wedge \beta \rightarrow \exists a_0 \ldots \exists a_{n-1}\ (\bigwedge_{i<n} x_i = a_i\ \wedge\ (\alpha\ \mathcal{U}\ (\alpha \wedge \beta \wedge \bigvee_{i<n} x_i \neq a_i))))] \rightarrow \Box\alpha$$

where $\{x_i : i < n\}$ is the finite set of program variables.[2]

[1] $(M,t) \models p\ \mathcal{U}\ q$ is defined to hold *iff* $(M,t) \models p$ and there is $r \geq t$ such that $(M,r) \models q$ and for all s, if $t \leq s < r$, then $(M,s) \models p$. Notice that even if $r = t$, p must hold at t.

[2] This rule might seem strange to those familiar with the more standard temporal logic induction rules, such as

$$(\alpha \wedge \Box(\alpha \rightarrow Next\alpha)) \rightarrow \Box\alpha.$$

First of all, the weakness of the *until* means that there is no definable *Next* operator; also, the weak *until* means that to guarantee "progress" we must explicitly include the conjunct representing the claim that some

Consider, for example, the program P that initially assigns to its only variable x the value zero and thereafter repeatedly decrements the value of x by one. Clearly one safety property, S, that this program satisfies is that at every time in the future, the value of x will be less than or equal to zero. However, a "natural" translation of this program into temporal logic, for example the translation into the formula simply asserting "x is equal to zero now and at every time in the future the value of x will be discretely decremented by one,"[3] will be true in the temporal structure whose timeline is $\omega + \omega$ and in which the values of x are

$$0, -1, -2, \ \ldots \ , 1, 0, -1, -2, \ \ldots$$

However, S fails to be true in this model; the problem is the limit point t_ω of the timeline at which the value of x is 1. Note that $\omega + 1$ is embeddable in this timeline.

It is easy to prove that the ω-Induction Rule above is valid for precisely those temporal structures in whose timeline $\omega + 1$ is not embeddable (Theorem 4).

Thus, to recapitulate, the use of the Negation Rule implies that the timeline is well-ordered, and the use of the ω-Induction Rule implies that the timeline does not embed $\omega + 1$. Thus, the use of both implies that the timeline is either finite or isomorphic to ω.

We give examples of two safety properties and their proofs in SDVS that use the above rules. Here we will be concerned only with understanding enough about SDVS in order to translate the results of Sections 2 and 3 into the logic and proof commands of SDVS. The logical formulas of SDVS are called *state deltas*, which are written in a precondition-postcondition style, but correspond roughly to temporal logic with weak box ($\Box$), diamond ($\Diamond$), and until ($\mathcal{U}$). This weak semantics was chosen because we wanted the truth of state delta formulas to be preserved under "stuttering," i.e., through a time interval where no values of local variables change.

The particular syntax of state deltas was chosen to facilitate intuitive proofs of program correctness by symbolic execution. The reader is referred to [4] for the exact correspondences.

The Negation Rule is incorporated in SDVS as the proof command **negate** and the ω-Induction Rule as the proof command **omegainduct** (Section 6). It thus follows that proofs in SDVS+**negate**+**omegainduct** are valid only for temporal structures whose timelines are either finite or isomorphic to ω.

What restriction on the *applicability* of such a safety proof does the timeline restriction impose? There are examples where, in order to represent faithfully a computation, it has been suggested to assume a non-ω timeline. In such cases, a proof of safety must utilize other rules. For examples see [5] and [6], or [7].

Now we define the structures over which we interpret temporal formulas. We are given

program variable has changed value. Second, the above rule without β (i.e., with $\beta = \mathit{true}$) is weaker in our proof system. We use β in the proofs of safety properties of loops, specifically by letting β encode the claim that execution is at the "top of the loop."

[3] Of course, there is another translation of P that by the above usage would have to fall into the "unnatural" category; this translation obviates the need for induction: add the sentence saying that x never increases.

a first-order structure that can be thought of as the domain of values, a set of variables ranging over those values, and a finite set of variables that can be thought of as the set of program variables.

Definition 1 *Let $\mathcal{A}$ be a first-order structure,* GlobalVars *a set, and* ProgramVars *a finite set. A first-order temporal structure M with base $\mathcal{A}$ is a triple $\langle (T, \leq), \Sigma, \sigma \rangle$ such that*

- *$(T, \leq)$, the timeline, is a linearly ordered set with a least element, usually denoted by t_0;*
- *$\Sigma : GlobalVars \rightarrow\mid \mathcal{A} \mid$; and*
- *$\sigma : T \times ProgramVars \rightarrow\mid \mathcal{A} \mid$.*

We assume the normal syntax for first-order temporal logic with propositional operators $\wedge$, $\vee$, $\neg$, and the "weak future" semantics for the temporal operator $\mathcal{U}$, as described above. The operators (weak) $\Box$ and (weak) $\Diamond$ are defined in terms of $\mathcal{U}$ in the standard manner, e.g.

$$(M,t) \models \Diamond q \;\equiv\; (M,t) \models (true\ \mathcal{U}\ q) \;\equiv\; \exists t' \geq t (M,t') \models q$$

We refer to this language as Weak Propositional Temporal Logic, WPTL. The state delta language and an intermediate language will be introduced in Section 4.

Notation: If Θ is an interval of the timeline T, then $M \cap \Theta \models \phi$ is defined by $\forall t \in \Theta\ (M,t) \models \phi$.

2 Negation

In this section we show that negations can be pushed inside WPTL formulas if and only if the timeline is well-ordered. Thus, using the Negation Rule

$$\neg(p\ \mathcal{U}\ q) \equiv [\Box\neg q \vee \neg p \vee (\neg q\ \mathcal{U}\ (\neg p \wedge \neg q))]$$

entails an assumption that the timeline is in fact well-ordered.

First we show that the above Negation Rule is not valid in general, and that no similar rule for simplifying negations is valid.

Definition 2 *A WPTL formula is positive if no temporal operator is in the scope of* $\neg$.

Theorem 1 *There is a WPTL formula* α *that is not equivalent to any positive WPTL formula.*

Note: Without $\mathcal{U}$ all WPTL formulas are equivalent to positive formulas.

Proof: Let $\alpha = \neg(\neg p\ \mathcal{U}\ p)$, i.e., that there is no first time at which p is true. We will define a temporal structure (over a non-well-ordered timeline) in which α is not equivalent to any positive formula.

Let M_1 be the temporal structure over timeline $\langle t_0, \ldots, s_i \ldots\rangle$ where i ranges over the negative integers, of order type $1+\omega^*$ (ω^* is the order type of the negative integers.) where p is true at t iff $t > t_0$, and N_1 be the temporal structure over $\langle t_0, t_1\rangle$ where p is defined similarly (i.e., false at t_0 and true at t_1.)

Note that $(M_1, t_0) \models \alpha$ and $(N_1, t_0) \models \neg\alpha$.

Let $\Gamma = \{\gamma : \gamma$ *is a WPTL formula such that* $(M_1, t_0) \models \gamma \rightarrow (N_1, t_0) \models \gamma\}$.

The theorem will follow if we show that Γ contains all the positive formulas, as implied by the following lemma:

Lemma 1 *The following facts are true of* Γ*:*

1. Γ *contains all static (containing no temporal operator) formulas .*
2. Γ *is closed under* $\wedge$ *and* $\vee$.
3. Γ *is closed under* $\Box$ *and* $\Diamond$
4. Γ *is closed under* $\mathcal{U}$.

Proof of Lemma: The first two items are obvious. The next one follows from:

Claim: For every WPTL formula τ and every negative integer i,

$$(M_1, s_i) \models \tau \rightarrow (N_1, t_1) \models \tau$$

The proof is a simple induction on the complexity of τ; it also is a trivial consequence of Corollary 3 in [4].

Now consider fact 4. Let $(M_1, t_0) \models \gamma \mathcal{U} \delta$ where $\gamma, \delta \in \Gamma$. If $(M_1, t_0) \models \delta$, we are through. Otherwise, $(M_1, s_i) \models \delta$ for some i and $M_1 \cap [t_0, s_i) \models \gamma$.

Thus, $(N_1, t_1) \models \delta$ by the above claim and $(N_1, t_0) \models \gamma$ (Lemma 1 and Theorem 1).⊣

The above theorem can be strengthened as follows:

Theorem 2 *There is a WPTL formula α such that for every timeline T that has an initial element but is not well-ordered, α is not equivalent to a positive formula over T.*

Proof: Let α be as before, and let t_0 be the initial element of T. Since T is not well-ordered, there is a decreasing sequence $\ldots > r_i > r_{i+1} > \ldots$. Let $J = \{t \in T : (\exists i)\ t \geq r_i\}$ and $I = T - J$.

Define M_2 over T by $(M_2, t) \models p$ *iff* $t \in I$. Let $t_1 > t_0$ and define N_2 by $(N_2, t) \models \neg p$ *iff* $t < t_1$.

Note that $(M_2, t_0) \models \alpha$ and $(N_2, t_0) \models \neg\alpha$.

By Theorem 3 of [4], for every formula γ, $(N_2, t_0) \models \gamma$ if and only if $(N_1, t_0) \models \gamma$. And likewise $(M_2, t_0) \models \gamma$ if and only if $(M_1, t_0) \models \gamma$. Thus, the desired result follows from Lemma 1. ⊣

Theorem 3 *The Negation Rule is true if (and only if) the timeline is well-ordered.*

Note that the "only if" part follows from Theorem 1.

Proof:

The right-to-left implication of the Negation Rule is always true.

Now assume T is well-ordered with initial element t_0, M is a temporal structure over T, and $(M, t_0) \models \neg(p \mathcal{U} q)$. If $(M, t_0) \models \Box\neg q$ or $(M, t_0) \models \neg p$, we are done.

So assume otherwise: let $(M, t_0) \models p$, $t_1 \geq t_0$ be the least such that $(M, t_1) \models q$. In fact, $t_0 < t_1$, otherwise $(M, t_0) \models p \mathcal{U} q$. Thus, $M \cap [t_0, t_1) \models \neg q$. Let $t_2 > t_0$ be the least such that $(M, t_2) \models \neg p$. Thus, $t_0 < t_2 < t_1$, otherwise, again, $(M, t_0) \models p \mathcal{U} q$. Thus, $(M, t_2) \models \neg p \wedge \neg q$, $M \cap [t_0, t_2) \models \neg q$, so $(M, t_0) \models (\neg q \mathcal{U} (\neg p \wedge \neg q))$. ⊣.

3 ω-Induction

In this section we show that the ω-Induction Rule is valid if and only if the timeline does not embed $\omega + 1$. Thus, the use of this rule entails the assumption that in fact the timeline does not embed $\omega + 1$.

The ω-Induction Rule states that

1. if α (the "always" formula) and β (the "auxiliary" formula) are true now,
2. and it is always true in the future that if α and β are true, then α is true until some variable has changed value and α and β are true again,
3. then α will be always true in the future.

Formally, the ω-Induction Rule is given by the following sentence:

$$[(\alpha \wedge \beta) \wedge \Box(\alpha \wedge \beta \rightarrow \exists a_0 \ldots \exists a_{n-1} \, (\bigwedge_{i<n} x_i = a_i \; \wedge \; (\alpha \, \mathcal{U} \, (\alpha \wedge \beta \wedge \bigvee_{i<n} x_i \neq a_i))))] \rightarrow \Box\alpha$$

where $\{x_i : i < n\}$ is the finite set of program variables.

Theorem 4 *The ω-Induction Rule is valid over T* iff *T does not embed $\omega + 1$.*

Proof:

$\leftarrow$: Assume T does not embed $\omega + 1$ and let M be a temporal structure over T such that (M, t_0) satisfies the antecedent of the ω-Induction Rule. Thus, there are $t_i \in T$, $t_i < t_{i+1}$, where $M \cap [t_i, t_{i+1}] \models \alpha$ and $(M, t_i) \models \beta$. If the conclusion of the ω-Induction Rule is not true, i.e., if $(M, t_0) \not\models \Box\alpha$, then there is some t_ω such that $(M, t_\omega) \models \neg\alpha$. Since $t_i < t_\omega$ for all i, the sequence $\{t_0, \ldots, t_i, \ldots, t_\omega\}$ constitutes an embedding of $\omega+1$ into T, a contradiction.

$\rightarrow$: Assume T embeds $\omega + 1$, and let $t_0 < t_1 < \ldots < t_\omega$ be the image of such an embedding. Define M over T such that M has one local variable x whose values are $(M, t) \models x = -i$ for $t \in [t_i, t_{i+1})$, $i < \omega$, and $x = 1$ elsewhere, including t_ω. Now for $\alpha = [x \leq 0]$ and $\beta = \mathit{true}$, we get that the proof rule is not true in M. $\dashv$

4 Syntax and Semantics of SDVS

In this section we give a brief introduction to the syntax and the semantics of SDVS. For the reader who is acquainted with the usual temporal logic notation, we also introduce an intermediate language between the language of SDVS, $L_{SD}[I]$, and the language of classical first-order temporal logic. This intermediate language is $L_{(\#,.)}(\mathcal{U})$. Since $L_{SD}[I]$ and $L_{(\#,.)}(\mathcal{U})$ differ only in their temporal operator, in the definitions that follow we denote both languages by L.

4.1 Syntax

Alphabet

The alphabet of L contains the following symbols:

- for every n-ary function f of $\mathcal{A}$, an n-ary function symbol $\bar{f}$;
- for every n-ary predicate p of $\mathcal{A}$, an n-ary predicate symbol $\bar{p}$;
- the binary predicate symbol $=$;
- the propositional constant **true**;
- the symbols $\neg$, $\wedge$, (,), . , #, and $\forall$;
- for $L_{(\#,.)}(\mathcal{U})$, the temporal operator symbol $\mathcal{U}$, *until*;
- for $L_{SD}[I]$, the temporal operator symbol $\rightsquigarrow$, *state delta*;
- a *finite* set, $ProgramVars$, of program variables (local variables); and
- a denumerable set, $GlobalVars$, of global variables.

Terms

The terms of L are defined by induction:

- Every global variable is a term.
- For every program variable x, $.x$ and $\#x$ are terms.
- Every 0-ary function symbol is a term.
- If $t_1, \ldots, t_n$ are terms and $\bar{f}$ is an n-ary function symbol, then $\bar{f}(t_1 \ldots t_n)$ is a term.
- All terms arise by application of the above four clauses.

Note that unadorned program variables are not terms.

Formulas

The atomic formulas of L consist of the propositional constant **true** and any string of the form $\bar{p}(t_1 \ldots t_n)$, where $\bar{p}$ is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms. The other formulas are defined inductively.

- Every atomic formula of L is a formula of L.
- If A and B are formulas of L, then $(A \wedge B)$ and $\neg A$ are formulas of L.
- If A is a formula of L and v is a global variable, then $\forall v A$ is a formula of L.
- If A and B are formulas of $L_{(\#,.)}(\mathcal{U})$, then $(A \; \mathcal{U} \; B)$ is a formula of $L_{(\#,.)}(\mathcal{U})$.
- If A, B, and C are formulas of $L_{SD}[I]$, then $(A \; {}_{x_1 \ldots x_n}\!\overset{C}{\rightsquigarrow}{}_{y_1 \ldots y_k} B)$ is a formula of $L_{SD}[I]$, where $x_1, \ldots, x_n$ and $y_1, \ldots, y_k$ are (possibly empty) strings of program variables. This formula is a state delta with precondition A, postcondition B, and invariant C.

Note that in the SDVS transcripts in Section 6.3 a state delta of the above form will be written as

[sd pre: A
  comod: $x_1 \ldots x_n$
  mod: $y_1 \ldots y_k$
  inv: C
  post: B]

Any degenerate fields are simply omitted, e.g. if the *comod* is empty or the *inv* is *true.*

4.2 Semantics

Let M be a temporal structure (recall Definition 1.) M determines an evaluation V^M that, for every pair of times t_1 and t_2 of T such that $t_1 \leq t_2$, maps every term τ of L to an element $V^M(t_1, t_2, \tau)$ of the universe of $\mathcal{A}$, and every formula A of L to a truth value $V^M(t_1, t_2, A)$ of the boolean algebra $< \{\mathbf{t}, \mathbf{f}\}; \wedge, \vee, \neg >$.

Evaluation of Terms

Base Case: For every global variable v, $V^M(t_1, t_2, v) = \Sigma(v)$. For every program variable x, $V^M(t_1, t_2, .x) = \sigma(t_1, x)$ and $V^M(t_1, t_2, \#x) = \sigma(t_2, x)$.

Step Case: For every term $\bar{f}(\tau_1 \ldots \tau_n)$,

$$V^M(t_1, t_2, \bar{f}(\tau_1 \ldots \tau_n)) = f(V^M(t_1, t_2, \tau_1) \ldots V^M(t_1, t_2, \tau_n))$$

Evaluation of Formulas

Base Case: For the propositional constant **true**, $V^M(t_1, t_2, \mathbf{true}) = \mathbf{t}$. For every other atomic formula $\bar{p}(\tau_1 \ldots \tau_n)$,

$$V^M(t_1, t_2, \bar{p}(\tau_1 \ldots \tau_n)) = p(V^M(t_1, t_2, \tau_1) \ldots V^M(t_1, t_2, \tau_n))$$

Step Case: Let A be a formula of $L_{(\#,.)}(\mathcal{U})$. We assume that $V^{M^*}(t_1', t_2', D)$ has been defined

- for every proper subformula D of A,
- for every pair of times t_1' and t_2' of T such that $t_1' \leq t_2'$; and
- for every temporal structure $M^* = << T, \leq >, \Sigma^*, \sigma >$, where Σ^* is any evaluation of the global variables of L.

We now consider the various cases for A.

- If $A = (B \wedge C)$, then $V^M(t_1, t_2, A) = V^M(t_1, t_2, B) \wedge V^M(t_1, t_2, C)$.
- If $A = \neg B$, then $V^M(t_1, t_2, A) = \neg V^M(t_1, t_2, B)$.
- If $A = \forall v B$, then $V^M(t_1, t_2, A) = \mathbf{t}$ iff for every

  $$\Sigma^* : GlobalVars \rightarrow\mid \mathcal{A} \mid$$

  such that $\Sigma^*(w) = \Sigma(w)$ for every global variable $w \neq v$, if

  $$M^* = << T, \leq >, \Sigma^*, \sigma >$$

  then $V^{M^*}(t_1, t_2, B) = \mathbf{t}$. Otherwise, $V^M(t_1, t_2, A) = \mathbf{f}$.
- If $A = (B \mathcal{U} C)$, then $V^M(t_1, t_2, A) = \mathbf{t}$ iff $V^M(t_2, t_2, B) = \mathbf{t}$ and there is a t_3 in T such that $t_3 \geq t_2$, $V^M(t_2, t_3, C) = \mathbf{t}$, and for all t^* in $[t_2, t_3)$, $V^M(t_2, t^*, B) = \mathbf{t}$. Otherwise, $V^M(t_1, t_2, A) = \mathbf{f}$.
- If $A = (B \ {}_{x_1 \ldots x_n}\!\!\downarrow\!\!\rightarrow_{y_1 \ldots y_k} C)$, then $V^M(t_1, t_2, A) = \mathbf{t}$ iff for every $t_3 \geq t_2$ such that the values of the program variables $x_1, \ldots, x_n$ remain constant in the time interval $[t_2, t_3]$ and $V^M(t_3, t_3, B) = \mathbf{t}$, there is a time $t_4 \geq t_3$ such that only the program variables $y_1, \ldots, y_k$ may change their value in the time interval $[t_3, t_4]$, and
  - $V^M(t_3, t_4, C) = \mathbf{t}$,
  - $V^M(t_3, t_3, I) = \mathbf{t}$,
  - for every time t in $[t_3, t_4)$, $V^M(t_3, t, I) = \mathbf{t}$.

Definition 3 *Let A be a formula of L, M a temporal structure, and $t_1 \leq t_2$ a pair of times of T. Then $M(t_1, t_2) \models A$ iff $V^M(t_1, t_2, A) = \mathbf{t}$.*

4.3 Definable Extensions of L

The boolean connectives $\vee$ and $\rightarrow$, and the existential quantifier $\exists$, are defined for L in the usual way. Henceforth, we assume that *all* is a canonical string consisting of all program variables $x_1 \ldots x_n$, and if $s = y_1 \ldots y_n$ is any string of program variables, then there is a canonical string, $-s = z_1 \ldots z_m$, of program variables that lists the program variables that do not appear in s.

Definition 4 *Let A be a formula of L, $s = y_1 \ldots y_n$ be any string of program variables, and $all - s = z_1 \ldots z_m$.*

- $\Diamond_{y_1 \ldots y_k} A = (((\#z_1 = .z_1 \wedge \ldots \wedge \#z_m = .z_m))\ \mathcal{U}\ ((\#z_1 = .z_1 \wedge \ldots \wedge \#z_m = .z_m) \wedge A))$[4] *for the language $L_{(\#,.)}(\mathcal{U})$, and*
- $\Diamond_{y_1 \ldots y_n} A = (true\ \mathcal{U}_{y_1 \ldots y_n}\ A)$ *for the language $L_{SD}[I]$.*
- $\Diamond A = \Diamond_{all} A = \Diamond_{x_1 \ldots x_n} A$,
- $\Box_{y_1 \ldots y_n} A = \neg \Diamond_{y_1 \ldots y_n} \neg A$, *and*
- $\Box A = \neg \Diamond \neg A$.

The semantics for these extensions are, of course, fixed by their definitions, but they are what one would expect. For example, if M is a temporal structure and $t_1 \leq t_2$ are times in T, then $M(t_1, t_2) \models \Box_{y_1 \ldots y_n} A$ iff for every $t_3 \geq t_2$, if the value of every program variable that is not in the set $\{y_i : 1 \leq i \leq n\}$ remains constant in the time interval $[t_2, t_3]$, then $M(t_2, t_3) \models A$.

A formula A of L *has an upper-level dot (pound)* if it contains an occurrence of a term of the form $.x$ $(\#x)$ that is not in the scope of a temporal operator of L. We note the following simple fact:

If a formula A of L has no upper-level dots, then for every temporal structure M and times $t_1 \leq t_2$ of T,

$$M(t_1, t_2) \models A \leftrightarrow M(t_2, t_2) \models A$$

In this case we write $(M, t_2) \models A$.

We also note without proof that $L_{SD}[I]$ is equivalent to $L_{(\#,.)}(\mathcal{U})$ [4].

[4] If $m = 0$, we take the invariant of the *until* to be *true* and the "eventually" formula to be A.

5 Proofs of Safety in SDVS

In this section we give a simple concurrent program, its translation, and the statement of two safety properties in the intermediate language $L_{(\#,.)}(\mathcal{U})$. The proof commands and transcripts of the proofs will be included in the next section.

This example is drawn from [8]; it was analyzed in [9] and proved in [10].

Program F
declare x : integer
declare y : integer
loop
assign
$(y := -y \text{ if } x \leq 0 \wedge y > 0) \parallel (x := x - 1)$
end {F}

The variables x and y have some input value at the time this parallel program begins to execute. The symbol $\parallel$ separating the parallel branches has the (liveness) semantics that both branches get executed infinitely often, at "random" times, and perhaps simultaneously. This requires slightly more care: we mean that for each time when one branch gets executed, there is a later time when the other branch gets executed. There is no requirement about the relative frequency of execution of the two branches over the space of all possible computations.

Looking at the left-most parallel component, that involving y, note that the above condition on execution guarantees only that the *test* will be evaluated at arbitrary times; if the test is true, then at some later time the assignment to y will be performed. It is not necessarily the case that at every time, if the test is true at that time, then the assignment will be made at some later time.

Also, the (safety) semantics determine that the following two properties hold:

- If y is ever < 0, then $y < 0$ thereafter.
- x is weakly decreasing, i.e., the value of x is never greater than it was at a previous time.

Our translation of the program is the conjunction of S_1, S_2, and S_5 below, where

$$S_0 : (\#x \leq 0 \wedge \#y > 0) \rightarrow (\#y = .y \; \mathcal{U} \; \#y = -.y)$$

$$S_1 : \; \Box\Diamond_x S_0$$

$$S_2 : \; \Box(\#x = .x \; \mathcal{U} \; \#x = .x - 1)$$

$$S_5 : \; \Box\neg([\neg(\#x \leq 0 \wedge \#y > 0) \vee \neg S_0] \; \mathcal{U} \; \#y \neq .y)$$

S_1 describes when y changes, and S_2 does the same for x. S_5 regulates the change in y by forcing any change in y to be *due* to S_0. No similar requirement need be placed on x.

The safety property that "$y \neq 0$ is stable" can be represented as

$$S_6 : \Box(\#y \neq 0 \rightarrow \Box\#y \neq 0)$$

and similarly for $y < 0$

$$S_7 : \Box(\#y < 0 \rightarrow \Box\#y < 0)$$

The fact that "x never increases" can be written as

$$S_8 : \Box(\#x \leq a \rightarrow \Box\#x \leq a)$$

or equivalently,

$$\Box\Box\#x \leq .x$$

(however, the former gives a more direct translation to state deltas).

The theorem representing the safety properties of the above program is

Theorem 5 $S_1 \wedge S_2 \wedge S_5 \rightarrow S_6 \wedge S_7 \wedge S_8$

We discuss only two parts:

Theorem 6 $S_2 \rightarrow S_8$

and

Theorem 7 $S_5 \rightarrow S_7$

The proof of S_6 is more difficult and will not be considered here.

The correspondences between S_2, S_5, S_7, and S_8 and their state delta representations $s2, s5, s7$ and $s8$ are given below.

```
s2:

[sd pre: (true)
    mod: (all)
    inv: (#x = .x)
   post: (#x = .x - 1)]


s5:

[sd pre: (true) post: (~(formula(p1)))]
```

```
p1:

[sd pre: (true)
  comod: (all)
    mod: (all)
    inv: (~(#x le 0 & #y gt 0) or ~(formula(s0)))
   post: (~(#y = .y))]


s0:

[sd pre: (.x le 0,.y gt 0)
  comod: (x,y)
    mod: (x,y)
    inv: (#y = .y)
   post: (#y = -.y)]


s7:

[sd pre: (.y lt 0) post: (formula(q1))]

q1:
[sd pre: (true) post: (#y lt 0)]

s8:

[sd pre: (.x le a)
   post: (formula(x.always.le.a))]


x.always.le.a:

[sd pre: (true) post: (#x le a)]
```

6 SDVS Proof Commands and Transcripts

In this section we give the SDVS proof commands corresponding to the Negation Rule and the ω-Induction Rule, and give the transcript of the SDVS proof of Theorems 6 and 7. We do not intend the transcripts to be totally intelligible, since full explanation of all that is involved would take us too far afield. They are presented here more as a "proof of existence." The *SDVS Users' Manual* [?] contains all the details needed to follow the proof traces.

6.1 The Negate Command

In this section we note the circumstances in which the **negate** command is invoked and the results of its invocation. Suppose that at a certain stage of a proof $\neg S$ is known to be true by the system, where S is the state delta

$$(p \ {}_{c}\overset{I}{\rightsquigarrow}_{m} q)$$

Then upon the user's invocation of the **negate** command with S as its argument, SDVS prompts the user for the names of the three formulas that it will create and insert in the postcondition of the negated state delta. Specifically, SDVS will create and assert the following state delta S^*:

$$(true \ {}_{all}\!\!\rightsquigarrow_{-c}(p[\#/.] \wedge (formula(or1) \vee formula(or2) \vee formula(or3))$$

where

- $formula(or1) \equiv (True \ {}_{-m}\!\!\rightsquigarrow_{\emptyset} \neg q^*)$
- $formula(or2) \equiv \neg I[\#/.]$
- $formula(or3) \equiv (True \ {}_{all}\!\!\rightsquigarrow^{q}_{\ m}(\neg I \wedge \neg q))$
- "or1", "or2", and "or3" are the names for the formulas given by the user;
- for any formula s, $s[\#/.]$ is obtained from s by replacing all upper-level dotted places in s by the corresponding #'s; and
- q^* is obtained from q by replacing all upper-level dotted places in q by their symbolic values. For example, if $q \equiv ((\#x = .y + 1) \wedge \sigma)$, where σ is a state delta, then q^* would have the form $((\#x = y123 + 1) \wedge \sigma)$, where $y123$ is the symbolic value of $.y$.

The state delta S^* asserts that there is a future time t_1 such that the value of every place in c remains constant between now and t_1, p is true at t_1, and either

- q is false at every time $t \geq t_1$ such that the value of every place in the complement of m remains constant in the interval $[t_1, t]$, or

- the invariant I is false at t_1, or
- there is a time $t_2 \geq t_1$ such that the value of every place in the complement of m remains constant in the interval $[t_1, t_2]$, the invariant I is false at t_2, and q is false in the interval $[t_1, t_2]$.

6.2 The Omegainduct Command

If **omegainduct** is used in the course of a proof, the user must enter the "always formula," α, and the "auxiliary formula," β, as parameters. The "always formula" α is the formula that will be asserted to be henceforth true. The purpose of the "auxiliary formula" is to allow the induction to proceed over loop bodies that are generated by the SDVS program translators. In these cases, the "auxiliary formula" is intended to be the state delta that asserts that execution is at the top of the loop. The form of the state deltas that are generated by the translators must be altered to allow proofs that involve the **omegainduct** command in these circumstances. This capability has not yet been implemented in SDVS and will not be discussed in this paper. If the user does not enter an auxiliary formula, the system assumes that the formula is "true."

After the "always" and "auxiliary" parameters have been added, SDVS opens the proof of the base-case of the induction, $(true \ {}_{all}\!\!\sim\!\!\!\!\rightarrow_{\emptyset} \alpha \wedge \beta)$. Once the user proves the base case state delta, SDVS opens the proof of the step-case state delta

$$(true \ {}_{\emptyset}\!\!\sim\!\!\!\!\rightarrow_{\emptyset} \rho)$$

where ρ is the state delta

$$(\alpha \wedge \beta \ {}_{all}\!\!\sim\!\!\!\!\xrightarrow{I}_{all} \alpha[\#/.] \wedge \beta[\#/.])$$

and where I is $\alpha[\#/.]$. Once the the step-case state delta has been proved as well, SDVS asserts the state delta $(true \ {}_{\emptyset}\!\!\sim\!\!\!\!\rightarrow_{\emptyset} \alpha[\#/.])$ at the state at which the **omegainduct** command was given.

6.3 Transcripts of Proofs

In this section we present the proofs of the two safety properties. The following transcripts include the proof commands (marked with asterisks), interspersed with query commands for readability.

First the proof of Theorem 6, called *safety1* below, using **negate**.

```
safety1:

 [sd pre: (formula(s5),covering(all,x,y))
    post: (formula(s7))]



* sdvs.1 prove
sd: safety1
proof[]:

 open - [sd pre: (formula(s5),covering(all,x,y))
         post: (formula(s7))]

 Complete the proof.


* sdvs.1.1 prove
sd: s7
proof[]:

   open - [sd pre: (.y lt 0)
           post: (formula(q1))]

 Complete the proof.

sdvs.1.1.1 simp
expression: .y lt 0

 true

sdvs.1.1.1 usable

 u(1) [sd pre: (true)
       post: (~(formula(p1)))]


 No usable quantifiers.

* sdvs.1.1.1 apply
sd[highest applicable]:
proof[]:

    apply - [sd pre: (true)
             post: (~(formula(p1)))]

* sdvs.1.1.1 negate
state delta : p1
formula name #1: dis1
formula name #2: dis2
formula name #3: dis3

    inserting negated state delta -
          [sd pre: (true)
            comod: (all)
              mod: (diff(all,all))
             post: (#y = y6166,true,

                  ([sd pre: (true)
                     comod: (diff(all,all))
                      post: (~((~(#y = y6166))))])

                 or

                  ~(~(#x le 0 & #y gt 0) or
                 ~([sd pre: (.x le 0,.y gt 0)
                     comod: (x,y)
                       mod: (x,y)
```

```
                    inv: (#y = .y)
                    post: (#y = -.y)]]))

                or

                 ([sd pre: (true)
                   comod: (all)
                    mod: (all)
                    inv: (~((~(#y = .y))))
                    post: (~(~(#x le 0 & #y gt 0) or
                          ~(formula(s0))),
                          ~((~(#y = .y))))]])]
sdvs.1.1.2 pp
this: dis1

 formula dis1: [sd pre: (true)
               comod: (diff(all,all))
                post: (~((~(#y = y6166))))]

sdvs.1.1.2 pp
this: dis2

 formula dis2: ~(~(.x le 0 & .y gt 0) or
                ~(([sd pre: (.x le 0,.y gt 0)
                    comod: (x,y)
                     mod: (x,y)
                     inv: (#y = .y)
                    post: (#y = -.y)])))

sdvs.1.1.2 pp
this: dis3

 formula dis3: [sd pre: (true)
               comod: (all)
                mod: (all)
                inv: (~((~(#y = .y))))
                post: (~(~(#x le 0 & #y gt 0) or ~(formula(s0))),
                      ~((~(#y = .y))))]

sdvs.1.1.2 usable

 u(1) [sd pre: (true)
       comod: (all)
        mod: (diff(all,all))
        post: (#y = y6166,true,

              ([sd pre: (true)
                comod: (diff(all,all))
                 post: (~((~(#y = y6166))))])

             or

               ~(~(#x le 0 & #y gt 0) or
                  ~(([sd pre: (.x le 0,.y gt 0)
                      comod: (x,y)
                       mod: (x,y)
                       inv: (#y = .y)
                      post: (#y = -.y)])))

             or

               ([sd pre: (true)
                 comod: (all)
                  mod: (all)
                  inv: (~((~(#y = .y))))
                  post: (~(~(#x le 0 & #y gt 0) or
                        ~(formula(s0))),
                        ~((~(#y = .y))))])])]

 u(2) [sd pre: (true)
       post: (~(formula(p1)))]


* sdvs.1.1.2 apply
sd[highest applicable]:
proof[]:

    apply - [sd pre: (true)
             comod: (all)
              mod: (diff(all,all))
              post: (#y = y6166,true,
                    ([sd pre: (true)
                      comod: (diff(all,all))
                       post: (~((~(#y = y6166))))])

                  or
```

```
                        ~(~(#x le 0 & #y gt 0) or
                           ~(([sd pre: (.x le 0,.y gt 0)
                               comod: (x,y)
                                 mod: (x,y)
                                 inv: (#y = .y)
                                post: (#y = -.y)])))

                      or

                        ([sd pre: (true)
                           comod: (all)
                            mod: (all)
                            inv: (~((~(#y = .y))))
                           post: (~(~(#x le 0 & #y gt 0) or
                                    ~(formula(s0))),
                                  ~((~(#y = .y))))])]

    non-trivial propagations - ([sd pre: (true)
                                 comod: (diff(all,all))
                                  post: (~((~(#y = y6166))))])

                               or

                                ([sd pre: (true)
                                   comod: (all)
                                    mod: (all)
                                    inv: (~((~(#y = .y))))
                                   post: (~(~(#x le 0 & #y gt 0) or
                                            ~(formula(s0))),
                                          ~((~(#y = .y))))])

sdvs.1.1.2 pp
this: dis2

 formula dis2: ~(~(.x le 0 & .y gt 0) or
                ~(([sd pre: (.x le 0,.y gt 0)
                    comod: (x,y)
                      mod: (x,y)
                      inv: (#y = .y)
                     post: (#y = -.y)])))

* sdvs.1.1.2 mcases
 number of cases: 2
       1st case: formula(dis1)
        proof[]:
       2nd case: formula(dis3)
        proof[]:

    mcases - 2

      open - [sd pre: (formula(dis1))
              comod: (all)
               post: ([sd pre: (true)
                       post: (#y lt 0)])]

* sdvs.1.1.2.1.1 prove
sd: q1
proof[]:

       open - [sd pre: (true)
               post: (#y lt 0)]

 Complete the proof.

sdvs.1.1.2.1.1.1 usable

 u(1) [sd pre: (true)
       comod: (diff(all,all))
        post: (~((~(#y = y6166))))]

 u(2) [sd pre: (true)
        post: (~(formula(p1)))]


* sdvs.1.1.2.1.1.1 apply
sd[highest applicable]: u
number: 1
proof[]:

         apply - [sd pre: (true)
                  comod: (diff(all,all))
                   post: (~((~(#y = y6166))))]

        close - 0 steps/applications
```

```
      close - 1 steps/applications


      open - [sd pre: (formula(dis3))
              comod: (all)
               post: ([sd pre: (true)
                       post: (#y lt 0)])]

 Complete the proof.

sdvs.1.1.2.2.1 usable

 u(1) [sd pre: (true)
       comod: (all)
        mod: (all)
        inv: (~((~(#y = .y))))
       post: (~(~(#x le 0 & #y gt 0) or ~(formula(s0))),
              ~((~(#y = .y))))]

 u(2) [sd pre: (formula(dis1))
       comod: (all)
        post: ([sd pre: (true)
                post: (#y lt 0)])]

 u(3) [sd pre: (true)
       comod: (all)
        mod: (diff(all,all))
       post: (#y = y6166,true,
              ([sd pre: (true)
                comod: (diff(all,all))
                 post: (~((~(#y = y6166))))]) or
             ~(~(#x le 0 & #y gt 0) or
                ~([sd pre: (.x le 0,.y gt 0)
                     comod: (x,y)
                      mod: (x,y)
                      inv: (#y = .y)
                     post: (#y = -.y)])))) or
             ([sd pre: (true)
               comod: (all)
                mod: (all)
                inv: (~((~(#y = .y))))
               post: (~(~(#x le 0 & #y gt 0) or
                          ~(formula(s0))),
                        ~((~(#y = .y))))])]

 u(4) [sd pre: (true)
       post: (~(formula(p1)))]


 No usable quantifiers.

* sdvs.1.1.2.2.1 apply
sd[highest applicable]: u
number: 1
proof[]:

       apply - [sd pre: (true)
                comod: (all)
                 mod: (all)
                 inv: (~((~(#y = .y))))
                post: (~(~(#x le 0 & #y gt 0) or
                           ~(formula(s0))),
                         ~((~(#y = .y))))]

       The postcondition of the last applied state delta is inconsistent
       with the current state.

      close - 0 steps/applications

    join - [sd pre: (formula(dis1) or formula(dis3))
            comod: (all)
             post: ([sd pre: (true)
                     post: (#y lt 0)])]

  close - 2 steps/applications

close - 1 steps/applications
```

Now the proof of Theorem 7, below called *safety2*, using **omegainduct**.

```
safety2:

[sd pre: (covering(all,x,y) & formula(s2))
  comod: (all)
   post: (formula(s8))]


* sdvs.1 prove
  state delta[]: safety2
  proof[]:

open - [sd pre: (covering(all,x,y) & formula(s2))
  comod: (all)
   post: (formula(s8))]

Complete the proof.

* sdvs.1.1 prove
  state delta[]: s8
  proof[]:

  open - [sd pre: (.x le a)
    post: (formula(x.always.le.a))]

Complete the proof.

sdvs.1.1.1 usable

u(1) [sd pre: (true)
 mod: (all)
 inv: (#x = .x)
post: (#x = .x - 1)]


No usable quantified formulas.

* sdvs.1.1.1 omegainduct
      always-formulas: .x le a
  auxiliary-formulas[]:
        base proof[]:
        step proof[]:

    omegainduction on  - (.x le a)

      open - [sd pre: (true)
comod: (all)
 post: (.x le a,true)]

      close - 0 steps/applications

      open - [sd pre: (true)
 post: ([sd pre: (.x le a,true)
  comod: (all)
   mod: (all)
   inv: (#x le a)
  post: (#all ~= .all,#x le a,true)])]

Complete the proof.

sdvs.1.1.1.2.1 goals

g(1) [sd pre: (.x le a,true)
comod: (all)
 mod: (all)
 inv: (#x le a)
post: (#all ~= .all,#x le a,true)]

* sdvs.1.1.1.2.1 prove
  state delta[]: g
       number: 1
  proof[]:

       open - [sd pre: (.x le a,true)
  comod: (all)
   mod: (all)
   inv: (#x le a)
  post: (#all ~= .all,#x le a,true)]

        comment - prove the invariant of the state delta to be proven
```

```
        open - [sd pre: (true)
    comod: (all)
     post: (#x le a)]

        close - 0 steps/applications

 Complete the proof.

sdvs.1.1.1.2.1.2 usable

 u(1) [sd pre: (true) comod: (all) post: (#x le a)]

 u(2) [sd pre: (true)
  mod: (all)
  inv: (#x = .x)
 post: (#x = .x - 1)]


 No usable quantified formulas.

* sdvs.1.1.1.2.1.2 apply
   sd/number[highest applicable/once]: u
                          number: 2

        comment - prove the invariant prior to the application

        open - [sd pre: (.x = x"12)
    comod: (all)
     post: (#x le a)]

        close - 1 steps/applications

        apply - [sd pre: (true)
      mod: (all)
      inv: (#x = .x)
     post: (#x = .x - 1)]

       close - 1 steps/applications

      close - 1 steps/applications

    assert always formula
     - [sd pre: (true) post: (#x le a)]

  close - 1 steps/applications

 close - 1 steps/applications

sdvs.2 usable

 u(1) [sd pre: (covering(all,x,y) & formula(s2))
comod: (all)
 post: (formula(s8))]


 No usable quantified formulas.
```

7 Conclusions

Proofs of typical safety properties of programs in temporal-logic-based systems can be facilitated by the use of two proof rules: the Rule of Negation and the ω-Induction Rule. We have shown that each of these rules is valid only on timelines of certain order types; the joint use of these two rules is valid only on timelines that are finite, or ordered like the natural numbers.

We have demonstrated the use of these rules by giving proofs of safety properties of a simple concurrent program in the State Delta Verification System (SDVS).

References

[1] F. Kroger, *Temporal Logic of Programs*, (Springer-Verlag, 1987). EATCS Monographs on Theoretical Computer Science, Volume 8.

[2] E. A. Emerson, "Temporal and Modal Logic," in *Handbook of Theoretical Computer Science* (ed. J. van Leeuwen), pp. 995–1072, (Elsevier Science Publishers B. V., 1990).

[3] J. V. Cook, I. V. Filippenko, B. H. Levy, L. G. Marcus, and T. K. Menas, "Formal Computer Verification in the State Delta Verification System (SDVS)," in *Proceedings of the AIAA Computing in Aerospace Conference*, (Baltimore, Maryland), pp. 77–87, American Institute of Aeronautics and Astronautics, October 1991.

[4] T. K. Menas, "The Relation of the Temporal Logic of the State Delta Verification System (SDVS) to Classical First-Order Temporal Logic," Technical Report ATR-90(5778)-10, The Aerospace Corporation, 1990.

[5] IEEE, *VHDL Language Reference Manual*, draft standard 1076/B ed., May 1987.

[6] I. V. Filippenko, "VHDL Verification in the State Delta Verification System (SDVS)," in *Proceedings of the 1991 International Workshop on Formal Methods in VLSI Design*, (Miami, FL), ACM, January 1991.

[7] R. Alur and T. A. Henzinger, "A Really Temporal Logic," Technical Report STAN-CS-89-1267, Stanford University, 1989.

[8] K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, (Reading, Massachusetts: Addison Wesley, 1988).

[9] L. Marcus, "The Semantics of Concurrency in SDVS," Technical Report ATR-89(8490)-4, The Aerospace Corporation, 1989.

[10] T. K. Menas, "A Proof of a Safety Property of a Concurrent Program Using the State Delta Verification System (SDVS) with Invariants," Technical Report ATR-90(5778)-9, The Aerospace Corporation, 1990.

[11] L. Marcus, "SDVS 10 Users' Manual," Technical Report ATR-91(6778)-10, The Aerospace Corporation, 1991.