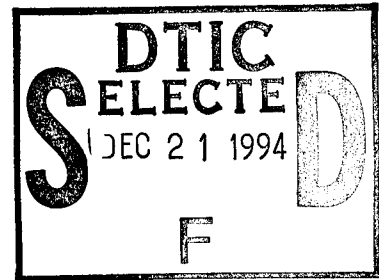


# Isolating and Transforming an Ada Heapsort for SDVS Analysis

30 September 1992

Prepared by

L. A. CAMPBELL  
Computer Systems Division



Prepared for

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

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

Engineering and Technology Group

19941215 145

DTIC QUALITY INSPECTED 1

ISOLATING AND TRANSFORMING AN ADA HEAPSORT  
FOR SDVS ANALYSIS

Prepared by

L. A. Campbell  
Computer Systems Division

30 September 1992

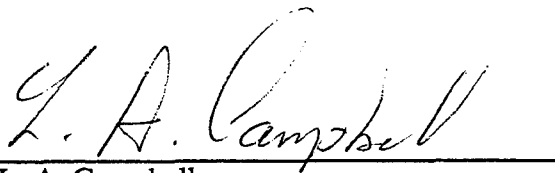
Engineering and Technology Group  
THE AEROSPACE CORPORATION  
El Segundo, CA 90245-4691

Prepared for

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

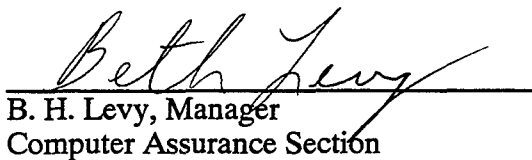
ISOLATING AND TRANSFORMING AN ADA HEAPSORT  
FOR SDVS ANALYSIS

Prepared



L. A. Campbell

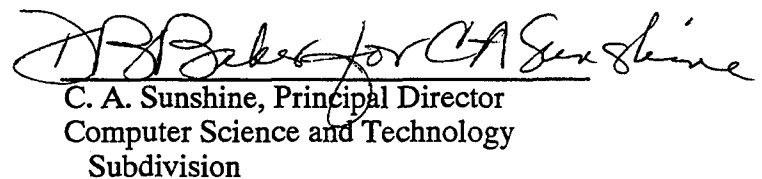
Approved



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

This report examines some of the issues that arise when the State Delta Verification System (SDVS) is used for the analysis of code fragments extracted from larger bodies of "production" code. The code fragment must be isolated from the larger body of code - by narrowing its interface to other program components. It must often be altered as well, to satisfy the narrowed interface semantics or to match available SDVS capabilities. These issues are illustrated by means of a running example, involving a heapsort written in Ada. The code is prepared for SDVS analysis, with the intention of proving that index-out-of-range conditions cannot arise during execution. A bug is uncovered in the original source code in the course of the analysis and the bug is fixed. The planned proof is then successfully carried out for the corrected heapsort code. The point of view is that of a relatively unsophisticated user of the SDVS system.

Accession For	
NTIS CRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution /	
Availability Codes	
Dist	Availability Special
A-1	

## Acknowledgments

My thanks all those who helped me in my struggles with learning to use SDVS. In particular, thanks to Leo Marcus, Telis Menas, Ivan Filippenko, and John Doner. I also appreciate the assistance of Rami Razouk, who supplied me with the program-specific code used as the basis for the primary example in this report.

## Contents

<b>Abstract</b>	<b>v</b>
<b>Acknowledgments</b>	<b>vi</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Original Source Code</b>	<b>3</b>
<b>3 SDVS Version of the Code</b>	<b>5</b>
<b>4 Analysis Using SDVS</b>	<b>9</b>
<b>5 The Critical Lemma</b>	<b>13</b>
<b>6 Discovering a Bug</b>	<b>15</b>
<b>7 Conclusions</b>	<b>17</b>
<b>Appendices:</b>	
<b>A. Proof of the adlemma heapsrt.returns.indices</b>	<b>19</b>
<b>References</b>	<b>31</b>

# 1 Introduction

This report examines some of the issues that arise in applying the State Delta Verification System (SDVS) to fragments of code extracted from a larger "production" code. The intended use of SDVS in such a case is the analysis of the code fragment by verification of selected properties. The properties selected for analysis are determined by the purpose for which the analysis is undertaken.

In this report, it is taken for granted that the reader is familiar with both the Ada language [1] and SDVS [2, 3].

In order to make SDVS analysis a practical proposition, a code fragment of interest must be isolated from the larger matrix of the production program that contains it. This can be a daunting task because of dependencies in the code. Typically, the fragment one wants must be extracted along with a procedure in which it is nested. (Of course, other types of compilation units may be involved.) The procedure may use data types, constants, functions, and procedures that are defined in packages identified in with-clause(s) of the enclosing program. Some minimal set of definitions, sufficient to exercise the code fragment, must be singled out. Actually, the set need not be strictly minimal, but it must be of practical proportions. Simply including everything in the contents of every package mentioned in a with-clause into the set of definitions needed will (given almost any design for a production code of even moderate size) produce too large a body of code for practical use in SDVS. The selection of an appropriate set of definitions to exercise the code fragment is what is here termed "narrowing the interface" (between the code fragment and the rest of the code).

The need to narrow the interface is primarily due to the fact that the current version of the SDVS system is designed to analyze entire Ada programs. While it has facilities to deal with subprograms (for instance, the adalemma capability for proving assertions about subprograms [4, 5]), it requires an entire program for context. Thus the code fragment plus the selected set of definitions on which it depends must be encapsulated in an Ada program. Of course, the appropriate context for a program fragment must be supplied somehow, and it is usually not completely local to the fragment; this is thus an issue faced by any verification system, and requiring a complete program is probably the best interim solution. In principle, one can input a large program to the SDVS Ada translator and subsequently restrict one's attention to the area of the code fragment of interest. This can be done by selecting local properties, involving the code fragment, for verification. In practice, however, inputting a large program will substantially increase the likelihood of running into some Ada language feature that is not within the capabilities of the current version of SDVS. For instance, at the time this report was written, SDVS did not handle floating point arithmetic, even though that is clearly likely to show up in production code (see [6] for future plans). Once the code fragment has been encapsulated in a "scaffold" Ada program, it can be considered isolated from its matrix.

Now consider source code transformations that may have to be applied to the fragment. If one wants to claim that one has verified (some property of) a code fragment extracted from a production program, it is obviously desirable to have analyzed an isolated fragment in SDVS that is syntactically identical to the original fragment. Even when that is the case, one must (informally, at least) show that the semantics of the isolated fragment and the

original fragment either agree, or are related in a way that does not violate the assumptions of the analysis. (Two syntactically identical pieces of code may well have entirely different semantics when encapsulated in different matrices; for instance, two procedures with the same name in the fragment may be defined differently.) Unfortunately, the ideal of syntactic identity is rarely achievable. Changes to the source of the original fragment may be required to deal with differences in the textual context (the selected scaffolding and the original production program). However, this is not the main reason why source code changes are required. Rather, that reason is the limited Ada language capabilities of the current version of SDVS. Constructs that the translator cannot handle must be replaced by equivalent (for analysis purposes) ones that can be handled. The term "transformation" in the title of this document refers to any syntactic changes to the original program fragment.

In so far as the analysis purports to apply to the original fragment, informal proofs must be supplied to show that the transformations applied to the source code either have no effect on the semantics of the program, or at least do not violate the assumptions of the analysis. The topics discussed in general terms in the preceding paragraphs are illustrated in concrete form in the remainder of this report. This is done by means of a running example, in which an Ada heapsort is isolated and transformed, and then is subjected to analysis using SDVS. The heapsort procedure is the code fragment to be analyzed. The procedure was written by Dr. Rami Razouk, of Aerospace, in support of the Defense Support Program (DSP) program office. It was consciously written as a line-for-line translation of a Fortran program. It represents an approximation to code that the contractor plans to deliver later, and was originally embedded in the contractor's prototype Ada software. The purpose that Dr. Razouk had in mind, in translating the Fortran code, was to use it to demonstrate some properties of the sort procedure related to speed of execution, under different sets of conditions involving the use of different compilers and different compiler options. SDVS entered the picture because one of the compiler options investigated was index checking for arrays, and Dr. Razouk was interested in the question of whether one could prove that index checking could be turned off (using `pragma suppress`) with the assurance that there would be no (unnoticed) out-of-range conditions.

To answer this question, one must verify the property that the variables used as indices for the array in question are always in bounds. This turns out to be eminently amenable to SDVS analysis, because SDVS checks that array indices are within bounds during symbolic execution. In detail, before each reference to an array element, SDVS requires the verification of the inequalities needed to establish that variables or expressions used as indices are within bounds. A program for which SDVS can prove termination therefore cannot corrupt memory by means of its array assignment statements, and this answers the principal concern involved in the use of index checking. Interestingly enough, the SDVS analysis turned up a bug that could be traced directly back to the Fortran original.

The point of view of this report is that of a naive user. This report springs from a dual effort to train a new SDVS user (the author) and to institute a search for production code (defined as program specific code, either prototype or deliverable) suitable for SDVS analysis.



## 2 Original Source Code

The code fragment that will be used as the basis for the running example in this document is a heapsort. It was extracted from a prototype version of a DSP contract deliverable. It should be noted that (1) the code was not a final version; (2) it was not therefore, in any sense, a contract deliverable; (3) the actual heapsort procedure was, in fact, not written by the contractor; and (4) the code is a direct translation into Ada of a Fortran heapsort. However, the code (5) was created for program specific analysis purposes; (6) was part of a larger system prototype; and (7) was tested repeatedly in the context of the system prototype.

In the system context, the purpose of the heapsort routine was to sort an array of *radar return records* by ascending value of one specific floating-point component. To accomplish this, the procedure was passed input parameters consisting of an index bound (N) and the array of floating-point values (X). The values to be sorted were thus defined as the values X(1), X(2), ..., X(N). These parameters were not to be changed; in particular, the order of values in the array could not be changed. Instead, the sort procedure was to operate by returning a permutation of the original index values 1, ..., N, namely an array (IOUT), with the property that the sequence X(IOUT(1)), ..., X(IOUT(N)) is in ascending order. Figure 1 shows the Ada specification for the package (HEAPSRT\_PACK) containing the heapsort procedure (HEAPSRT), just as it appeared (except for minor name changes) in the system prototype.

```
with ATYPES_PACK;      use ATYPES_PACK;
package HEAPSRT_PACK is

  procedure HEAPSRT (N : in RR_RANGE; --
                    X : in X_RR_ARRAY; --
                    IOUT : in out RR_INDEX_ARRAY);

end HEAPSRT_PACK;
```

Figure 1: Specification for the HEAPSRT Package

Without a look at ATYPES\_PACK, the above is not very informative. The package ATYPES\_PACK defines the type RR\_RANGE as a subtype of integer, X\_RR\_ARRAY as an array type with index type RR\_RANGE and value type REAL8, and RR\_INDEX\_ARRAY as an array type with index type RR\_RANGE and value type RR\_RANGE. The type REAL8 is a floating-point type, so it is floating-point values that the routine sorts. Refer to Figure 2, the body of the procedure, and note that all the variables used in index calculations are of type RR\_RANGE, and that the single variable, Q, used in value comparisons is of type REAL8 (and is compared to array values and assigned the value of array entries). In the next section the isolation of the procedure and its encapsulation in a stand-alone scaffold program will be addressed.

```

with ATYPES_PACK;      use ATYPES_PACK;
package body HEAPSRT_PACK is
procedure HEAPSRT (N : in RR_RANGE; --
                  X : in X_RR_ARRAY ; --
                  IOUT : in out RR_INDEX_ARRAY) is
  I,L,IR,J,IOUTT: RR_RANGE;  Q: REALS;
begin
  for K in 1..N loop
    IOUT(K):=K;
  end loop; --for K
  L:=RR_RANGE(INTEGER(N)/2+1);
  IR:=N;
  <<L10>>>null;
  if L>1 then
    L:=L-1;
    IOUTT:=IOUT(L);
    Q:=X(IOUTT);
  else
    IOUTT:=IOUT(IR);
    Q:=X(IOUTT);
    IOUT(IR):=IOUT(1);
    IR:=IR-1;
    if IR=1 then
      IOUT(1):=IOUTT;
      goto LEND;
    end if;
  end if;
  I:=L;
  J:=L+L;
  <<L20>>>null;
  if J<=IR then
    if J<IR then
      if X(IOUT(J))<X(IOUT(J+1)) then
        J:=J+1;
      end if;
    end if;
    if Q<X(IOUT(J)) then
      IOUT(I):=IOUT(J);
      I:=J;
      J:=J+J;
    else
      J:=IR+1;
    end if;
    goto L20;
  end if;
  IOUT(I):=IOUTT;
  goto L10;
  <<LEND>>>null;
end HEAPSRT;
end HEAPSRT_PACK;

```

Figure 2: Body for the HEAPSRT Package

### 3 SDVS Version of the Code

The fragment of interest is the HEAPSRT procedure, so the package HEAPSRT\_PACK can be ignored. To create a scaffold program for procedure HEAPSRT, a substitute for ATYPES\_PACK is required. In addition, it will be desirable to have a main program that exercises HEAPSRT at least minimally. This is not strictly necessary, as the ultimate objective is to establish desired properties of the procedure solely by proof.

The original, production version ATYPES\_PACK can be stripped of additional items to the point where it contains only the definitions truly needed for HEAPSRT, namely those of RR\_RANGE, X\_RR\_ARRAY, RR\_INDEX\_ARRAY, and REAL8. These are all defined directly in terms of Ada basics. However, they involve two constructs that the current version of SDVS is not prepared to handle: subtypes (of integer) and floating point types. (This is expected to change soon, with the likely introduction of subtypes in 1993 and the introduction of floating point under study [6]).

To avoid floating point types, the value types of the arrays will have to be something else, and the integer type is a sensible choice. At this point one must invoke the first informal argument. Claim: for the purposes of the analysis (verifying that index constraints are not violated during assignment), it does not matter what the value type of the array is, as long as it is ordered. Not even an informal proof of this claim will be attempted. Indeed, none should be necessary, as the claim is fairly obvious. The purpose of these remarks, though, is to make that claim explicit, and to observe that (1) it is necessary to make the claim to support the analysis, and (2) the claim is not being formally verified.

Currently, SDVS handles arrays whose indices are integers within a range of the form  $i..j$ , where  $i$  and  $j$  are themselves integers. It does not handle the direct declaration of ranges (e.g.  $i..j$ ). Nor does it allow for the declaration of subtypes of integer, or of derived integer types. The only practical remaining possibility is to declare all the array types involved to have syntactically identical (anonymous) index types that are implicit integer subtypes of the range ( $i..j$ ) type. As the original code assumes a lower bound of 1, the appropriate definition is that of a global upper bound for all indices, MAX\_RR. In those terms, X\_RR\_ARRAY becomes an array with index type 1..MAX\_RR and value type integer, and RR\_INDEX\_ARRAY becomes an array type with index type 1..MAX\_RR and value type integer. Ideally, the value type of RR\_INDEX\_ARRAY would be constrained to 1..MAX\_RR, but this constraint cannot be expressed without the ability to designate the index type 1..MAX\_RR directly. The use of a formally (syntactically) different form of declaration for the array types that appear in procedure HEAPSRT once again makes an implicit appeal to a claim that the analysis is not affected by this change. Here the nature of the claim is that of the semantic equivalence of differing Ada declarations. In fact, the Ada reference manual [1, 3.6.1] supports the claim that two different array declarations of the form *type ... is array(i..j) of ...* implicitly define anonymous, compatible integer subtypes as the index type for the arrays or array types.

Figure 3 shows the scaffolding code built to support analysis of the HEAPSRT procedure. The array type definitions have been encapsulated in a package ATYPES\_PACK, to make it as similar to the original as possible, and that package is declared directly in the outer scope of the main program. In the original version of ATYPES\_PACK there was a definition of the

```

with text_io; use text_io;
with integer_io; use integer_io;
procedure sdvs_heapsrt is
----- packaged type declarations
  package ATYPES_PACK is
    MAX_RR : constant integer := 22;
    type X_RR_ARRAY is array(1..MAX_RR) of integer;
    type RR_INDEX_ARRAY is array(1..MAX_RR) of integer;
  end ATYPES_PACK;
  use ATYPES_PACK;
----- data declarations
  K : integer;
  X : X_RR_ARRAY;
  Z : RR_INDEX_ARRAY;
----- heapsort procedure
  {See Figure 4 for the procedure body}
----- begin main procedure
begin -- sdvs_heapsrt
  X(1) := 1;
  X(2) := -16;
  X(3) := 4;
  X(4) := 3567;
  X(5) := -42;
  X(6) := 0;
  K := 1;
  while K <= 6 loop
    put(X(K));
    K := K+1;
  end loop;
  K := 6;
  HEAPSRT(K,X,Z);
  K := 1;
  while K <= 6 loop
    put(X(Z(K)));
    K := K + 1;
  end loop;
end sdvs_heapsrt;

```

Figure 3: Scaffolding Code for SDVS Version

type REAL8 as well. This cannot be done for the SDVS scaffold version, because REAL8 is to be replaced by a basic type (integer), and there is currently no aliasing possibility in SDVS (one cannot define REAL8 to be a subtype or derived type of the integer type).

The scaffold code contains a number of executable statements, including assignments to define array values, a call to the procedure HEAPSRT, and I/O statements to print out the sorted values. Clearly, a trivial test of the operation of the procedure HEAPSRT, such as that coded into the scaffold, does nothing to prove properties of the procedure in general. In principle, one could use scaffold code that never calls the procedure of interest and still verify properties of the procedure. In fact, in the application of SDVS to the analysis of the procedure HEAPSRT, the heart of the analysis lies in the proof of a lemma establishing that the output array IOU<sub>T</sub> is indeed an array of indices, and the executable statements in the outer scope are of virtually no interest. But in practice, to guard against gross errors (e.g. typos and omissions), the entire program (scaffold and fragment) should be tested by compiling it using a reliable Ada compiler and running the resulting executable, before any attempt at SDVS analysis.

```

procedure HEAPSRT (N : in integer;
                  X : in X_RR_ARRAY ;
                  IOUT : in out RR_INDEX_ARRAY) is

  I,L,IR,J,IOUTT: integer;   Q: integer;

  K,LABEL: integer;
begin
  K := 1;
  while K <= N loop
    IOUT(K):=K;
    K := K + 1;
  end loop;

  L:=N/2+1;
  IR:=N;

  LABEL := 10;
  while LABEL /= 30 loop
    if LABEL = 10 then
      if L>1 then
        L:=L-1;
        IOUTT:=IOUT(L);
        Q:=X(IOUTT);
      else
        IOUTT:=IOUT(IR);
        Q:=X(IOUTT);
        IOUT(IR):=IOUT(1);
        IR:=IR-1;
        if IR=1 then
          IOUT(1):=IOUTT;
          LABEL := 30;
        end if;
      end if;
    if LABEL /= 30 then
      I:=L;
      J:=L+L;
      LABEL := 20;
    end if;
    elsif LABEL = 20 then
      if J<=IR then
        if J<IR then
          if X(IOUT(J))<X(IOUT(J+1)) then
            J:=J+1;
          end if;
        end if;
        if Q<X(IOUT(J)) then
          IOUT(I):=IOUT(J);
          I:=J;
          J:=J+J;
        else
          J:=IR+1;
        end if;
      else
        IOUT(I):=IOUTT;
        LABEL := 10;
      end if;
    end if; -- "case" statement on LABEL values
  end loop; -- while LABEL /= 30
end HEAPSRT;

```

Figure 4: SDVS Version of HEAPSRT Procedure

Figure 4 shows the transformed version of the HEAPSRT procedure used for SDVS analysis. Two types of transformations have been applied to the original source code. First, there are the transformations required by the choice of the scaffolding code generated to isolate the procedure. These transformations consist (1) of the systematic replacement of local variables of type `RR_RANGE` by variables of the same name but of type integer, and (2) of the replacement of the local variable `Q` (originally of type `REAL8`) by a variable `Q` of type integer. As a further consequence of these changes, the one explicit type conversion in the original fragment disappears. Second, there are transformations to the source code designed to overcome limitations of the SDVS translator. At the time this project was started, SDVS did not support `for`-loops, so the `for`-loop used for initialization in the original code was replaced by a `while`-loop (SDVS 11 now supports `for`-loops). For clarity, and to match the semantics of a `for`-loop more closely, a fresh integer variable, `K`, is used in the initialization `while`-loop. The original code also contained `GOTO`s. These were replaced by the introduction of an integer variable `LABEL`, which assumes only the values 10, 20, or 30, and of a `while`-loop containing what amounts to a case statement for the value of the variable `LABEL`. Each path of the case statement contains code originally jumped to by one of the `GOTO`s, and changes in the value of the variable `LABEL` determine (as the `GOTO`s did originally) which portion of code is executed next. Both of the transformations of the second type implicitly require the informal proof of a claim that the transformed code is equivalent to the original code (with respect to the properties being analyzed).

The example used here is a good illustration of the problems that are currently likely to arise in isolating and transforming a code fragment for SDVS analysis. It shows that even the isolation and encapsulation of code in a scaffold program is liable to lead to compromises that require arguments to support claims of equivalent semantics with respect to properties being analyzed. Even if all Ada language constructs were handled by SDVS, such compromises would tend to arise from attempting to "narrow the interface" by retaining only truly relevant portions of the production code matrix containing the fragment. As the example shows, choices made in creating the scaffold code can propagate into transformations of the original fragment source code. The transformations of the second kind, those required to remold the fragment code in terms of Ada language constructs handled by SDVS, are more of a practical than a theoretical issue. From a practical (more specifically public relations) point of view, they are, however, quite significant. A customer whose code is being verified would surely like to see the source code, character for character, being used as input to the analysis. Not only is this desirable, but the claims of equivalence that must be made for transformed code cloud the issue of the extent to which the code is being (automatically, mechanically, logically, mathematically – pick your poison) verified.

It should be noted that problems have a positive aspect as well. Any individual problem encountered in isolating and transforming a code fragment is likely to be one that can be easily remedied by a plausible enhancement to SDVS, and therefore suggests the possibility of such an enhancement. Furthermore, examples, such as the heapsort used here, help prioritize tasks for future development of the SDVS theory and tool.

## 4 Analysis Using SDVS

This section describes how SDVS can be used on the isolated and transformed code to analyze the original question of interest. To recall, that question is whether the use of the procedure HEAPSRT can lead to attempts to store data in unsafe or nonexistent locations. This happens if an array-element assignment statement is executed and the index of the array expression is out of the range of allowable indices for the array in question. Note that the applicability of an analysis to the original code fragment, in its original context, is subject to the validity of the informal arguments (supporting isolation and transformations) described in previous sections.

In SDVS any array assignment statement must have provably-in-range indices before it can be symbolically executed. Thus any proof of termination of a program guarantees that the memory corruption problem cannot occur. A look at the SDVS version of the code shows two different varieties of array assignment statements – those in the outer scope and those occurring within procedure HEAPSRT. For the outer scope statements, termination shows only that the particular scaffold chosen will be safe, and carries no guarantee of general applicability. However, the proof will be by means of a lemma concerning the procedure HEAPSRT – one that establishes that HEAPSRT returns (in IOUT) an array of in-range indices. The meaning of this lemma is that, regardless of context, an assignment statement will execute safely if the index is of the form IOUT(J), where J is an in-range index and IOUT denotes the array returned by HEAPSRT. For the second variety of array assignment statements, those that occur within the body of HEAPSRT, the proof of the lemma will establish safety for all calls to HEAPSRT that satisfy the preconditions of the lemma, because that proof requires the symbolic execution of HEAPSRT.

Below is the lemma in question, expressed in the form of a `createadalemma` command. (Definitions, proofs, commands, and the like are shown in the compact form produced by a `write` command in SDVS, rather than as they would be typed at the terminal. Lower case is used for readability, taking advantage of the case insensitivity of Ada identifiers.)

```
createadalemma heapsrt.returns.indices
      file: \"/u/campbell/ada/heapsrtfiles/sdvs_heapsrt.a\"
      procedure: heapsrt
      qualified name: sdvs_heapsrt.heapsrt
      precondition: (.n ge 1,.n le range(iout))
      mod list: (iout)
      postcondition: (formula(iout.indexes.slice))
```

The precondition states that the input argument N to HEAPSRT must satisfy  $1 \leq N \leq \text{range}(\text{IOUT})$ , and the postcondition is defined by

```
(defformula iout.indexes.slice
  "forall u (1 le u & u le #n --> 1 le #iout[u] & #iout[u] le #n)")
```

which expresses the property that each index value IOUT(u) is in the range 1..N, for each u in that range.

The proof of termination of the main program, assuming the truth of the lemma, is captured in the following goal definition and proof scheme

```
(defsd sdvs_heapsrt.terminates.sd
  "[sd pre: (ada(sdvs_heapsrt.a))
   comod: (all)
   mod: (all)
   post: (terminated(sdvs_heapsrt))]"")

(defproof proof.mod.adalemma
  "(prove sdvs_heapsrt.terminates.sd
   proof:
    (go #sdvs_heapsrt\\pc = at(sdvs_heapsrt.heapsrt),
     invokeadalemma heapsrt.returns.indices,
     go #iout = #z,
     provebygeneralization forall v (1 le v & v le 6
                                     --> 1 le .z[v] & .z[v] le 6)

      using: (q(1)),
    go,
     provebyinstantiation formula(z.k.in.bounds)
      using: q(1)
      substitutions: (v=.k),
    go,
     provebyinstantiation formula(z.k.in.bounds)
      using: q(1)
      substitutions: (v=.k),
    go,
     provebyinstantiation formula(z.k.in.bounds)
      using: q(1)
      substitutions: (v=.k),
    go,
     provebyinstantiation formula(z.k.in.bounds)
      using: q(1)
      substitutions: (v=.k),
    go,
     provebyinstantiation formula(z.k.in.bounds)
      using: q(1)
      substitutions: (v=.k),
    go))")
```

The proof has a simple pattern. Symbolic execution of the program proceeds automatically to the point where the main program makes its single call to the procedure HEAPSRT. At



that point the lemma is invoked, and symbolic execution leads to the point at which the procedure is exited and the postcondition formula(*iout.indexes.slice*) is true. The next command proceeds to the point at which the output formal argument (IOUT) is identified with the actual argument (Z) in the call. At this point the postcondition, which was a property of IOUT is transferred to Z (by means of a **provebygeneralization** command and the equality of Z and IOUT), and becomes a property of the actual argument Z. The proof then proceeds automatically, by symbolic execution, to the first iteration of the loop for displaying the sorted output, where the "put(X(Z(K)))" statement is encountered. At this point a proof is required that the symbolic array index Z(K) is within bounds, before the "put" statement can be executed. The formula proved on exit from HEAPSRT

```
forall v (1 le v & v le 6 --> 1 le .z[v] & .z[v] le 6)
```

which is known as *q(1)* at this point in the proof, is used to establish the required property, namely formula(*z.k.in.bounds*), defined as

```
(defformula z.k.in.bounds
  ".z[k] ge origin(x) & .z[k] le (origin(x) + range(x)) - 1")
```

The same pattern is followed for the remaining five iterations of the loop body, and then the program proceeds automatically to termination. There are more elegant ways of symbolically executing through the output loop than by re-proving the same result six times, but this way works and requires no additional lemmas. Note that the proof is very short, meaning that SDVS did most of the work in carrying the symbolic execution to its conclusion. The form in which the proof was captured (i.e., a list of proof commands) omits a great deal of detail that would be seen in a full proof trace.

What has been proved is, of course, only that the main program terminates *if* one assumes the truth of the adalemma. It is the proof of the adalemma that deals with the intricacies of the code in procedure HEAPSRT. The proof of the adalemma is considerably lengthier than the above termination proof. As far as the issues of interest here (isolating and transforming a code fragment for SDVS analysis) are concerned, the actual proof of the adalemma is virtually a technical detail. However, it turns out that carrying out that proof revealed a bug in the original HEAPSRT code. This is addressed in the next section, and the actual proof of the adalemma is given in Appendix A.

The following points are worth making in considering the relative roles of the termination proof and the proof of the adalemma.

- The adalemma is the central result of the analysis, since it provides a tool that can be carried over to the analysis of any program that invokes procedure HEAPSRT (so that the preconditions of the adalemma are satisfied, of course).
- Once the adalemma has been properly defined (pre- and postconditions, mod- and comodlists), its proof, while lengthy, is largely a matter of applying the SDVS mechanisms, and does not raise any further questions about claims concerning semantic

equivalence or the validity of transformations. Rather, the proof pertains strictly to the piece of code as given.

- The selection of the pre- and postconditions for the adalemma is largely a function of what is required for the adalemma to function in the program termination proof.
- Even though the scaffold represents one very particular instance of the use of procedure HEAPSRT, the termination proof provides confidence that the adalemma is the appropriate tool for proving the safety of the use of procedure HEAPSRT in other contexts (that is, within other main programs). It illustrates that the adalemma is actually a functional tool for proving termination. One would not have the same confidence if, say, the scaffold just encapsulated HEAPSRT without invoking it.
- The adalemma only establishes that HEAPSRT does indeed return indices within range. It does not establish that the indices serve to sort the values in HEAPSRT. Thus if the goal of the analysis were to prove that HEAPSRT sorts correctly, a correspondingly more complex adalemma would have to be proved. Such proofs have already been carried out in SDVS for implementations of certain other sorting algorithms, e.g. quicksort [7].

## 5 The Critical Lemma

This section outlines the proof of the adalemma `heapsrt.returns.indices`. The adalemma can be stated informally as follows: if procedure `HEAPSRT` is called with arguments `N`, `X` and `IOUT`, and `N` is positive and less than or equal to the length of the array `IOUT`, then, on exit from `HEAPSRT`, `N` and `X` are unchanged and each `IOUT(I)` for  $1 \leq I \leq N$  is an integer in the same range ( $1 \leq IOUT(I) \leq N$ ). Note that the adalemma carries with it the obligation to prove that procedure `HEAPSRT` terminates.

The following is an informal proof of termination. Refer to Figure 4 for the complete text of the SDVS version of the `HEAPSRT` procedure. The code has the following structure:

```
declarations
begin

  while-loop to initialize IOUT(k) to k, for k in 1..N

    L:=N/2+1;
    IR:=N;

    LABEL := 10;
    while LABEL /= 30 loop
      if LABEL = 10 then

        code block to execute if LABEL is 10

      elsif LABEL = 20 then

        code block to execute if LABEL is 20

      end if; -- "case" statement on LABEL values
    end loop; -- while LABEL /= 30
end HEAPSRT;
```

Figure 5: Outline of `HEAPSRT` Procedure

The while-loop that initializes `IOUT` terminates. Then the while-loop on `LABEL` is entered, with `LABEL` equal to 10. The code block executed when `LABEL` is 10, namely

```
if L>1 then
  L:=L-1;
  IOUTT:=IOUT(L);
  Q:=X(IOUTT);
else
  IOUTT:=IOUT(IR);
  Q:=X(IOUTT);
  IOUT(IR):=IOUT(1);
  IR:=IR-1;
  if IR=1 then
    IOUT(1):=IOUTT;
    LABEL := 30;
  end if;
end if;
if LABEL /= 30 then
  I:=L;
  J:=L+L;
  LABEL := 20;
end if;
```

decrements the quantity  $L+IR$  by exactly one. Neither  $L$  nor  $IR$  is changed in the code block executed when  $LABEL$  is 20, namely

```

if J<=IR then
  if J<IR then
    if X(IOUT(J))<X(IOUT(J+1)) then
      J:=J+1;
    end if;
  end if;
  if Q<X(IOUT(J)) then
    IOUT(I):=IOUT(J);
    I:=J;
    J:=J+J;
  else
    J:=IR+1;
  end if;
else
  IOUT(I):=IOUTT;
  LABEL := 10;
end if;

```

If one assumes that both  $L$  and  $IR$  should be positive at all times (since they are used as indices), then it is clear that the first block can be executed at most finitely many times. Whenever the first block is exited, there are three possible values for  $LABEL$ : 10, 20, 30. If  $LABEL$  is 30, termination occurs; if  $LABEL$  is 10, we get another execution of the first block. Finally, if  $LABEL$  is 20, we execute the second block. Execution remains within the second block until  $J > IR$ , at which point control passes back to the first block. Thus it remains to show that the second block can be repeatedly executed in succession only a bounded number of times. But each successive execution of the second block with  $J \leq IR$  increases the value of  $J$  without changing  $IR$ . This completes the informal proof of termination.

Of course, termination is not the only thing to be proved. One must also verify that the desired conclusion (that the values of  $IOUT$  are proper indices) holds. Consider the formula

$$\text{forall } u \ (1 \leq u \ \& \ u \leq .n \ \rightarrow \ 1 \leq .iout[u] \ \& \ .iout[u] \leq .n)$$

It holds once  $IOUT$  has been initialized, so if it can be shown that it continues to hold during the execution of the `while`-loops on  $LABEL$ , it will hold when execution of procedure `HEAPSRT` terminates (at procedure exit).

The SDVS proof of `adalemma heapsrt.returns.indices` was set up along the lines suggested by the above. That is, in outline, the proof consists of an induction to establish that  $IOUT$  has been initialized, symbolic execution to the point just before the `while`-loop on  $LABEL$ , and then two nested inductions, one on  $L+IR$  and one on  $IR-J$ . For technical reasons the proof is actually more complicated than this; e.g. the induction on  $IR-J$  is repeated to deal with two different cases in the  $L+IR$  induction, and additional stretches of symbolic execution (without induction) must be added to the proof to deal with boundary cases.

The proof is fairly long, partly because there are many different paths for possible symbolic execution through the code, due to the large number of conditionals. One of these paths led to the discovery of a bug in the `HEAPSRT`. The bug, and a fix, are described in the next section. The proof of the `adalemma` (for the corrected code) is listed in Appendix A.

## 6 Discovering a Bug

What actually happened when the proof of `adalemma rudheap.returns.indices` was attempted was that a previously unnoticed bug in HEAPSRT surfaced. It was unnoticed precisely because it was on the path of a symbolic execution that would not tend to be encountered in practice. Consider the block of code executed when LABEL is 10, and, more exactly, the statements

```
IR:=IR-1;
if IR=1 then
  IOUT(1):=IOUTT;
  LABEL := 30;
end if;
```

If procedure HEAPSRT is called with  $N \geq 2$ , then the initial value of IR is 2 or more, and the statements will (eventually) cause IR to be decremented to 1, LABEL will be set to 30, and the while-loop on LABEL will terminate in short order. On the other hand, if HEAPSRT is called with  $N=1$ , these statements will be reached almost immediately, IR will be decremented from 1 to 0, and LABEL will *not* be set to 30. Inspection of the code shows that if out-of-bounds array indices are not detected, the result will be an infinite loop (the second block, where LABEL is 20, sets IOUT(1) to IOUTT and then resets LABEL to 10). That infinite loop will also then copy values into the locations IOUT(0), IOUT(-1), and so on, thus overwriting supposedly safe memory locations. If out-of-bounds array accesses *are* detected, then the second execution of the block of code for LABEL equal to 10 will result in detecting the erroneous access IOUT(0).

This is definitely a bug, even though attempting to sort arrays of length 1 is not a frequent occurrence. It is also obvious that the bug was in the original Fortran code, and had nothing to do with either the translation into Ada or the isolation/transformation of the procedure. When the bug was found, an appropriate test case was set up, and the execution of the Ada main program was abandoned when a constraint error was raised (because RR\_RANGE was declared to be a derived type of POSITIVE). Thus the bug could have (should have?) been caught if there had been just a little more sophisticated testing of the executable Ada version.

Here is the patch that was made to the code to correct the bug. The statements above were replaced by

```
if IR>2 then
  IR:=IR-1;
else
  IOUT(1):=IOUTT;
  LABEL := 30;
end if;
```

The reason that the attempted proof led to the discovery of the bug was that one of the induction invariants was  $IR > 0$ , and this could not be proved with the flawed code. With the above fix in place the proof was successfully carried out. It is listed in Appendix A.

## 7 Conclusions

The discussion in the previous sections is designed to highlight the following aspects of the isolation, transformation, and analysis of code fragments from larger programs (production codes).

1. Compromises of the semantics of production code are virtually inevitable when a code fragment is isolated from a larger matrix.
2. Source code transformations are currently almost inevitable when the code fragment is adapted for SDVS analysis.
3. Both items 1 and 2, above, impact the formal nature of verification.
4. Informal arguments are necessary to support analysis of code fragments.
5. Scaffold code is a distinct help in analyzing a code fragment.

The last point mentioned concerns the fact that although the scaffold code is often theoretically irrelevant to the goal of the analysis, it provides significant support in analyzing a program fragment: feedback from compilation, execution, and interactive proofs in a realistic setting.

The fact that a bug was found in the HEAPSRT procedure chosen for analysis illustrates one of the benefits of formal verification. The bug really had nothing to do with either the isolation of the code fragment or the transformations applied to it. The points above would be equally valid if the code had been free of bugs.

Finally, the following are some recommendations for near-term changes to SDVS that would have been helpful in this exercise:

1. Allow for subtypes and derived types. Even if implemented only for the base-type integer, this would (a) allow for the more convenient naming of index types, and (b) permit greater fidelity to the original source code by providing arbitrary character strings as type names (for the subtypes or derived types).
2. Identify formal and actual parameters more closely. Currently, properties of *scalar* formal parameters to a procedure carry over to actual parameters of output type [2]. In the proof of termination for HEAPSRT, it was explicitly necessary to transfer a property of the formal parameter IOUT to the actual output parameter Z, because the parameters were arrays. It would be useful if the transfer were automatic for arrays as well.

## A. Proof of the adalemma heapsrt.returns.indices

This appendix provides a listing of the proof of the adalemma heapsrt.returns.indices and some supporting materials. To avoid any possible confusion, the first item is a listing of the (bug-fixed) version of procedure HEAPSRT to which the lemma applies.

```
procedure HEAPSRT (N : in integer;
                  X : in X_RR_ARRAY ;
                  IOUT : in out RR_INDEX_ARRAY) is
  I,L,IR,J,IOUTT: integer;  Q: integer;
  K,LABEL: integer;
begin
  K := 1;
  while K <= N loop
    IOUT(K):=K;
    K := K + 1;
  end loop;
  L:=N/2+1;
  IR:=N;
  LABEL := 10;
  while LABEL /= 30 loop
    if LABEL = 10 then
      if L>1 then
        L:=L-1;
        IOUTT:=IOUT(L);
        Q:=X(IOUTT);
      else
        IOUTT:=IOUT(IR);
        Q:=X(IOUTT);
        IOUT(IR):=IOUT(1);
        if IR>2 then
          IR:=IR-1;
        else
          IOUT(1):=IOUTT;
          LABEL := 30;
        end if;
      end if;
    end if;
    if LABEL /= 30 then
      I:=L;
      J:=L+L;
      LABEL := 20;
    end if;
    elsif LABEL = 20 then
      if J<=IR then
        if J<IR then
          if X(IOUT(J))<X(IOUT(J+1)) then
            J:=J+1;
          end if;
        end if;
        if Q<X(IOUT(J)) then
          IOUT(I):=IOUT(J);
          I:=J;
          J:=J+J;
        else
          J:=IR+1;
        end if;
      else
        IOUT(I):=IOUTT;
        LABEL := 10;
      end if;
    end if; -- "case" statement on LABEL values
  end loop; -- while LABEL /= 30
end HEAPSRT;
```

Next, here are a variety of definitions of items that appear in the proof of the adalemma.

```
(defformula iout.indexes.slice
  "forall u (1 le u & u le #n --> 1 le #iout[u] & #iout[u] le #n)")

(defformula iout.in.range
  "forall u (1 le u & u le .n --> 1 le .iout[u] & .iout[u] le .n)")

(defsd dropi.sd
  "[sd pre: (1 le .n & .n le range(iout), 1 le .i & .i le .n,
    forall u ((1 le u & u le .n) & u ^= .i
      --> 1 le .iout[u] & .iout[u] le .n),
    1 le .iout[.i] & .iout[.i] le .n)
  post: (forall u (1 le u & u le .n --> 1 le .iout[u] & .iout[u] le .n))"]

(defsd disjoint.sd
  "[sd pre: (true)
  post: (alldisjoint(iout[1:(.i + (-1))], iout[.i:.i]),
    alldisjoint(iout[(.i + 1):.n], iout[.i:.i]))"]

(defformula while.invariant
  "forall u (1 le u & u le .n --> 1 le .iout[u] & .iout[u] le .n) &
  (1 le .l & .l le .n) &
  (1 le .ir & .ir le .n)")

(defsd initial.values.in.range.sd
  "[sd pre: (forall u (1 le u & u le .n --> .iout[u] = u))
  comod: (all)
  post: (forall u (1 le u & u le .n --> 1 le .iout[u] & .iout[u] le .n))"]

(defsd stays.true.sd
  "[sd pre: (true)
  comod: (heapsrt.k, iout[1:(.heapsrt.k - 1)])
  mod: ()
  post: (forall u (1 le u & u lt .heapsrt.k --> .iout[u] = u))"]

(defsd bridge.sd
  "[sd pre: (.n ge 1, forall u (1 le u & u le .n
    --> 1 le .iout[u] & .iout[u] le .n))
  comod: (n)
  mod: ()
  post: ([sd pre: (true)
    comod: (i, n, iout[1:(.i-1)], iout[(.i+1):.n])
    post: (forall u ((1 le u & u le .n) & u ^= .i
      --> 1 le .iout[u] & .iout[u] le .n)))]")

(defsd bridger.sd
  "[sd pre: (.n ge 1, forall u (1 le u & u le .n
    --> 1 le .iout[u] & .iout[u] le .n))
  comod: (n)
  mod: ()
  post: ([sd pre: (true)
    comod: (ir, n, iout[1:(.ir-1)], iout[(.ir+1):.n])
    post: (forall u ((1 le u & u le .n) & u ^= .ir
      --> 1 le .iout[u] & .iout[u] le .n)))]")

(defsd bridge1.sd
  "[sd pre: (.n ge 1, forall u (1 le u & u le .n
    --> 1 le .iout[u] & .iout[u] le .n))
  comod: (n)
  mod: ()
  post: ([sd pre: (true)
    comod: (n, iout[2:.n])
    post: (forall u ((1 le u & u le .n) & u ^= 1
      --> 1 le .iout[u] & .iout[u] le .n)))]")
```



```

(defsd weaken.ir.sd
  "[sd pre: (true)
   comod: (ir,n,iout[1:(.ir-1)],iout[(.ir+1):.n])
   mod: ()
   post: (forall u ((1 le u & u le .n) & u ^= .ir
                    --> 1 le .iout[u] & .iout[u] le .n))"]

(deformulas induct.lir.invariants
  ".l ge 1"
  ".ir ge 1"
  ".l le .n"
  ".ir le .n"
  ".l + .ir = lir"
  "formula(lab30)"
  "formula(not30)"
  ".label = 10"
  "formula(iout.in.range)")

(deformulas induct.gap.invariants
  ".ir - .j le gap"
  ".label = 20"
  "formula(lab20)"
  "formula(not20)"
  "i le .i"
  ".i le .n"
  "i le .j"
  "forall u (i le u & u le .n --> 1 le .iout[u] & .iout[u] le .n)")

```

Finally, here is the proof itself, in the form of a proof scheme produced by an SDVS write command. Two slight liberties have been taken with the format: in "using:" clauses and in provebygeneralization commands, the term formula(iout.in.range) has been used in this listing instead of the actual quantified formula itself. It is expected that this will be acceptable input to SDVS in the near future.

```

(defproof proof.of.adalemma
  "(adatr \"/u/campbell/ada/heapsrtfiles/sdvs_heapsrt.a\",
   read \"/u/campbell/ada/heapsrtfiles/defns\",
   createadalemma heapsrt.returns.indices
     file: \"/u/campbell/ada/heapsrtfiles/sdvs_heapsrt.a\"
     procedure: heapsrt
     qualified name: sdvs_heapsrt.heapsrt
     precondition: (.n ge 1,.n le range(iout))
     mod list: (iout)
     postcondition: (formula(iout.indexes.slice)),
   proveadalemma heapsrt.returns.indices
   proof:
     (go #heapsrt.k = 1,
      letsd u1 = u(1),
      letsd u2 = u(2),
      let nn = .n,
      induct on: .heapsrt.k
        from: 1
        to: nn + 1
        invariants: (formula(u1),formula(u2),
                    forall u (1 le u & u lt .heapsrt.k --> .iout[u] = u),
                    .n = nn, nn le range(iout))
        comodlist: ()
        modlist: (sdvs_heapsrt\\pc,heapsrt.k,iout[.heapsrt.k])
        base proof:
        step proof:
          (let k1 = .heapsrt.k,
           provebygeneralization forall u (1 le u & u lt k1 --> .iout[u] = u)
           using: (forall u (1 le u & u lt .heapsrt.k --> .iout[u] = u)),

```

```

    apply u(1),
    let oldiout = .iout,
    provebygeneralization forall u (1 le u & u lt k1 --> oldiout[u] = u)
      using: (forall u (1 le u & u lt .heapsrt.k --> .iout[u] = u)),
    readaxioms "\/u/versys/sdvs/axioms/arraycoverings.axioms\",
    provebyaxiom alldisjoint(iout[1:(.heapsrt.k - 1)],iout[.heapsrt.k:.heapsrt.k])
      using: disjoint\\slices,
    prove stays.true.sd
      proof: ,
    apply u(2),
    apply u(2),
    apply u(1),
    provebygeneralization g(3)
      using: (forall u (1 le u & u lt k1 --> .iout[u] = u))),
go #label = 10,
prove initial.values.in.range.sd
  proof: provebygeneralization g(1)
    using: (q(1)),
provebygeneralization forall u (1 le u & u le .n --> .iout[u] = u)
  using: (forall u (1 le u & u lt nn + 1 --> .iout[u] = u)),
apply initial.values.in.range.sd,
read "\/u/versys/sdvs/axioms/div.axioms\",
provebyaxiom .n / 2 ge 0
  using: divge0,
provebyaxiom .n gt .n / 2
  using: divlt,
notice formula(while.invariant),
letsd lab30 = u(2),
letsd not30 = u(3),
prove disjoint.sd
  proof:
    (provebyaxiom alldisjoint(iout[1:(.i - 1)],iout[.i:.i])
      using: disjoint\\slices,
    provebyaxiom alldisjoint(iout[(.i + 1):.n],iout[.i:.i])
      using: disjoint\\slices),
letsd disjoint = u(1),
prove bridge.sd
  proof: prove g(1)
    proof: provebygeneralization g(1)
      using: (formula(iout.in.range)),
letsd bridge = u(1),
prove bridger.sd
  proof: prove g(1)
    proof: provebygeneralization g(1)
      using: (formula(iout.in.range)),
letsd bridger = u(1),
prove bridge1.sd
  proof:
    (provebyaxiom alldisjoint(iout[2:.n],iout[1:1])
      using: disjoint\\slices,
    prove g(1)
      proof: provebygeneralization g(1)
        using: (formula(iout.in.range))),
letsd bridge1 = u(1),
prove dropi.sd
  proof: provebygeneralization g(1)
    using: (q(1)),
letsd dropi = u(1),
cases .l + .ir le 3
then proof:
  (apply not30,
  apply u(2),
  apply u(1),
  provebyaxiom alldisjoint(iout[1:(.ir - 1)],iout[.ir:.ir])
    using: disjoint\\slices,
  provebyaxiom alldisjoint(iout[(.ir + 1):.n],iout[.ir:.ir])

```

```

using: disjoint\\slices,
prove weaken.ir.sd
proof: provebygeneralization g(1)
      using: (formula(iout.in.range)),
apply u(2),
apply u(2),
provebyinstantiation 1 le .iout[.ir] & .iout[.ir] le .n
  using: formula(iout.in.range)
  substitutions: (u=.ir),
notice 1 le .ioutt & .ioutt le .n,
apply u(1),
provebyinstantiation 1 le .iout[1] & .iout[1] le .n
  using: formula(iout.in.range)
  substitutions: (u=1),
apply u(1),
apply u(3),
provebygeneralization formula(iout.in.range)
  using: (q(1)),
apply u(1),
provebyaxiom alldisjoint(iout[2:.n],iout[1:1])
  using: disjoint\\slices,
apply bridge1,
apply u(2),
apply u(2),
provebygeneralization formula(iout.in.range)
  using: (q(1)),
apply 7)
else proof:
  (induct on:   lir
   from:       .l + .ir
   to:         3
   invariants: (formulas(induct.lir.invariants))
   comodlist:  (n,heapsrt.x)
   modlist:    (label,l,ir,i,j,ioutt,iout,q,sdvs_heapsrt\\pc)
   base proof:
   step proof:
     (apply u(1),
      apply u(2),
      cases .l le 1
      then proof:
        (apply u(1),
         provebyaxiom alldisjoint(iout[1:(.ir - 1)],iout[.ir:.ir])
          using: disjoint\\slices,
         provebyaxiom alldisjoint(iout[(.ir + 1):.n],iout[.ir:.ir])
          using: disjoint\\slices,
         prove weaken.ir.sd
          proof: provebygeneralization g(1)
                using: (formula(iout.in.range)),
         apply u(2),
         provebyinstantiation 1 le .iout[.ir] & .iout[.ir] le .n
          using: formula(iout.in.range)
          substitutions: (u=.ir),
         notice 1 le .ioutt & .ioutt le .n,
         apply u(1),
         provebyinstantiation 1 le .iout[1] & .iout[1] le .n
          using: formula(iout.in.range)
          substitutions: (u=1),
         apply u(1),
         apply u(3),
         provebygeneralization forall u (1 le u & u le .n --> 1 le .iout
          [u] & .iout[u]
          le .n)
          using: (q(1)),
         apply u(2),
         apply u(1),
         apply u(2),

```

```

apply u(1),
apply u(1),
apply u(1),
apply u(2),
apply u(1),
letsd lab20 = u(2),
letsd not20 = u(1),
induct on: gap
  from: .ir - .j
  to: -1
  invariants: (formulas(induct.gap.invariants))
  comodlist: (ir,heapsrt.x,n,l,q,ioutt)
  modlist: (i,j,iout,label,sdvs_heapsrt\\pc)
  base proof:
  step proof:
    (apply u(2),
     apply u(2),
     cases .j lt .ir
     then proof:
       (apply u(2),
        provebyinstantiation 1 le .iout[.j] & .iout[.j] le .n
         using: formula(iout.in.range)
         substitutions: (u=.j),
         provebyinstantiation 1 le .iout
           [.j + 1] & .iout
             [.j + 1] le .n
         using: formula(iout.in.range)
         substitutions: (u=.j + 1),
         cases .heapsrt.x[.iout[.j]]
           lt .heapsrt.x[.iout[.j + 1]]
         then proof:
           (apply u(2),
            apply u(1),
            cases .q lt .heapsrt.x[.iout[.j]]
            then proof:
              (apply u(2),
               apply disjoint,
               apply bridge,
               apply u(2),
               apply u(2),
               apply dropi,
               apply u(1),
               apply u(1),
               apply u(2),
               apply u(1))
              else proof:
                cases .q lt .heapsrt.x[.iout[.j]]
                then proof:
                  else proof:
                    (apply u(2),
                     apply u(1),
                     apply u(2),
                     apply u(1)))
            else proof:
              (apply u(2),
               cases .q lt .heapsrt.x[.iout[.j]]
               then proof:
                 (apply u(2),
                  apply disjoint,
                  apply bridge,
                  apply u(2),
                  apply u(2),
                  apply dropi,
                  apply u(1),
                  apply u(1),
                  apply u(2),

```

```

        apply u(1))
    else proof:
      cases .q lt .heapsrt.x[.iout[j]]
      then proof:
        else proof:
          (apply u(2),
            apply u(1),
            apply u(2),
            apply u(1)))
    else proof:
      cases .j lt .ir
      then proof:
        else proof:
          (apply u(2),
            provebyinstantiation 1 le .iout
              [.j] & .iout[j] le .n
            using: formula(iout.in.range)
            substitutions: (u=.j),
            cases .q lt .heapsrt.x[.iout[j]]
            then proof:
              (apply u(2),
                apply disjoint,
                apply bridge,
                apply u(2),
                apply u(2),
                apply dropi,
                apply u(1),
                apply u(1),
                apply u(2),
                apply u(1))
              else proof:
                (apply u(2),
                  apply u(1),
                  apply u(2),
                  apply u(1))),
            comment \"first induction on gap has closed\",
            apply u(2),
            apply u(1),
            apply disjoint,
            apply bridge,
            apply u(2),
            apply u(2),
            provebygeneralization forall u (1 le u & u le .n --> 1 le .iout
              [u] & .iout[u]
              le .n)

          using: (q(1)),
          apply u(1))
    else proof:
      (apply u(3),
        apply u(1),
        provebyinstantiation 1 le .iout[.1] & .iout[.1] le .n
          using: formula(iout.in.range)
          substitutions: (u=.1),
          apply u(1),
          notice 1 le .ioutt & .ioutt le .n,
          apply u(1),
          apply u(2),
          apply u(1),
          apply u(1),
          apply u(1),
          apply u(1),
          apply u(2),
          apply u(1),
          letsd lab20 = u(2),
          letsd not20 = u(1),
          cases .ir lt .j
          then proof:

```

```

(apply u(2),
 apply u(1),
 apply disjoint,
 apply bridge,
 apply u(2),
 apply u(2),
 apply dropi,
 apply u(1))
else proof:
(induct on:      gap
 from:          .ir - .j
 to:            -1
 invariants:    (formulas(induct.gap.invariants))
 comodlist:    (ir,heapsrt.x,n,l,q,ioutt)
 modlist:      (i,j,iout,label,
               sdvs_heapsrt\\pc)

base proof:
step proof:
(apply u(2),
 apply u(2),
 cases .j lt .ir
 then proof:
  (apply u(2),
   provebyinstantiation 1 le .iout
                        [.j] & .iout[.j] le .n
   using: formula(iout.in.range)
   substitutions: (u=.j),
   provebyinstantiation 1 le .iout
                        [.j +
                        1] & .iout
                        [.j + 1]
                        le .n
   using: formula(iout.in.range)
   substitutions: (u=.j + 1),
   cases .heapsrt.x[.iout[.j]]
         lt .heapsrt.x
         [.iout[.j + 1]]
  then proof:
   (apply u(2),
    apply u(1),
    cases .q lt .heapsrt.x[.iout[.j]]
    then proof:
     (apply u(2),
      apply disjoint,
      apply bridge,
      apply u(2),
      apply u(2),
      apply dropi,
      apply u(1),
      apply u(1),
      apply u(2),
      apply u(1))
    else proof:
     cases .q lt .heapsrt.x
           [.iout[.j]]
     then proof:
     else proof:
      (apply u(2),
       apply u(1),
       apply u(2),
       apply u(1)))
    else proof:
     (apply u(2),
      cases .q lt .heapsrt.x[.iout[.j]]
      then proof:
       (apply u(2),

```

```

        apply disjoint,
        apply bridge,
        apply u(2),
        apply u(2),
        apply dropi,
        apply u(1),
        apply u(1),
        apply u(2),
        apply u(1))
    else proof:
      cases .q lt .heapsrt.x
        [.iout[.j]]
      then proof:
        else proof:
          (apply u(2),
           apply u(1),
           apply u(2),
           apply u(1)))
    else proof:
      cases .j lt .ir
      then proof:
      else proof:
        (apply u(2),
         provebyinstantiation 1 le .iout
           [.j] & .iout
             [.j]
             le .n
           using: formula(iout.in.range)
           substitutions: (u=.j),
         cases .q lt .heapsrt.x[.iout[.j]]
         then proof:
           (apply u(2),
            apply disjoint,
            apply bridge,
            apply u(2),
            apply u(2),
            apply dropi,
            apply u(1),
            apply u(1),
            apply u(2),
            apply u(1))
           else proof:
             (apply u(2),
              apply u(1),
              apply u(2),
              apply u(1))),
        comment "\"second induction on gap has closed\"",
        apply u(2),
        apply u(1),
        apply disjoint,
        apply bridge,
        apply u(2),
        apply u(2),
        apply dropi,
        apply u(1))),
    apply u(1),
    apply u(2),
    notice .l + .ir = 3,
    cases .l gt 1
    then proof:
      (apply u(2),
       apply u(1),
       provebyinstantiation 1 le .iout[.l] & .iout[.l] le .n
         using: formula(iout.in.range)
         substitutions: (u=.l),
       apply u(1),

```

```

notice 1 le .ioutt & .ioutt le .n,
apply u(1),
apply u(2),
apply u(1),
apply u(1),
apply u(1),
apply u(2),
apply u(1),
apply u(2),
apply u(1),
apply disjoint,
apply bridge,
apply u(2),
apply u(2),
provebygeneralization formula(iout.in.range)
  using: (q(1)),
apply u(1),
apply u(2),
apply u(2),
apply u(1),
provebyinstantiation 1 le .iout[.ir] & .iout[.ir] le .n
  using: formula(iout.in.range)
  substitutions: (u=.ir),
apply u(1),
notice 1 le .ioutt & .ioutt le .n,
apply u(1),
provebyinstantiation 1 le .iout[1] & .iout[1] le .n
  using: formula(iout.in.range)
  substitutions: (u=1),
provebyaxiom alldisjoint(iout[1:(.ir - 1)],iout[.ir:.ir])
  using: disjoint\\slices,
provebyaxiom alldisjoint(iout[(.ir + 1):.n],iout[.ir:.ir])
  using: disjoint\\slices,
apply bridger,
apply u(2),
apply u(3),
provebygeneralization formula(iout.in.range)
  using: (q(1)),
apply u(1),
notice 1 le .ioutt & .ioutt le .n,
provebyaxiom alldisjoint(iout[2:.n],iout[1:1])
  using: disjoint\\slices,
apply bridge1,
apply u(2),
apply u(2),
provebygeneralization formula(iout.in.range)
  using: (q(1)),
apply 7)
else proof:
  (apply u(2),
  provebyinstantiation 1 le .iout[.ir] & .iout[.ir] le .n
    using: formula(iout.in.range)
    substitutions: (u=.ir),
  apply u(1),
  apply u(1),
  provebyinstantiation 1 le .iout[1] & .iout[1] le .n
    using: formula(iout.in.range)
    substitutions: (u=1),
  provebyaxiom alldisjoint(iout[1:(.ir - 1)],iout[.ir:.ir])
    using: disjoint\\slices,
  provebyaxiom alldisjoint(iout[(.ir + 1):.n],iout[.ir:.ir])
    using: disjoint\\slices,
  apply bridger,
  apply u(2),
  apply u(3),
  provebygeneralization formula(iout.in.range)

```



```

        using: (q(1)),
        apply u(1),
        provebyaxiom alldisjoint(iout[2:.n],iout[1:1])
            using: disjoint\\slices,
        apply bridge1,
        apply u(2),
        apply u(2),
        provebygeneralization formula(iout.in.range)
            using: (q(1)),
        go))),
    comment \"adalemma heapsrt.returns.indices has been proved\")")

```

The proof was developed interactively, of course, and it is entirely possible that it could be shortened or made more elegant. For instance, several **notice** commands that originally served primarily as reminders have been left in place.

The proof takes a bit more than twenty minutes on the most recent version (12a) of SDVS, which became available as this report was being completed. The following is a display of the proof state directly after completing the proof. A few **date** commands have been inserted to demonstrate the statement about the length of time it takes to run the proof.

```

<< initial state >>
  adatr /u/campbell/ada/heapsrtfiles/sdvs_heapsrt.a <7>
  read /u/campbell/ada/heapsrtfiles/defns <6>
  createadalemma heapsrt.returns.indices <5>
  date "11/2/92 10:29:33 Elapsed time is 4 seconds." <4>
  proveadalemma heapsrt.returns.indices <3>
  date "11/2/92 10:52:37 Elapsed time is 23 minutes and 4 seconds." <2>
  comment: adalemma heapsrt.returns.indices has been proved <1>
  --> you are here <--

```

## References

- [1] U. S. Department of Defense, *Reference Manual for the Ada Programming Language (ANSI/MIL-STD-1815A)*, 22 January 1983.
- [2] T. K. Menas, "SDVS 11 Tutorial," Technical Report ATR-92(2778)-12, The Aerospace Corporation, September 1992.
- [3] L. G. Marcus, "SDVS 11 Users' Manual," Technical Report ATR-92(2778)-8, The Aerospace Corporation, September 1992.
- [4] J. E. Doner and J. V. Cook, "Offline Characterization of Procedures in the State Delta Verification System (SDVS)," Technical Report ATR-90(8590)-5, The Aerospace Corporation, September 1990.
- [5] J. V. Cook and J. E. Doner, "Example Proofs Using Offline Characterization of Procedures in the State Delta Verification System (SDVS)," Technical Report TR-0090(5920-07)-3, The Aerospace Corporation, September 1990.
- [6] L. G. Marcus, "Preliminary Investigations into Specifying and Proving Ada Floating-Point Programs in the State Delta Verification System (SDVS)," Technical Report ATR-91(6778)-4, The Aerospace Corporation, September 1991.
- [7] J. V. Cook and J. E. Doner, "A Modular Correctness Proof of a Quicksort Procedure Written in Ada using SDVS," Technical Report ATR-91(6778)-8, The Aerospace Corporation, September 1991.