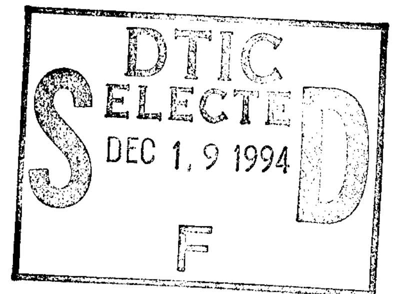
AEROSPACE REPORT NO.
ATR-92(2778)-2

Safety Properties of Terminating and Nonterminating Ada Programs in the State Delta Verification System (SDVS)



30 September 1992

Prepared by

T. K. MENAS
Computer Systems Division

Prepared for

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

Engineering and Technology Group

This document has been approved
for public release and sale; its
distribution is unlimited.

19941214 004

PUBLIC RELEASE IS AUTHORIZED

THE AEROSPACE
CORPORATION
El Segundo, California

DTIC QUALITY INSPECTED 1

Aerospace Report No.
ATR-92(2778)-2

SAFETY PROPERTIES OF TERMINATING AND NONTERMINATING ADA PROGRAMS IN THE STATE DELTA VERIFICATION SYSTEM (SDVS)

Prepared by

T. K. Menas
Computer Systems Division

Accesion For	
NTIS CRA&I	☑
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and / or Special
A-1	

30 September 1992

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

Prepared for

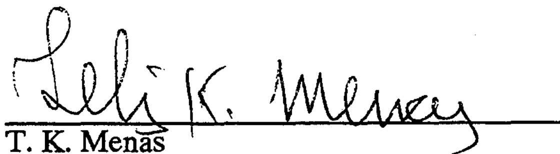
NATIONAL SECURITY AGENCY
Ft. George G. Meade, MD 20755-6000

PUBLIC RELEASE IS AUTHORIZED

Report No.
ATR-92(2778)-2

SAFETY PROPERTIES OF TERMINATING AND NONTERMINATING ADA PROGRAMS IN THE STATE DELTA VERIFICATION SYSTEM (SDVS)

Prepared

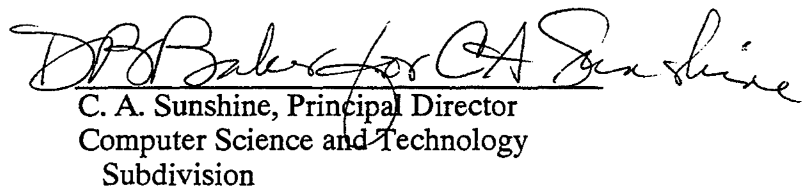


T. K. Menas

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

We describe recent enhancements to the implementation of invariance in SDVS, the purpose of the new **weaknext_tr** flag, and enhancements to the **omegainduct** command. We use these enhancements to SDVS to prove a safety property of both a terminating and a nonterminating Ada program.

Contents

Abstract v

1 Introduction 1

2 A Safety Property of a Terminating Program 3

2.1 Enhancements to Invariance and the **weaknext_tr** Flag 3

2.2 A Proof of a Safety Property of a Terminating Ada Program 4

3 A Safety Proof of a Nonterminating Ada Program 21

3.1 The **omegainduct** Command 21

3.2 A Proof of a Safety Property of a Nonterminating Ada Program 22

4 Conclusions 37

References 39

1 Introduction

This report describes some recent enhancements to SDVS that support the specification and verification of safety properties of programs, and then demonstrates these enhancements through two Ada examples.

We noted in [1] that "Although SDVS without invariance is well-suited for the proof of liveness properties of programs, most safety properties cannot be proved in its logic, because the change in the value of a local variable cannot be constrained to be discrete by any statement of its language. In SDVS with invariance this constraint can, in a sense, be imposed by a simple statement of the language; consequently, a large class of safety properties of terminating programs should be provable in its logic (see [2])." It is now possible for the user to impose this constraint on the state deltas generated by the language translators of SDVS by means of the **weaknext_tr** flag. The application of these state deltas advances the execution of a program by almost discrete steps. Thus, for a terminating program, the only states that are allowed by the symbolic execution of the program are precisely those states that are required by the program execution. The generation of the restricted translator state deltas was easy to implement, but it was impossible to obtain the full theoretical benefits of their meaning in the system. For this, we had to change the SDVS **apply** command at one minor but important point. We discuss this change and the **weaknext_tr** flag in Section 2.1.

The above enhancements to SDVS are not sufficient for proofs of a large class of safety properties of *nonterminating* programs. For a proof of a safety property of a nonterminating program, it is often not enough to require that each state change in its execution be discrete. We must also disallow possible states in its execution that are infinite limits of other states. This restriction is obtained by the use of the **omegainduct** command, which we discussed in [1]. We found it necessary to alter **omegainduct** to make it usable in a wider class of cases. These changes are discussed in Section 3.1.

Our first example of the above changes concerns a terminating Ada program whose function is to calculate the sum of two integers, x and y, by means of a while loop in which two other integers, i and s, are incremented by one, y times. The variable i is initially set to zero and is therefore nonnegative throughout the loop. The latter fact is the safety property of the program that we have chosen to prove in Section 2.2.

The second example concerns a nonterminating Ada program whose function is to switch the values of two variables infinitely often. The safety property we prove is that after the initialization of the two variables, it is the case that at every time in the future (in the execution of the program), there is a still later time at which the values of the variables are switched. We prove this in Section 3.2 and present our conclusions in Section 4.

2 A Safety Property of a Terminating Program

In this section we briefly discuss the enhancements to the invariance code that we found necessary to prove safety properties of programs, the **weaknext_tr** flag, and the proof of a simple safety property of a terminating program. The enhancements and the **weaknext_tr** flag are also needed, along with a revised **omegainduct** command, to prove safety properties of nonterminating programs. We discuss the latter in the next section.

2.1 Enhancements to Invariance and the weaknext_tr Flag

Let S be the state delta $p \; {}_{c}\overset{I}{\leadsto}_{m} \; q$. If in the course of a proof of a state delta with an interpreted invariant J, S is applicable, then the use of the **apply** command[1] with S as the state delta to be applied results in the following sequence of events:

(i) The upper-level dotted places in I are replaced by their possibly symbolic values at the current state, resulting in the interpreted invariant I^*.

(ii) Any information of the current state that depends on the places in the modification list m of S is deleted from the current state.

(iii) The system opens a proof of the state delta $I^* \; {}_{all}\!\leadsto_{\emptyset} J$. The effect of step (ii) is to force the user to prove this state delta at a time intermediate to the precondition and postcondition times of S.

(iv) After the completion of the proof of the state delta of step (iii), the state delta $p \; {}_{c}\!\leadsto_{m} \; q$ is applied.

Suppose that the invariant I of S is the formula $\#all = .all$, and suppose that t_i is the current time (prior to the application of S). Then there is a least time $t_j \geq t_i$ at which the postcondition q of S is true, and, furthermore, all the states in the time interval $[t_i, t_j)$ are stutterings of t_i. By Theorem 1 of [3], the truth of a temporal formula of SDVS is constant in the interval $[t_i, t_j)$. Therefore, in this case, step (ii) above may be eliminated: if $I^* \rightarrow J$ is proved at t_i, then it follows that $I^* \rightarrow J$ at every state in the interval $[t_i, t_j)$. We have amended the implementation of the **apply** command to reflect this fact because, under certain circumstances and for certain J, for example if x is in m and $J \equiv (\#x \geq 0)$, it is not possible to prove the state delta of (iii) after the completion of step (ii).

Leo Marcus first dealt with this limitation by implementing of a new flag, **strongcoverings**. When this flag is on, SDVS asserts certain facts about places at every application. But we discovered that this flag mires the system in an almost endless computation, and that it is also insufficient for proofs in all but the simplest of cases. A partial implementation of Theorem 1 of [3] (in the case of the application of a state delta with the invariant

[1] For a more complete explanation of the **apply** command for a state delta with an invariant, see [2].

($\#all = .all$)), the addition of the **weaknext_tr** flag, and a change in the **omegainduct** command provide a basis for proofs of a large class of safety properties of programs.

We are often interested in the case of $I \equiv (\#all = .all)$ because we have concluded that in order to prove even simple safety properties of a program, the symbolic execution of that program in SDVS must proceed in a manner in which the only states that are allowed in the timeline of the execution are those that are necessitated by the program. To ensure this, we have added a new flag to SDVS, **weaknext_tr**; when it is on, the language translators of SDVS translate a program into state deltas with the invariant ($\#all = .all$). Thus, with this flag on, the execution of the program is forced to proceed in almost discrete steps.

2.2 A Proof of a Safety Property of a Terminating Ada Program

Consider the simple program "add" in the file "add.ada":

```
with text_io; use text_io;
with integer_io; use integer_io;
procedure add is
   i,s,x,y : integer;
   begin
    get(x);
    get(y);
      i := 0;
      s := x;
      while i < y loop
        i := i+1;
        s := s+1;
      end loop;
    put(s);
  end add;
```

If the input value of y is greater than or equal to zero, then the program terminates and the value that is output for s is equal to the sum of the input values of x and y. This is a liveness property of the program. One possible safety property, the one that we will prove, is that in the execution of the program, $i \geq 0$ from the time that i is assigned the value 0 to the time that the "put(s)" statement is executed. In fact, we will prove that both the safety and the liveness properties are true of "add."

We first set the appropriate flags and create the state deltas that encapsulate these properties.

```
<sdvs.3> init
   proof name[]: <CR>

State Delta Verification System, Version 12
```

```
Restricted to authorized users only.

<sdvs.1> setflag
   flag variable: invariance
   on or off[off]: on

 setflag invariance -- on

<sdvs.2> setflag
   flag variable: weaknext_tr
   on or off[off]: on

 setflag weaknext_tr -- on

<sdvs.3> init
   proof name[]: <CR>

State Delta Verification System, Version 12

Restricted to authorized users only.

<sdvs.1> adatr
   path name[/eg5/add.ada]: /eg5/add.ada

 Previously translated Stage 3 Ada file
   -- "/eg5/add.ada"

<sdvs.2> createsd
      name: addsd
   [SD pre: ada(add.ada) , .stdin[2] ge 0, ~(.stdin[1]=.stdin[2])
   comod[]: all
     mod[]: all
     inv[]: <CR>
      post: #i=0 and formula(event1)
   ]

<sdvs.2> createsd
      name: event1
   [SD pre: true
   comod[]: all
     mod[]: all
     inv[]: #i ge .i
      post: #add\pc = exited(standard.text_io.put) and #s=.x + .y
   ]
```

The condition `add\pc =exited(standard.text_io.put)` is attained when the program exits the "put(s)" statement (`add\pc` is the program counter). The state delta *event1* is true at a time t_k iff there is a time $t_l \geq t_k$ such that

$$s(t_l) = x(t_k) + y(t_k)$$

and $i \geq i(t_k)$ in the interval $[t_k, t_l)$.[2] The state delta *addsd* asserts that if the second input value (the value of y) is greater than or equal to zero and the program is executed, then there will be a time in the future at which *event1* will be true.

We first open the proof of *addsd*, advance to the execution of the "i := 0" statement, and omit most of the SDVS output of the intermediate applications. The omitted portions are marked by "etc."

```
<sdvs.2> prove
   state delta[]: addsd
   proof[]: <CR>

 open -- [sd pre: (ada(add.ada),.stdin[2] ge 0,
                   ~(.stdin[1] = .stdin[2]))
            comod: (all)
              mod: (all)
             post: (#i = 0 & formula(event1))]

 Complete the proof.

<sdvs.2.1> until
   formula: #i=0

   apply -- [sd pre: (true)
              comod: (all)
                mod: (add\pc)
                inv: (#all = .all)
               post: (<adatr procedure add is
                                 i, ... : integer
                              begin
                                get (x);
                                ...
                              end add;>)]

----etc.

   apply -- [sd pre: (true)
              comod: (all)
```

[2] For any variable u and time t, $u(t)$ is the value of u at t.

```
                 mod: (add\pc,i)
                 inv: (#all = .all)
                post: (#i = 0,
                       <adatr i := 0;>)]

   until break point reached -- #i = 0
```

Now we open the proof of *event*1.

```
<sdvs.2.16> prove
   state delta[]: event1
   proof[]: <CR>

   open -- [sd pre: (true)
              comod: (all)
                mod: (all)
                inv: (#i ge .i)
               post: (#add\pc = exited(standard.text_io.put) &
                      #s = .x + .y)]

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

     open -- [sd pre: (true)
               comod: (all)
                post: (#i ge 0)]

     close -- 0 steps/applications

 Complete the proof.
```

SDVS automatically opened the proof of the invariant of *event1* at the current state and closed it automatically, since it is clearly true.

We now execute the "s := x" statement by applying the corresponding state delta.

```
<sdvs.2.16.2> usable

 u(1) [sd pre: (true) comod: (all) post: (#i ge 0)]

 u(2) [sd pre: (true)
         comod: (all)
           mod: (add\pc,s)
           inv: (#all = .all)
          post: (#s = .x,
                 <adatr s := x;>)]
```

```
No usable quantified formulas.

<sdvs.2.16.2> apply
  sd/number[highest applicable/once]: u
                             number: 2

    comment -- prove the invariant prior to the application

    open -- [sd pre: (.all = all\305)
              comod: (all)
               post: (#i ge 0)]

    close -- 1 steps/applications

    apply -- [sd pre: (true)
               comod: (all)
                 mod: (add\pc,s)
                 inv: (#all = .all)
                post: (#s = .x,
                       <adatr s := x;>)]

Complete the proof.
```

Note that prior to the application of $u(2)$, SDVS opened and automatically closed a proof of the invariant of the state delta to be proved, *event1*. Henceforth, as long as the state delta to be proved is *event1* (or in "cases" and "induction" proofs therein), then at every application of a state delta with the invariant ($\#all = .all$), SDVS will first open a proof, at the state in which the **apply** command is given, that the interpretation of ($\#all = .all$) implies the invariant of *event*1.

The state of the execution is now at the "while" loop.

```
<sdvs.2.16.2> usable

u(1) [sd pre: (~(.i lt .y))
       comod: (all)
         mod: (add\pc)
         inv: (#all = .all)
        post: (<adatr while i < y

                            i := i + 1;
                            ...
                          end loop;>)]
```

```
u(2) [sd pre: (.i lt .y)
       comod: (all)
         mod: (add\pc)
         inv: (#all = .all)
        post: (<adatr while i < y

                             i := i + 1;
                             ...
                           end loop;>)]
```

```
No usable quantified formulas.
```

We proceed by doing a **cases** on the predicate $(.i \not< .y)$. This is the easy case, since under this condition $y = 0$ and $s = x + y = x$. After five **apply**'s (which we know will suffice from previous executions of the proof), we will exit the standard output of s, and SDVS will open the case of $(.i < .y)$.

```
<sdvs.2.16.2> usable

u(1) [sd pre: (~(.i lt .y))
       comod: (all)
         mod: (add\pc)
         inv: (#all = .all)
        post: (<adatr while i < y

                             i := i + 1;
                             ...
                           end loop;>)]

u(2) [sd pre: (.i lt .y)
       comod: (all)
         mod: (add\pc)
         inv: (#all = .all)
        post: (<adatr while i < y

                             i := i + 1;
                             ...
                           end loop;>)]
```

```
No usable quantified formulas.
```

```
<sdvs.2.16.2> cases
```

```
cases -- ~(.i lt .y)

  open -- [sd pre: (~(.i lt .y))
           comod: (all)
             mod: (all)
             inv: (#i ge 0)
            post: (#add\pc = exited(standard.text_io.put) &
                      #s = stdin\280 + stdin\278)]

<sdvs.2.16.2.1.1> apply
   sd/number[highest applicable/once]:  5

        comment -- prove the invariant prior to the application

        open -- [sd pre: (.all = all\311)
                 comod: (all)
                  post: (#i ge 0)]

        close -- 1 steps/applications

        apply -- [sd pre: (~(.i lt .y))
                  comod: (all)
                    mod: (add\pc)
                    inv: (#all = .all)
                   post: (<adatr while i < y

                                       i := i + 1;
                                       ...
                                    end loop;>)]

        comment -- prove the invariant prior to the application

        open -- [sd pre: (.all = all\315)
                 comod: (all)
                  post: (#i ge 0)]

        close -- 1 steps/applications

        apply -- [sd pre: (true)
                  comod: (all)
                    mod: (add\pc,add)
                    inv: (#all = .all)
                   post: (alldisjoint(add,.add,put\item),
                          covering(#add,.add,put\item),
```

```
                    declare(put\item,type(polymorphic)),
                    <adatr put (s)>)]

comment -- prove the invariant prior to the application

open -- [sd pre: (.all = all\320)
          comod: (all)
           post: (#i ge 0)]

close -- 1 steps/applications

apply -- [sd pre: (true)
           comod: (all)
             mod: (add\pc,put\item)
             inv: (#all = .all)
            post: (#put\item = .s,
                    <adatr put (s)>)]

comment -- prove the invariant prior to the application

open -- [sd pre: (.all = all\324)
          comod: (all)
           post: (#i ge 0)]

close -- 1 steps/applications

apply -- [sd pre: (true)
           comod: (all)
             mod: (add\pc)
             inv: (#all = .all)
            post: (#add\pc = at(standard.text_io.put),
                    <adatr put (s)>)]

comment -- prove the invariant prior to the application

open -- [sd pre: (.all = all\327)
          comod: (all)
           post: (#i ge 0)]

close -- 1 steps/applications

apply -- [sd pre: (.add\pc = at(standard.text_io.put))
           comod: (all)
             mod: (add\pc,stdout[.stdout\ctr],stdout\ctr)
             inv: (#all = .all)
```

```
                    post: (#stdout[.stdout\ctr] = .put\item,
                           #stdout\ctr = .stdout\ctr + 1,
                           #add\pc = exited(standard.text_io.put),
                           <adatr null;>)]

close -- 0 steps/applications

open -- [sd pre: (~((~(.i lt .y))))
          comod: (all)
            mod: (all)
            inv: (#i ge 0)
           post: (#add\pc = exited(standard.text_io.put) &
                  #s = stdin\280 + stdin\278)]
```

```
Complete the proof.
```

We are now in the second case of the "while" loop and must proceed with an induction from $i = 0$ to $i = y$ with the invariant being the formula $s = x + i$ plus the conjunction of the two state deltas that represent the "while" loop.

<sdvs.2.16.2.2.1> *usable*

```
u(1) [sd pre: (~(.i lt .y))
      comod: (all)
        mod: (all)
        inv: (#i ge 0)
       post: (#add\pc = exited(standard.text_io.put) &
              #s = stdin\280 + stdin\278)]

u(2) [sd pre: (~(.i lt .y))
      comod: (all)
        mod: (add\pc)
        inv: (#all = .all)
       post: (<adatr while i < y

                            i := i + 1;
                            ...
                          end loop;>)]

u(3) [sd pre: (.i lt .y)
      comod: (all)
        mod: (add\pc)
        inv: (#all = .all)
       post: (<adatr while i < y
```

```
                              i := i + 1;
                              ...
                            end loop;>)]


 No usable quantified formulas.

<sdvs.2.16.2.2.1> letsd
   name: addloopu2
   state delta[]: u
          number: 2

         letsd -- addloopu2 = u(2)

<sdvs.2.16.2.2.2> letsd
   name: addloopu3
   state delta[]: u
          number: 3

         letsd -- addloopu3 = u(3)

<sdvs.2.16.2.2.3> induct
    induction expression: .i
                    from: 0
                      to: .y
        invariant list[]: .s=.x+.i, formula(addloopu2), formula(addloopu3)
   comodification list[]: x,y
     modification list[]: i,s,add\pc
            base proof[]: <CR>
            step proof[]: <CR>

         induction -- .i from 0 to .y

           open -- [sd pre: (true)
                     comod: (all)
                      post: (.s = .x + .i,
                             [sd pre: (~(.i lt .y))
                               comod: (all)
                                 mod: (add\pc)
                                 inv: (#all = .all)
                                post: (<adatr while i < y

                                                  i := i + 1;
                                                  ...
                                                end loop;>)],
```

```
                    [sd pre: (.i lt .y)
                      comod: (all)
                        mod: (add\pc)
                        inv: (#all = .all)
                       post: (<adatr while i < y

                                            i := i + 1;
                                            ...
                                         end loop;>)],
                    .i = 0)]

close -- 0 steps/applications

open -- [sd pre: (.i ge 0,.i lt .y,.s = .x + .i,
                    [sd pre: (~(.i lt .y))
                      comod: (all)
                        mod: (add\pc)
                        inv: (#all = .all)
                       post: (<adatr while i < y

                                            i := i + 1;
                                            ...
                                         end loop;>)],
                    [sd pre: (.i lt .y)
                      comod: (all)
                        mod: (add\pc)
                        inv: (#all = .all)
                       post: (<adatr while i < y

                                            i := i + 1;
                                            ...
                                         end loop;>)])
             comod: (x,y)
               mod: (i,s,add\pc)
               inv: (#i ge 0)
              post: (#s = #x + #i,
                    [sd pre: (~(.i lt .y))
                      comod: (all)
                        mod: (add\pc)
                        inv: (#all = .all)
                       post: (<adatr while i < y

                                            i := i + 1;
                                            ...
                                         end loop;>)],
```

```
                                        end loop;>)],
                  [sd pre: (.i lt .y)
                    comod: (all)
                      mod: (add\pc)
                      inv: (#all = .all)
                     post: (<adatr while i < y

                                          i := i + 1;
                                          ...
                                        end loop;>)],
                  #i = .i + 1)]
```

```
Complete the proof.
```

The base case was trivially true. We are now in the step case of the induction proof. Eight applications will close the proof (we know this from previous executions; a **go** command would also close the proof).

```
<sdvs.2.16.2.2.3.2.1> apply
   sd/number[highest applicable/once]: 8

          comment -- prove the invariant prior to the application

          open -- [sd pre: (.all = all\338)
                    comod: (all)
                     post: (#i ge 0)]

          close -- 1 steps/applications

          apply -- [sd pre: (.i lt .y)
                     comod: (all)
                       mod: (add\pc)
                       inv: (#all = .all)
                      post: (<adatr while i < y

                                          i := i + 1;
                                          ...
                                        end loop;>)]

          comment -- prove the invariant prior to the application

          open -- [sd pre: (.all = all\341)
                    comod: (all)
                     post: (#i ge 0)]

          close -- 1 steps/applications
```

```
      apply -- [sd pre: (true)
                 comod: (all)
                   mod: (add\pc,i)
                   inv: (#all = .all)
                  post: (#i = .i + 1,
                          <adatr i := i + 1;>)]

      comment -- prove the invariant prior to the application

      open -- [sd pre: (.all = all\344)
                comod: (all)
                 post: (#i ge 0)]

      close -- 1 steps/applications

      apply -- [sd pre: (true)
                 comod: (all)
                   mod: (add\pc,s)
                   inv: (#all = .all)
                  post: (#s = .s + 1,
                          <adatr s := s + 1;>)]

    close -- 0 steps/applications

join induction cases -- [sd pre: (0 le .y)
                          comod: (all,x,y)
                            mod: (i,s,add\pc)
                           post: (#i = .y,#s = #x + #y,
                                   [sd pre: (~(.i lt .y))
                                     comod: (all)
                                       mod: (add\pc)
                                       inv: (#all = .all)
                                      post: (<adatr while i ...

                                                      i :=  ...;
                                                      ...
                                                    end loop;>)],
                                   [sd pre: (.i lt .y)
                                     comod: (all)
                                       mod: (add\pc)
                                       inv: (#all = .all)
                                      post: (<adatr while i ...

                                                      i :=  ...;
                                                      ...
```

```
                                      end loop;>)])]

comment -- prove the invariant prior to the application

open -- [sd pre: (.all = all\350)
          comod: (all)
           post: (#i ge 0)]

close -- 1 steps/applications

apply -- [sd pre: (~(.i lt .y))
           comod: (all)
             mod: (add\pc)
             inv: (#all = .all)
            post: (<adatr while i < y

                                i := i + 1;
                                ...
                              end loop;>)]

comment -- prove the invariant prior to the application

open -- [sd pre: (.all = all\354)
          comod: (all)
           post: (#i ge 0)]

close -- 1 steps/applications

apply -- [sd pre: (true)
           comod: (all)
             mod: (add\pc,add)
             inv: (#all = .all)
            post: (alldisjoint(add,.add,put\item),
                    covering(#add,.add,put\item),
                    declare(put\item,type(polymorphic)),
                    <adatr put (s)>)]

comment -- prove the invariant prior to the application

open -- [sd pre: (.all = all\359)
          comod: (all)
           post: (#i ge 0)]

close -- 1 steps/applications
```

```
    apply -- [sd pre: (true)
               comod: (all)
                 mod: (add\pc,put\item)
                 inv: (#all = .all)
                post: (#put\item = .s,
                       <adatr put (s)>)]

    comment -- prove the invariant prior to the application

    open -- [sd pre: (.all = all\363)
              comod: (all)
               post: (#i ge 0)]

    close -- 1 steps/applications

    apply -- [sd pre: (true)
               comod: (all)
                 mod: (add\pc)
                 inv: (#all = .all)
                post: (#add\pc = at(standard.text_io.put),
                       <adatr put (s)>)]

    comment -- prove the invariant prior to the application

    open -- [sd pre: (.all = all\366)
              comod: (all)
               post: (#i ge 0)]

    close -- 1 steps/applications

    apply -- [sd pre: (.add\pc = at(standard.text_io.put))
               comod: (all)
                 mod: (add\pc,stdout[.stdout\ctr],stdout\ctr)
                 inv: (#all = .all)
                post: (#stdout[.stdout\ctr] = .put\item,
                       #stdout\ctr = .stdout\ctr + 1,
                       #add\pc = exited(standard.text_io.put),
                       <adatr null;>)]

  close -- 3 steps/applications

join -- [sd pre: (true)
          comod: (all)
            mod: (all)
           post: (#add\pc = exited(standard.text_io.put) &
```

```
                         #s = stdin\280 + stdin\278)]

  close -- 2 steps/applications

 close -- 16 steps/applications
```

<sdvs.3> *usable*

```
 u(1) [sd pre: (ada(add.ada),.stdin[2] ge 0,~(.stdin[1] = .stdin[2]))
        comod: (all)
          mod: (all)
         post: (#i = 0 & formula(event1))]


 No usable quantified formulas.
```

3 A Safety Proof of a Nonterminating Ada Program

We first discuss the enhanced version of the **omegainduct** command and then illustrate its use in a safety proof.

3.1 The omegainduct Command

For the reader's convenience, we give a complete description of the command. The **omegainduct** command is based on the formula I_ω:

$$[(true\ {}_{all}{\leadsto}_{\emptyset}\alpha \wedge \beta)\wedge(true\ {}_{\emptyset}{\leadsto}_{\emptyset}((\alpha \wedge \beta)\ {}_{all}\overset{I}{\leadsto}{}_{all}(\alpha[\#/.] \wedge \beta[\#/.] \wedge OR)))] \rightarrow (true\ {}_{\emptyset}{\leadsto}_{\emptyset}\alpha[\#/.])$$

where $I \equiv \alpha[\#/.]$ and $OR \equiv (\#x_1 \neq .x_1 \vee \#x_2 \neq .x_2 \vee \ldots \vee \#x_n \neq .x_n)$ for some local variables $x_1, \ldots, x_n$. This formula is true on precisely those timelines T in which $\omega + 1$ is not embeddable (See [1] and [4]).

The first conjunct in the antecedent of I_ω is the base-case state delta, and the second is the step-case state delta. If **omegainduct** is used in the course of a proof, the user must enter as parameters the formula α on which the induction will proceed, the optional "auxiliary formula" β, and a nonempty set of places $x_1, x_2, \ldots, x_n$. Both formulas must be of type precondition.

The induction 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 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. If the user does not enter an auxiliary formula, the system assumes the formula is "true."

The list of places must have the property that, in the induction step of the proof, at least one of the places will change its value.

After the parameters to **omegainduct** have been given, SDVS opens the proof of the base case of the induction:

$$(true\ {}_{all}{\leadsto}_{\emptyset}\alpha \wedge \beta)$$

Once the base case state delta is proved, SDVS will open the proof of the step-case state delta:

$$[true\ {}_{\emptyset}{\leadsto}_{\emptyset}((\alpha \wedge \beta)\ {}_{all}\overset{I}{\leadsto}{}_{all}(\alpha[\#/.] \wedge \beta[\#/.] \wedge OR))]$$

where $I \equiv \alpha[\#/.]$, and $OR \equiv (\#x_1 \neq .x_1 \vee \#x_2 \neq .x_2 \vee \ldots \vee \#x_n \neq .x_n)$. After the step-case state delta has been proved, SDVS will assert the state delta

$$(true\ {}_{\emptyset}{\leadsto}_{\emptyset}\alpha[\#/.])$$

at the state at which the **omegainduct** command was given.

3.2 A Proof of a Safety Property of a Nonterminating Ada Program

Consider the following Ada program:

```
with text_io; use text_io;
with integer_io; use integer_io;


procedure infswitch is
v, x, y, temp : integer;
begin
 get(x);
 get(y);
 v:= 1;
 while true loop
   temp:= x;
   x:= y;
   y:= temp;
 end loop;
end infswitch;
```

The purpose of *infswitch* is to switch the values of the program variables x and y infinitely often. The purpose of the variable v and the assignment statement "v := 1" is only to allow us to demarcate the point in the execution at which the switching will begin: a possible improvement to SDVS would be a tool that generates and inserts statement labels to a program. This would enable the user to refer to particular points in the execution of the program by means of the program counter "pc." For example, if "<10>" were the statement label of "while true loop" and if in the SDVS symbolic execution of *infswitch*, the value of "infswitch\pc" were equal to "<10>" at the top of the loop, then we could use this fact to demarcate the beginning of the loop.

It is clear from the program, that every time execution is at the top of the loop, there will be a strictly later time at which the program variables x and y will have switched values, that is, the state delta *swap*

$$true \;{}_{all}\!\!\leadsto_{all}(\#x = .y \wedge \#y = .x)$$

is always true at the top of the loop. However, it is not possible to express "at the top of the loop" in the current version of SDVS.

A closer examination of the program reveals that *swap* is always true not only at the top of the loop but at every time within the loop as well. The statement that *swap* is always true is expressed by the state delta *infevent*:

$$true \;{}_{\emptyset}\!\!\leadsto_{\emptyset} formula(swap)$$

Thus, *infevent* is true at the time the assignment statement for v is executed.

There is one more wrinkle in our proof: if $x = y$ initially, then $x = y$ thereafter, and furthermore, all the program variables remain constant from some point on in the execution. In the present implementation of SDVS we cannot prove that, if $x = y$ initially, then the execution of the program does not terminate, a circumstance that makes impossible the use of the **omegainduct** command. (Statement labels and an implementation that forces the program counter to have these labels as values at the appropriate times would correct this defect.) So we will assume that the input values of x and y are different and add this condition to the precondition of *infswap*.

Thus, we will prove the state delta *infswap*:

```
  [sd pre: (ada(infswitch.ada),~(.stdin[1] = .stdin[2]))
    comod: (all)
      mod: (all)
     post: (#v = 1 & formula(infevent))]
```

We first open the proof, apply until $v = 1$, and omit part of the SDVS output (The omitted portions are marked by "etc.").

```
<sdvs.4> prove
   state delta[]: infswap
   proof[]: <CR>

 open -- [sd pre: (ada(infswitch.ada),~(.stdin[1] = .stdin[2]))
           comod: (all)
             mod: (all)
            post: (#v = 1 & formula(infevent))]

 Complete the proof.

<sdvs.4.1> until
   formula: #v=1

   apply -- [sd pre: (true)
              comod: (all)
                mod: (infswitch\pc)
                inv: (#all = .all)
               post: (<adatr procedure infswitch is
                                  v, ... : integer
                                begin
                                  get (x);
                                  ...
                                end infswitch;>)]
```

```
-------etc.

   apply -- [sd pre: (true)
              comod: (all)
                mod: (infswitch\pc,v)
                inv: (#all = .all)
               post: (#v = 1,
                      <adatr v := 1;>)]

   until break point reached -- #v = 1

<sdvs.4.16> usable

 u(1) [sd pre: (~true)
        comod: (all)
          mod: (infswitch\pc)
          inv: (#all = .all)
         post: (<adatr while true

                            temp := x;
                            ...
                          end loop;>)]

 u(2) [sd pre: (true)
        comod: (all)
          mod: (infswitch\pc)
          inv: (#all = .all)
         post: (<adatr while true

                            temp := x;
                            ...
                          end loop;>)]


 No usable quantified formulas.

<sdvs.4.16> letsd
   name: swaploop
   state delta[]: u
         number: 2

   letsd -- swaploop = u(2)
```

The state delta *swaploop* is the branch of the loop that will always be applicable at the top of the loop. We may now use the **omegainduct** command with the induction formula

swap. The successful completion of this command will assert our current goal, *infevent*, at the current time.

```
<sdvs.4.17> omegainduct
                       on:  formula(swap)
   auxiliary formulas[]:  formula(swaploop) and .x~=.y
                   places:  x
           base proof[]:  <CR>
           step proof[]:  <CR>

   omegainduction on  -- (formula(swap))

     open -- [sd pre: (true)
               comod: (all)
                post: ([sd pre: (true)
                         comod: (all)
                           mod: (all)
                          post: (#x = .y & #y = .x)],
                      formula(swaploop) & .x ~= .y)]
```

SDVS has opened the proof of the base case of the omega induction. We must prove that both the induction formula and the auxiliary formulas are true at the current time. The formula that is not known to be true at the current state is the induction formula *swap*. We prove this part of the goal by advancing through part of the loop.

```
<sdvs.4.17.1.1> prove
   state delta[]:  g
          number:  1
   proof[]:  <CR>

       open -- [sd pre: (true)
                 comod: (all)
                   mod: (all)
                  post: (#x = .y & #y = .x)]

 Complete the proof.

<sdvs.4.17.1.1.1> until
   formula:  #x=.y and #y=.x

         apply -- [sd pre: (true)
                    comod: (all)
                      mod: (infswitch\pc)
                      inv: (#all = .all)
                     post: (<adatr while true
```

```
                                        temp := x;
                                        ...
                                      end loop;>)]

    apply -- [sd pre: (true)
               comod: (all)
                 mod: (infswitch\pc,temp)
                 inv: (#all = .all)
                post: (#temp = .x,
                       <adatr temp := x;>)]

    apply -- [sd pre: (true)
               comod: (all)
                 mod: (infswitch\pc,x)
                 inv: (#all = .all)
                post: (#x = .y,
                       <adatr x := y;>)]

    apply -- [sd pre: (true)
               comod: (all)
                 mod: (infswitch\pc,y)
                 inv: (#all = .all)
                post: (#y = .temp,
                       <adatr y := temp;>)]

  close -- 4 steps/applications

close -- 1 steps/applications

open -- [sd pre: (true)
           post: ([sd pre: ([sd pre: (true)
                              comod: (all)
                                mod: (all)
                               post: (#x = .y & #y = .x)],
                            formula(swaploop) & .x ~= .y)
                    comod: (all)
                      mod: (all)
                      inv: ([sd pre: (true)
                              comod: (all)
                                mod: (all)
                               post: (#x = .y & #y = .x)])
                     post: (#x ~= .x,
                            [sd pre: (true)
                              comod: (all)
```

```
                           mod: (all)
                          post: (#x = .y & #y = .x)],
                       formula(swaploop) & #x ~= #y)])]
```

```
Complete the proof.
```

After the completion of the base case, SDVS opened the proof of the step case of the omega induction. We open the proof of the state delta $g(1)$ that comprises the postcondition of the step-case state delta. Note that this state delta has *swap* as its invariant. Henceforth, until the end of the proof of $g(1)$, we will have to prove *swap* at every application, unless SDVS already knows it to be true and closes the proof automatically.

```
<sdvs.4.17.2.1> prove
   state delta[]: g
          number: 1
   proof[]: <CR>

       open -- [sd pre: ([sd pre: (true)
                            comod: (all)
                              mod: (all)
                             post: (#x = .y & #y = .x)],
                         formula(swaploop) & .x ~= .y)
                  comod: (all)
                    mod: (all)
                    inv: ([sd pre: (true)
                            comod: (all)
                              mod: (all)
                             post: (#x = .y & #y = .x)])
                   post: (#x ~= .x,
                          [sd pre: (true)
                            comod: (all)
                              mod: (all)
                             post: (#x = .y & #y = .x)],
                         formula(swaploop) & #x ~= #y)]

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

         open -- [sd pre: (true)
                   comod: (all)
                    post: ([sd pre: (true)
                             comod: (all)
                               mod: (all)
                              post: (#x = .y & #y = .x)])]

         close -- 0 steps/applications
```

```
Complete the proof.

<sdvs.4.17.2.1.2> usable

 u(1) [sd pre: (true)
       comod: (all)
        post: ([sd pre: (true)
                 comod: (all)
                   mod: (all)
                  post: (#x = .y & #y = .x)])]

 u(2) [sd pre: (true)
       comod: (all)
         mod: (infswitch\pc)
         inv: (#all = .all)
        post: (<adatr while true

                             temp := x;
                             ...
                           end loop;>)]

 u(3) [sd pre: (true)
       comod: (all)
         mod: (all)
        post: (#x = .y & #y = .x)]


 No usable quantified formulas.
```

Note that SDVS opened (and closed) the proof of the invariant *swap* at the current state. It closed automatically because we had just proved it.

```
<sdvs.4.17.2.1.2> apply
   sd/number[highest applicable/once]: u
                              number: 2

         comment -- prove the invariant prior to the application

         open -- [sd pre: (.all = all\44)
                   comod: (all)
                    post: ([sd pre: (true)
                             comod: (all)
                               mod: (all)
                              post: (#x = .y & #y = .x)])]
```

```
close -- 1 steps/applications

apply -- [sd pre: (true)
           comod: (all)
             mod: (infswitch\pc)
             inv: (#all = .all)
            post: (<adatr while true

                                   temp := x;
                                   ...
                                 end loop;>)]
```

```
Complete the proof.
```

```
<sdvs.4.17.2.1.2> usable

u(1) [sd pre: (true)
       comod: (all)
         mod: (infswitch\pc,temp)
         inv: (#all = .all)
        post: (#temp = .x,
               <adatr temp := x;>)]


No usable quantified formulas.
```

The proof of *swap* closed automatically because the invariant of the state delta that is applied is the special formula "#all=.all." Thus the state at which *swap* is to be proved is the current state. This was not the case in the previous implementation of invariance. We now execute the statement "temp := x."

```
<sdvs.4.17.2.1.2> apply
  sd/number[highest applicable/once]: <CR>

        comment -- prove the invariant prior to the application

        open -- [sd pre: (.all = all\47)
                  comod: (all)
                   post: ([sd pre: (true)
                               comod: (all)
                                 mod: (all)
                                post: (#x = .y & #y = .x)])]

<sdvs.4.17.2.1.2.1.2> prove
```

```
   state delta[]:  g
          number:  1
   proof[]:  <CR>

          open -- [sd pre: (true)
                   comod: (all)
                     mod: (all)
                    post: (#x = .y & #y = .x)]

 Complete the proof.

<sdvs.4.17.2.1.2.1.2.1> until
   formula:  #x=.y and #y=. x

            apply -- [sd pre: (true)
                       comod: (all)
                         mod: (infswitch\pc,temp)
                         inv: (#all = .all)
                        post: (#temp = .x,
                               <adatr temp := x;>)]

            apply -- [sd pre: (true)
                       comod: (all)
                         mod: (infswitch\pc,x)
                         inv: (#all = .all)
                        post: (#x = .y,
                               <adatr x := y;>)]

            apply -- [sd pre: (true)
                       comod: (all)
                         mod: (infswitch\pc,y)
                         inv: (#all = .all)
                        post: (#y = .temp,
                               <adatr y := temp;>)]

          close -- 3 steps/applications

        close -- 2 steps/applications

        apply -- [sd pre: (true)
                   comod: (all)
                     mod: (infswitch\pc,temp)
                     inv: (#all = .all)
                    post: (#temp = .x,
                           <adatr temp := x;>)]
```

```
Complete the proof.

<sdvs.4.17.2.1.2> usable

u(1) [sd pre: (true)
       comod: (all)
         mod: (infswitch\pc,x)
         inv: (#all = .all)
        post: (#x = .y,
               <adatr x := y;>)]


No usable quantified formulas.
```

The next statement is "x := y."

```
<sdvs.4.17.2.1.2> apply
   sd/number[highest applicable/once]: <CR>

         comment -- prove the invariant prior to the application

         open -- [sd pre: (.all = all\54)
                    comod: (all)
                     post: ([sd pre: (true)
                               comod: (all)
                                 mod: (all)
                                post: (#x = .y & #y = .x)])]

<sdvs.4.17.2.1.2.1.2> prove
   state delta[]: g
          number: 1
   proof[]: <CR>

           open -- [sd pre: (true)
                      comod: (all)
                        mod: (all)
                       post: (#x = .y & #y = .x)]

Complete the proof.

<sdvs.4.17.2.1.2.1.2.1> until
   formula: #x=.y and #y=.x

             apply -- [sd pre: (true)
```

```
                    comod: (all)
                      mod: (infswitch\pc,x)
                      inv: (#all = .all)
                     post: (#x = .y,
                            <adatr x := y;>)]

        apply -- [sd pre: (true)
                    comod: (all)
                      mod: (infswitch\pc,y)
                      inv: (#all = .all)
                     post: (#y = .temp,
                            <adatr y := temp;>)]

      close -- 2 steps/applications

    close -- 2 steps/applications

    apply -- [sd pre: (true)
                comod: (all)
                  mod: (infswitch\pc,x)
                  inv: (#all = .all)
                 post: (#x = .y,
                        <adatr x := y;>)]

 Complete the proof.
```

<sdvs.4.17.2.1.2> *usable*

```
 u(1) [sd pre: (true)
        comod: (all)
          mod: (infswitch\pc,y)
          inv: (#all = .all)
         post: (#y = .temp,
                <adatr y := temp;>)]


 No usable quantified formulas.
```

The last statement of the loop is "y := temp."

<sdvs.4.17.2.1.2> *apply*
   sd/number[highest applicable/once]: *<CR>*

```
        comment -- prove the invariant prior to the application
```

```
        open -- [sd pre: (.all = all\60)
                  comod: (all)
                   post: ([sd pre: (true)
                            comod: (all)
                              mod: (all)
                             post: (#x = .y & #y = .x)])]

<sdvs.4.17.2.1.2.1.2> prove
   state delta[]: g
          number: 1
   proof[]: <CR>

          open -- [sd pre: (true)
                    comod: (all)
                      mod: (all)
                     post: (#x = .y & #y = .x)]

          close -- 0 steps/applications

        close -- 2 steps/applications

        apply -- [sd pre: (true)
                   comod: (all)
                     mod: (infswitch\pc,y)
                     inv: (#all = .all)
                    post: (#y = .temp,
                           <adatr y := temp;>)]

 Complete the proof.

<sdvs.4.17.2.1.2> usable

 u(1) [sd pre: (~true)
       comod: (all)
         mod: (infswitch\pc)
         inv: (#all = .all)
        post: (<adatr while true

                              temp := x;
                              ...
                            end loop;>)]

 u(2) [sd pre: (true)
       comod: (all)
         mod: (infswitch\pc)
```

```
          inv: (#all = .all)
         post: (<adatr while true

                          temp := x;
                          ...
                        end loop;>)]
```

```
No usable quantified formulas.
```

We are now at the top of the loop. But of our three goals, the second one, *swap*, is not known to be true.

<sdvs.4.17.2.1.2> *goals*

```
g(1) #x ~= x\41
g(2) [sd pre: (true)
       comod: (all)
         mod: (all)
        post: (#x = .y & #y = .x)]
g(3) ([sd pre: (true)
         comod: (all)
           mod: (infswitch\pc)
           inv: (#all = .all)
          post: (<adatr while true

                           temp := x;
                           ...
                         end loop;>)]) &
       #x ~= #y
```

<sdvs.4.17.2.1.2> *whynotgoal*
  simplify?[no]: *<CR>*

```
g(2) [sd pre: (true)
       comod: (all)
         mod: (all)
        post: (#x = .y & #y = .x)]
```

We will have to go through a part of the loop once more to establish that *swap* is true at this state as well.

<sdvs.4.17.2.1.2> *prove*
  state delta[]: *g*
         number: *2*

```
proof□: <CR>

      open -- [sd pre: (true)
                comod: (all)
                  mod: (all)
                 post: (#x = .y & #y = .x)]
```

Complete the proof.

```
<sdvs.4.17.2.1.2.1> until
   formula: #x=.y and #y=.x

         apply -- [sd pre: (true)
                    comod: (all)
                      mod: (infswitch\pc)
                      inv: (#all = .all)
                     post: (<adatr while true

                                  temp := x;
                                  ...
                                end loop;>)]

         apply -- [sd pre: (true)
                    comod: (all)
                      mod: (infswitch\pc,temp)
                      inv: (#all = .all)
                     post: (#temp = .x,
                            <adatr temp := x;>)]

         apply -- [sd pre: (true)
                    comod: (all)
                      mod: (infswitch\pc,x)
                      inv: (#all = .all)
                     post: (#x = .y,
                            <adatr x := y;>)]

         apply -- [sd pre: (true)
                    comod: (all)
                      mod: (infswitch\pc,y)
                      inv: (#all = .all)
                     post: (#y = .temp,
                            <adatr y := temp;>)]

      close -- 4 steps/applications
```

```
        close -- 2 steps/applications

      close -- 1 steps/applications

    assert always formula
     -- [sd pre: (true)
           post: (formula(swap))]

  close -- 17 steps/applications
```

<sdvs.5> *quit*

```
 Q.E.D.  The proof for this session is in 'sdvsproof'.

State Delta Verification System, Version 12

Restricted to authorized users only.
```

4 Conclusions

We have demonstrated proofs of simple safety properties of terminating and nonterminating Ada programs. Yet even for the simple safety property of the nonterminating program, the proof was long and, at least in part, somewhat contrived. The addition of statement labels and the suggested implementation for the program counter should alleviate some of these problems.

In general, proofs in SDVS of safety properties of programs require that the timeline of execution of the programs be discrete and, furthermore, that every state (time) allowed in the execution of a program is a state necessary to the execution of the program. This report showed that these restrictions suffice for the proof of safety properties of both terminating and nonterminating Ada programs.

References

[1] T. Menas, "Safety, Invariance, and a New Induction Command in SDVS," Technical Report ATR-92(2778)-1, The Aerospace Corporation, September 1992.

[2] T. K. Menas, "The Implementation of Invariance in the State Delta Verification System (SDVS)," Technical Report ATR-90(5778)-8, The Aerospace Corporation, September 1990.

[3] T. K. Menas, "Variants of Invariance," Technical Report ATR-89(8490)-5, The Aerospace Corporation, September 1989.

[4] T. K. Menas and L. G. Marcus, "Timelines and Proofs of Safety Properties in the State Delta Verification System (SDVS)," Technical Report ATR-92(2778)-9, The Aerospace Corporation, September 1992. Submitted to *Journal of Automated Reasoning.*