Comput. Lang. Vol. 18, No. 3, pp. 197–216, 1993
Printed in Great Britain. All rights reserved

0096-0551/93 $5.00 + 0.00
Copyright © 1992 Pergamon Press Ltd

ARRAY REPRESENTATION IN SYMBOLIC EXECUTION

ALBERTO COEN-PORISINI and FLAVIO DE PAOLI
Dipartimento di Elettronica, Politecnico di Milano, Piazza Leonardo da Vinci, 32, Milano 20133, Italy

(Received 10 April 1991; revision received 22 October 1991)

Abstract—Symbolic execution is a well known and powerful technique that allows to perform several activities as program testing, program formal verification, program specialization, etc. However, symbolic execution suffers from some problems which disable it to become a wide used technique. For instance, symbolic execution fails when dealing with indexed variables as arrarys, or dynamic data structures as pointers.

In this paper we discuss the problem of symbolic execution of programs involving indexed variables by providing a formal definition of symbolic execution. From this starting point we present a practical technique of representing indexed variables that we argue to be more effective than other approaches found in literature. Finally, we present two examples of application of our technique. The former is an example of symbolic testing while the latter is a formal verification of a program.

Symbolic execution   Indexed variables   Program testing   Program verification   Ada

1. INTRODUCTION

Symbolic execution is a powerful technique that underlies several activities. For instance, symbolic testing is more effective than numeric testing since a symbolic execution represents a class of conventional executions [1, 2]. Most of the existing pathwise data test generators are usually based on symbolic execution [3–5]. Symbolic execution can also be used to verify program correctness by adding to the program some first order predicates describing its logical properties [6, 7]. Furthermore, a program transformation that simplifies (specializes) source code can be performed by means of symbolic execution [8, 9].

Although symbolic execution is a well-known technique, some problems have to be solved to make it become practically applicable. Among them, dealing with arrays is a crucial topic. Let us consider a reference such as A(I). Since I could be bound to a symbolic value, it may be impossible to identify unequivocally which array element is referenced. Even if I has a numeric value it may be impossible to identify unequivocally the value of A(I) because of previous assignments with symbolic indexes. The same problem arises when dealing with pointers; here it can be unknown even the number of objects a point may refer to.

Several tools have been built to provide environments performing symbolic execution. Among them EFFIGY [2] provides an environment to test PL/1 programs, DISSECT [10] for testing FORTRAN programs, UNISEX [6] for testing and verifying Pascal programs, SYMBAD [11] for testing and verifying sequential Ada programs and SESADA [12] for specializing sequential Ada programs.

However, the tools described in [10], [6], [2] are unsatisfactory when dealing with arrays or reference type (pointers). In [2] the proposed approach is to proceed with N parallel computations each time a symbolic reference is encountered, where N is the size of the array. Other solutions consist in letting conditional expressions represent the value of array elements [6] or in asking the user to provide an actual value each time an ambiguous reference is encountered [10], [6]. Unfortunately, these solutions are practically unfeasible even for small sized arrays and are not applicable to pointers.

In this paper we present an original approach for representing arrays that is more manageable and easily extensible to handle pointers as well. In our approach each variable is bound to a set. Such set contains every symbolic value that the variable can have and the constraints under which each value will hold. The sets are incrementally updated to avoid an excessive growth of their contents. The approach described in this paper has been used in both SYMBAD and SESADA.

The paper is organized in the following way: Section 2 presents our approach to array handling through the discussion and the formal definition of symbolic execution involving arrays. In Section 3 we provide some hints about the implementation of suitable algorithms for handling arrays. Section 4 contains two examples of symbolic execution; the former is an application of symbolic execution to program testing while in the latter symbolic execution is used to formally verify program correctness. Finally, Section 5 draws some conclusions.

2. ARRAY HANDLING

In symbolic execution, variables are bound to symbolic expressions. Each time a decision point is reached values of variables may not suffice to choose along which branch the execution has to continue. In such a case some assumptions on the values of variables have to be taken; a first order predicate, which is usually called *path condition*, represents such assumptions. Thus, the *symbolic state* of a program is the pair ⟨State,PC⟩, where State represents the *program state* and PC represents the path condition. The program state describes the variable-value bindings; it is usually defined as a set of pairs ⟨*Var,Var*⟩, where Var is a variable identifier and Val is a symbolic expression.

This definition of the program state, however, is inadequate to represent symbolic execution involving arrays. In fact, simple variables and array elements may not be bound to a unique symbolic value because of previous assignments with symbolic indexes. For instance, the statement X:=A(I) causes variable X not to be bound unequivocally whenever A(I) is not. As a consequence, the program state should associate multiple values with every variable. Further, some constraints are stored in the program state to select the actual value of each variable.

For example, let us consider the following fragment of program (in Ada syntax):

```
for I in 1..N loop
   A(I):=0;
end loop;
GET(J);
A(J):=1;
```

The value of the kth array element is 1 if the value of variable J is equal to k, otherwise it is 0.

Therefore, we define the program state as the pair ⟨*Var,Value_set*⟩. Var is a variable identifier and Value_set is a set containing pairs of the form ⟨*Val,V_constraint*⟩, where Val is a symbolic expression and V_constraint is a boolean expression stating under which constraints Var has value Val. Thus, the value* of a simple variable X is described by a value-set $\mathscr{X}$: $\{\langle\beta_1,Q_1\rangle,\ldots,\langle\beta_n,Q_n\rangle\}$, i.e. X has value β_i if and only if Q_i holds. In the same way, the values of an indexed variable are described by a value-set $\mathscr{A}$: $\{\langle\alpha_1,P_1(I)\rangle,\ldots,\langle\alpha_m,P_m(I)\rangle\}$, where I represents the formal index of the array. Therefore, the Ith array element has value α_i iff $P_i(I)$ holds.

For example: the symbolic state after the execution of the previous program is:

State: $\{\langle A,\{\langle 0,I\neq\varphi\rangle,\langle 1,I=\varphi\rangle\}\rangle,\langle J,\{\langle\varphi,\text{true}\rangle\}\rangle\}$
PC:...

In what follows we discuss the activity of symbolic execution involving indexed variables by providing the semantics, in a denotational semantic style [13, 14], of two functions: ***eval*** and ***exec.eval*** represents the evaluation of expressions, while ***exec*** describes the activity of symbolic execution.

2.1. *Evaluation of expressions*

The function ***eval***(Expr,State) returns the value-set of the expression Expr in the program state State. In the sequel we provide the definition of function **eval** for simple variables, indexed variables and expressions.

*In the following we use the term value to represent what is bound to a variable in the Program state.

Let State be:

State: $\{\langle X,\mathscr{X}\rangle,\langle A,\mathscr{A}\rangle,\ldots\} =$
$\{\langle X,\{\langle\beta_1,Q_1\rangle,\ldots,\langle\beta_n,Q_n\rangle\}\rangle,\langle A,\{\langle\alpha_1,P_1(I)\rangle,\ldots,\langle\alpha_m,P_m(I)\rangle\}\rangle,\ldots\}$

where X is a simple variable, A is an array, and I represents its formal index.

Simple variables. The definition of function ***eval*** for simple variables is:

$$\textit{\textbf{eval}}(X,\text{State}) = \mathscr{X} = \{\langle\beta_1,Q_1\rangle,\ldots,\langle\beta_n,Q_n\rangle\}$$

Indexed variables. The definition of function ***eval*** for indexed variables is:

$$\textit{\textbf{eval}}(A(X),\text{State}) = \mathscr{A}(\textit{\textbf{eval}}(X,\text{State})) = \mathscr{A}(\chi) =$$
$$\{\langle\alpha_1,P_1(\beta_1)\wedge Q_1\vee\ldots\vee P_1(\beta_{11})\wedge Q_n\rangle,\ldots,\langle\alpha_m,P_m(\beta_1)\wedge Q_1\vee\ldots\vee P_m(\beta_n)\wedge Q_n\}$$

Expressions. Let us consider an expression $\text{Expr}(V_1,\ldots,V_n)$. We denote the value-set of each V_i, either simple or indexed, by:

$$v_1:\{\langle\alpha_{11},P_{11}\rangle,\ldots,\langle\alpha_{1m},P_{1m}\rangle\}$$
$$\ldots$$
$$v_n:\{\langle\alpha_{n1},P_{n1}\rangle,\ldots,\langle\alpha_{nr},P_{nr}\rangle\}$$

The evaluation of $\text{expr}(V_1,\ldots,V_n)$ is:

$$\textit{\textbf{eval}}(\text{Expr}(V_1,\ldots,V_n),\text{State}) = \text{Expr}(\textit{\textbf{eval}}(V_1,\text{State}),\ldots,\textit{\textbf{eval}}(V_n,\text{State})) =$$
$$\{\langle\text{Expr}(\alpha_{11},\ldots,\alpha_{n1}),P_{11}\wedge\cdots\wedge P_{n1}\rangle,\ldots,\langle\text{Expr}(\alpha_{1m},\ldots,\alpha_{nr}),P_{1m}\wedge\ldots\wedge P_{nr}\rangle\}$$

The cardinality of the value-set associated to an expression equals the product of the cardinality of the value-sets involved in the expression:

$$|\textit{\textbf{eval}}(\text{Expr}(V_1,\ldots,V_n),\text{State})| = |v_1| * \ldots * |v_n|$$

Some V_constraints of the resulting pairs could be *false*. If this is the case the corresponding values are unfeasible and such pairs can be removed from the resulting value-set.

For example, given the following symbolic state

State: $\{\langle A,\{\langle\alpha,I=0\rangle,\langle\beta,I\neq0\rangle\}\rangle,\langle N,\{\langle\eta,\text{true}\rangle\}\rangle,\langle X,\{\langle\gamma,\alpha\geqslant0\rangle,\langle\delta,\alpha<0\rangle\}\rangle\}$

PC:...

we have that:

$$\textit{\textbf{eval}}(X,\text{State}) = \{\langle\gamma,\alpha\geqslant0\rangle,\langle\delta,\alpha\neq0\rangle\}$$

$$\textit{\textbf{eval}}(A(X),\text{State}) = \{\langle\alpha,(\gamma=0\wedge\alpha\geqslant0)\vee(\delta=0\wedge\alpha<0)\rangle,\langle\beta,(\gamma\neq0\wedge\alpha\geqslant0)\vee(\delta\neq0\wedge\alpha<0)\rangle\}$$

$$\textit{\textbf{eval}}(A(3)+X,\text{State}) = \textbf{eval}(A(3),\text{State}) + \textbf{eval}(X,\text{State}) =$$
$$\{\langle\alpha,3=0\rangle,\langle\beta,3\neq0\rangle\} + \{\langle\gamma,\alpha\geqslant0\rangle,\langle\delta,\alpha<0\rangle\} = \{\langle\beta+\gamma,\alpha\geqslant0\rangle,\langle\beta+\delta,\alpha<0\rangle\}$$

2.2. Symbolic execution

Symbolic execution is defined by the function **exec**(S,⟨State,PC⟩) that returns the symbolic state obtained by executing statement S from the symbolic state ⟨State,PC⟩. In what follows we provide the definition of **exec** for the basic statements of an Ada-like language.

Initialization. The initial value-set of a variable V is: {⟨*undef*,true⟩}, i.e. V is undefined. If V is an array then all array elements are undefined.

Assignment statement. Let us consider an assignment of the form $X:=\text{Expr}(V_1,\ldots,V_n)$. To execute such statement it is necessary to update the program state by evaluating the right hand side expression of the assignment.

Let State be $\{\langle X,\mathscr{X}\rangle,\langle A,\mathscr{A}\rangle,\ldots\}$; the definition of function **exec** for the assignment statement of simple variables is:

exec(X:=Expr(. . .), ⟨State,PC⟩) = ⟨State′,PC⟩, where
State′:{⟨X,***eval***(Expr(. . .),State)⟩, ⟨A,$\mathscr{A}$⟩, . . . }

After the execution of the assignment the value-set of the expression becomes the value-set of X.

For example, given the following symbolic state:

State: $\{\langle A,\{\langle\alpha,I=0\rangle,\langle\beta,I\neq 0\rangle\}\rangle,\langle N,\{\langle\eta,\text{true}\rangle\}\rangle,\langle X,\{\langle\gamma,\alpha\geqslant 0\rangle,\langle\delta,\alpha<0\rangle\}\rangle\}$
PC:...

we have that

$$\textbf{\textit{exec}}(\mathscr{X}{:=}A(3)+X,\langle \text{State},\text{PC}\rangle)=\{\langle A,\{\langle\alpha,I=0\rangle,\langle\beta,I\neq 0\rangle\}\rangle,\langle N,\{\langle\eta,\text{true}\rangle\}\rangle,$$
$$\langle X,\{\langle\beta+\gamma,\alpha\geqslant 0\rangle,\langle\beta+\delta,\alpha\langle 0\rangle\}\rangle\},\text{PC}\rangle$$

Now, let us consider the statement A(X):=Expr(...). The definition of function ***exec*** for assignment statements of indexed variables is:

exec(A(X):=Expr(...), ⟨State,PC⟩) = ⟨State′,PC⟩, where
State′: $\{\langle X,\mathscr{X}\rangle,\langle A,\mathscr{A}'\oplus\mathscr{E}'\rangle,\ldots\}$

The new value-set of A is composed of two value-sets $\mathscr{A}'$ and $\mathscr{E}'$. $\mathscr{A}'$ contains every pair ⟨Val, V_constraint⟩ of $\mathscr{A}$; but, each V_constraint is modified to express that an array element has value Val if and only if the assignment does not affect the element. $\mathscr{E}'$ contains all the new pairs created by the assignment. $\mathscr{A}'$ and $\mathscr{E}'$ are built as follows. Let State be:

State: $\{\langle X,\mathscr{X}\rangle,\langle A,\mathscr{A}\rangle,\ldots,\}=$
$\{\langle X,\{\langle\beta_1,Q_1\rangle,\ldots,\langle\beta_n,Q_n\rangle\}\rangle,\langle A,\{\langle\alpha_1,P_1(I)\rangle,\ldots,\langle\alpha_m,P_m(I)\rangle\}\rangle,\ldots\}$

and let ***eval***(Expr(...),State) be $\mathscr{E}\{\langle\gamma_1,T_1\rangle,\ldots,\langle\gamma_s,T_s\rangle]$, then

$$\mathscr{A}'=\{\langle\alpha_1,P_1(I)\wedge I\neq\beta_1\wedge\ldots,\langle\alpha_m,P_m(I)\wedge I\neq\beta_1\wedge\ldots\wedge I\neq\beta_n\rangle\}$$
$$\mathscr{E}'=\{\langle\gamma_1,T_1\wedge(I=\beta_1\wedge\ldots\wedge I=\beta_n)\rangle,\ldots,\langle\gamma_s,T_s\wedge(I=\beta_1\vee\cdots\vee I=\beta_n)\rangle\}$$

The two sets, $\mathscr{A}'$ and $\mathscr{E}'$ are composed by the operator ⊕ which takes as arguments two value-sets and produces as result a value-set containing one pair for each value in either of its arguments.

Given two value-sets $\mathscr{A}$: $\{\langle\alpha_1,P_1\rangle,\ldots,\langle\alpha_m,P_m\rangle\}$ and $\mathscr{B}$: $\{\langle\beta_1,Q_1\rangle,\ldots,\langle\beta_n,Q_n\rangle\}$ the operator ⊕ is defined as follows:

$$\mathscr{A}\oplus\{\,\}=\mathscr{A}$$
$$\mathscr{A}\oplus(\beta\cup\{\langle\beta_{n+1},Q_{n+1}\rangle\})=\textbf{\textit{if}}\ \exists j\in[1\ldots m]\,|\,\alpha_j=\beta_{n+1}$$
$$\textbf{\textit{then}}\ \{\langle\alpha_1,P_1\rangle,\ldots,\langle\alpha_j,P_j\vee Q_{n+1}\rangle,\ldots,\langle\alpha_m,P_m\rangle\}\oplus\mathscr{B}$$
$$\textbf{\textit{else}}\{\langle\alpha_1,P_1\rangle,\ldots,\langle\alpha_m,P_m\rangle,\langle\beta_{n+1},Q_{n+1}\rangle\}\oplus\mathscr{B}$$

For example: given the following symbolic state:

State: $\{\langle A,\{\langle\alpha,I=0\rangle,\langle\beta,I\neq 0\rangle\}\rangle,\langle N,\{\langle\eta,\text{true}\rangle\}\rangle,\langle X,\{\langle\gamma,\alpha\geqslant 0\rangle,\langle\delta,\alpha<0\rangle\}\rangle\}$
PC:...

we have that:

$$\textbf{\textit{exec}}(A(X){:=}X,\langle \text{State},\text{PC}\rangle)=\langle\{\langle A,\mathscr{A}'\oplus\mathscr{E}'\rangle,\langle N,\{\langle\eta,\text{true}\rangle\}\rangle,$$
$$\langle X,\{\gamma,\alpha\geqslant 0\rangle,\langle\delta,\alpha<0\rangle\}\rangle\},\text{PC}\rangle$$

where

$$\mathscr{A}':\ \{\langle\alpha,I=0\wedge I\neq\gamma\wedge I\neq\delta\wedge I\neq\gamma\wedge I\neq\delta\rangle,\langle\beta,I\neq 0\wedge I\neq\gamma\wedge I\neq\delta\rangle\}$$
$$\mathscr{E}':\ \{\langle\gamma,\alpha\geqslant 0\wedge(I=\gamma\vee I=\delta)\rangle,\langle\delta,\alpha<0\wedge(I=\gamma\vee I=\delta)\rangle\}$$
$$\mathscr{A}'\oplus\mathscr{E}':\ \{\langle\alpha,I=0\wedge I\neq\gamma\wedge I\neq\delta\rangle,\langle\beta,I\neq 0\wedge I\neq\gamma\wedge I\neq\delta\rangle,\langle\gamma,\alpha\geqslant 0\vee(I=\gamma\vee I=\delta)\rangle,$$
$$\langle\delta,\alpha\langle 0\rangle\wedge(I=\gamma\vee I=\delta)\rangle\}$$

Note that the V_constraints of a given value-set $\mathscr{A}$: $\{\langle\alpha_1,P_1\rangle,\ldots,\langle\alpha_n,P_n\rangle\}$ are mutually exclusive by construction:

$$\forall\, i,j\in(1\ldots n)\,|\,i\neq j(P_i\wedge P_j=\text{false}) \qquad [P1]$$

Further, we have that:

$$P_1\vee\ldots\vee P_n=\text{true} \qquad [P2]$$

This is not surprising since such properties reflect the semantics of program execution. [P1] states that in any state of a numerical execution there is at most one P_i true, i.e. each variable can have at most one value. [P2] states that in any state of a numerical execution there is a P_i true, i.e. each variable has at least one value. Combining [P1] and [P2] we obtain that in any state of a numerical execution each variable has exactly one value, as expected.

Sequence of statement. The definition of function ***exec*** for a sequence of statements S1; S2;...Sn is:

$$\textit{\textbf{exec}}(\text{S1;S2;}\ldots\text{Sn},\langle\text{State,PC}\rangle) = \textit{\textbf{exec}}(\text{S2;}\ldots\text{Sn},\textit{\textbf{exec}}(\text{S1},\langle\text{State}\rangle,\text{PC}\rangle))$$

Conditional statement. Whenever a conditional statement of the form **if** C **then** S1 **else** S2 **end if** is reached, the boolean expression C is evaluated along which branch the computation has to continue. Thus, if PC implies the result of the evaluation of C ($\neg$C) in the current program state, the **then** (**else**) branch is executed.

Let state be

$$\textbf{State: } \{\langle v_1, \{\langle\alpha_{11},P_{11}\rangle,\ldots,\langle\alpha_{1m},P_{1m}\rangle\},\ \ldots,\ \langle v_n, \{\langle\alpha_{n1},P_{n1}\rangle,\ldots,\langle\alpha_{nr},P_{nr}\rangle\}\rangle$$

The result of the evaluation of $C(V_1,\ldots,V_n)$ in the current state is:

$$\textit{\textbf{eval}}(C(V_1,\ldots,V_n),\text{State}) = C(\textit{\textbf{eval}}(V_1,\text{State}),\ldots,\textit{\textbf{eval}}(V_n,\text{State})) =$$
$$= \{\langle C(\alpha_{11},\ldots,\alpha_{n1}), P_{11}\wedge\ldots\wedge P_{n1}\rangle,\ldots,\langle C(\alpha_{1m},\ldots,\alpha_{nr}), P_{1m}\wedge\ldots\wedge P_{nr}\rangle\},$$

In what follows we denote such result by

$$\{\langle\chi_1,Q_1\rangle,\ldots,\langle\chi_n,Q_n\rangle\}$$

Since the Q_j terms, $1\leqslant j\leqslant n$, are mutually exclusive by construction PC implies ***eval***(C $(V_1,\ldots,V_n)$, State) iff it implies both one χ_j and the corresponding Q_j:

$$\begin{aligned}&\text{PC}\Rightarrow \textit{\textbf{eval}}(C(V_1,\ldots,V_n),\text{State}) =\\ &\text{PC}\Rightarrow \{\langle\chi_1,Q_1\rangle,\ldots,\langle\chi_n,Q_n\rangle\} =\\ &\text{PC}\Rightarrow \textstyle\bigvee_{j=1}^{n}\chi_j\wedge Q_j\end{aligned}$$

In the same way we have:

$$\begin{aligned}&\text{PC}\Rightarrow \textit{\textbf{eval}}(\neg C(V_1,\ldots,V_n),\text{State}) =\\ &\text{PC}\Rightarrow \{\langle\neg\chi_1,Q_1\rangle,\ldots,\langle\neg\chi_n,Q_n\rangle\} =\\ &\text{PC}\Rightarrow \textstyle\bigvee_{j=1}^{n}\neg\mathscr{X}_j\wedge Q_j\end{aligned}$$

Thus, the definition of function ***exec*** for the conditional statement is:

$$\textit{\textbf{exec}}(\textbf{if } C \textbf{ then } S1 \textbf{ else } S2 \textbf{ end if},\langle\text{State,PC}\rangle) =$$

$$\textit{if}\ \text{PC}\Rightarrow \textstyle\bigvee_{j=1}^{n}\chi_j\wedge Q_j\quad \textit{\textbf{then exec}}(\text{S1},\langle\text{State,PC}\rangle)$$

$$\textit{\textbf{elsif}}\ \text{PC}\Rightarrow \textstyle\bigvee_{j=1}^{n}\neg\chi_j\wedge Q_j\quad \textit{\textbf{then exec}}(\text{S2},\langle\text{State,PC}\rangle)$$

$$\textit{\textbf{otherwise}}\quad \textit{\textbf{exec}}\left(\text{S1},\left\langle\text{State,PC}\wedge \textstyle\bigvee_{j=1}^{n}\chi_j\wedge Q_j\right\rangle\right)\ \text{or}$$

$$\textit{\textbf{exec}}\left(\text{S2},\left\langle\text{State,PC}\wedge \textstyle\bigvee_{j=1}^{n}\neg\chi_j\wedge Q_j\right\rangle\right)$$

If the current assumptions stored in PC are strong enough to satisfy the (negation of the) condition the **then** (**else**) branch of the conditional statement is executed; otherwise, either of the branches is selected by adding the corresponding assumption to PC.

Loop statement. When a loop statement of the form **while** C **loop** S **end loop** is reached, the boolean expression C is evaluated to determine whether the loop is executed or not. Such evaluation is performed as for the conditional statement. Thus, the definition of function ***exec*** for loop statement is:

$exec(\textbf{while}\ C\ \textbf{loop}\ S\ \textbf{end loop}, \langle State, PC\rangle) =$

$\textit{if}\ PC \Rightarrow \bigvee_{j=1}^{n} \chi_j \wedge Q_j \quad \textit{then}\ exec(\textbf{while}\ C\ \textbf{loop}\ S\ \textbf{end loop}, exec(S, \langle State, PC\rangle))$

$\textit{elsif}\ PC \Rightarrow \bigvee_{j=1}^{n} \neg\chi_j \wedge Q_j \quad \textbf{then}\ \langle State, PC\rangle$

$\textit{otherwise}\quad exec(\textbf{while}\ C\ \textbf{loop}\ S\ \textbf{end loop}, exec(S, \langle State, PC \wedge \bigvee_{j=1}^{n} \chi_j \wedge Q_j\rangle))$ or

$\langle State, PC \wedge \bigvee_{j=1}^{n} \neg\chi_j \wedge Q_j\rangle$

3. A PRACTICAL TECHNIQUE FOR SYMBOLIC EXECUTION

In this section we introduce a technique to handle effectively symbolic execution involving complex variables. The technique is based on an incremental representation of the program state. Every value-set may assume two different forms called the *canonical* and the *non-canonical* form, respectively. The non-canonical form allows to represent incrementally the program state through a data structure called *picture*.

3.1. Forms of the value-set

Since the value of any variable is needed only when a decision point is reached, the execution of assignment statements does not require the computation of the actual value-set. The execution of an assignment statement can be performed by temporarily storing the information about the variables involved. Thus, the computation of a value-set can be postponed until it is needed to solve a decision point.

For example, let us consider the following program:

State: $\{\langle J, \mathcal{J}\rangle, \langle X, \mathcal{X}\rangle, \langle Y, \mathcal{Y}\rangle, \langle Z, \mathcal{Z}\rangle, \langle A, \mathcal{A}\rangle\}$
PC: . . .

```
1 X:=A(J);
2 A(J − 1):=X + 1;
3 Z:=A(X) − X;
4 Y:=A(X) + 1;
5 if Z > 0 then . . .
```

When executing statement at line 1 we represent the value of X by storing how it is computed, i.e. X's value is equal to the value of the Jth element of A, where both J's and A's value are those holding at line 1. Therefore, denoting X's value by $[\mathcal{A}, \mathcal{J}]$ the problem state becomes:

State: $\{\langle J, \mathcal{J}\rangle, \langle X, [\mathcal{A}, \mathcal{J}]\rangle, \langle Y, \mathcal{Y}\rangle, \langle Z, \mathcal{Z}\rangle, \langle A, \mathcal{A}\rangle\}$

The assignment at line 2 refers to an indexed variable: the value of the J − 1st element of A is equal to the value of X + 1. Denoting such value by $[\mathcal{A}, \mathcal{J} - 1, [\mathcal{A}, \mathcal{J}] + 1]$ the program state becomes:

State: $\{\langle J, \mathcal{J}\rangle, \langle X, [\mathcal{A}, \mathcal{J}]\rangle, \langle Y, \mathcal{Y}\rangle, \langle Z, \mathcal{Z}\rangle, \langle A, [\mathcal{A}, \mathcal{J} - 1, [\mathcal{A}, \mathcal{J}] + 1]\rangle\}$

We represent in the same way the value of Z and Y at lines 3 and 4 respectively. Thus, the program state after line 4 is:

State: $\{\langle J, \mathcal{J}\rangle, \langle X, [\mathcal{A}, \mathcal{J}]\rangle, \langle Y, [[\mathcal{A}, \mathcal{J} - 1, [\mathcal{A}, \mathcal{J}] + 1], [\mathcal{A}, \mathcal{J}]] + 1\rangle,$
$\langle Z, [[\mathcal{A}, \mathcal{J} - 1, [\mathcal{A}, \mathcal{J}] + 1], [\mathcal{A}, \mathcal{J}]] - [\mathcal{A}, \mathcal{J}]\rangle, \langle A, [\mathcal{A}, \mathcal{J} - 1, [\mathcal{A}, \mathcal{J}] + 1]\rangle\}$

The execution of the conditional statement (line 5), requires to determine whether Z is greater than zero or not. Thus, the actual value-set of Z must be computed starting from its representation.

The previous example shows that during the execution of a program the value-set of simple variables has either of the following forms:

SValue_set:: = $\{\langle value, SCond\rangle^+\}$ | [CValue_set,SValue_set]

The former is called the canonical form while the latter is the non-canonical. The canonical form is obtained from the non-canonical one by taking the SValue_set th element of the indexed variable having CValue_set as value-set. Similarly, the value-set of an indexed variable has either of the following forms:

$$\text{CValue\_set} ::= \{\langle \text{value,CCond}\rangle^{+}\} \mid [\text{CValue\_set,SValue\_set1,SValue\_set2}]$$

The former is called the canonical form while the latter is the non-canonical. Here, we obtain the canonical form by assigning the value SValue_set2 to the SValue_set1 th element of the indexed variable having value-set CValue_set. The usage of non-canonical value-sets decreases the computation required to symbolically execute each statement. In the previous example, for instance, when the decision point is reached the value of Y is not needed. Thus, the test can be solved without computing the canonical form of the value-set of Y.

3.2. Pictures

Let us consider an assignment of the form $V_i := \text{Expr}(V_1, \ldots, V_n)$, where each V_i can be simple or indexed. Using the terminology of dataflow analysis [15] we say that variables appearing in the right hand side of the assignment $(V_1, \ldots, V_n)$ are *used*, while the variable appearing in the left hand side of the assignment (V_i) is *defined*. Note that by definition every time a variable is defined it receives a new value. As a consequence, the execution of a statement does not change the value of the variables that are not defined by that statement. The value of a variable is said to be *live* until a new definition of that variable occurs. Therefore, the value of a variable is likely not to change for some statements, i.e. the value remains live. For instance, in the previous example we note that the value assigned to X at line 1 does not change until line 5. Moreover, X's value is used in lines 2, 3 and 4 to assign a new value to A, Z and Y, respectively.

The value of A, Z and Y can be described through a reference to the value of X at line 1. In other words, at line 1 we *take a picture* of X that will be used to represent the value of A, Z and Y.

Every time a variable is defined a new picture is taken; every time a variable is used a reference to the last picture is made.

Note that each picture may refer to the previous ones and remains live until a new one is taken.

For example, let us consider the previous program; we denote by P_{iX}, the i th picture of X. Let the initial symbolic state be:

State: $\{\langle J,\{\langle\varphi,\text{true}\rangle\}\rangle,\langle X,\{\langle\text{undef},\text{true}\rangle\}\rangle,\langle Y,\{\langle\text{undef},\text{true}\rangle\}\rangle,$
$\langle Z,\{\langle\text{undef},\text{true}\rangle\}\rangle,\langle A,\{\langle I,\text{true}\rangle\}\rangle\}$
PC: true

the program state evolves in the following way.

1	$X := A(J);$	$\langle X,[P_{1A},P_{1J}]\rangle$	$P_{1J} = \{\langle\varphi,\text{true}\rangle\}, P_{1A} = \{\langle I,\text{true}\rangle\}$
2	$A(J-1) := X;$	$\langle A,[P_{1A},P_{1J}-1,P_{1X}]\rangle$	$P_{1X} = [P_{1A},P_{1J}]$
3	$Z := A(X) - X;$	$\langle Z,[P_{2A},P_{1X}] - P_{1X}\rangle$	$P_{2A} = [[P_{1A},P_{1J}-1,P_{1X}],P_{1J}+1,P_{1X}+1]$
4	$Y := A(X) + 1;$	$\langle Y,[P_{2A},P_{1X}] + 1\rangle$	
5	**if** $Z > 0$ **then** ...		

The program state at line 5 is:

State: $\{\langle X,[P_{1A},P_{1J}]\rangle,\langle A,[P_{1A},P_{1J}-1,P_{1X}]\rangle,\langle Z,[P_{2A},P_{1X}]-P_{1X}\rangle,\langle Y,[P_{2A},P_{1X}]+1\rangle$
Picture: $P_{1J} = \{\langle\varphi,\text{true}\rangle\},$
$P_{1A} = \{\langle I,\text{true}\rangle\},$
$P_{1X} = [P_{1A},P_{1J}],$
$P_{2A} = [[P_{1A},P_{1J}-1,P_{1X}],P_{1J}+1,P_{1X}+1]$

Figure 1 depicts the evolution of the program state by showing the value of every variable before the execution of each statement.

A picture can be discarded when neither the program state nor other pictures have references to it. As a consequence, a suitable garbage collection mechanism will prevent an exessive growth of the stored information.

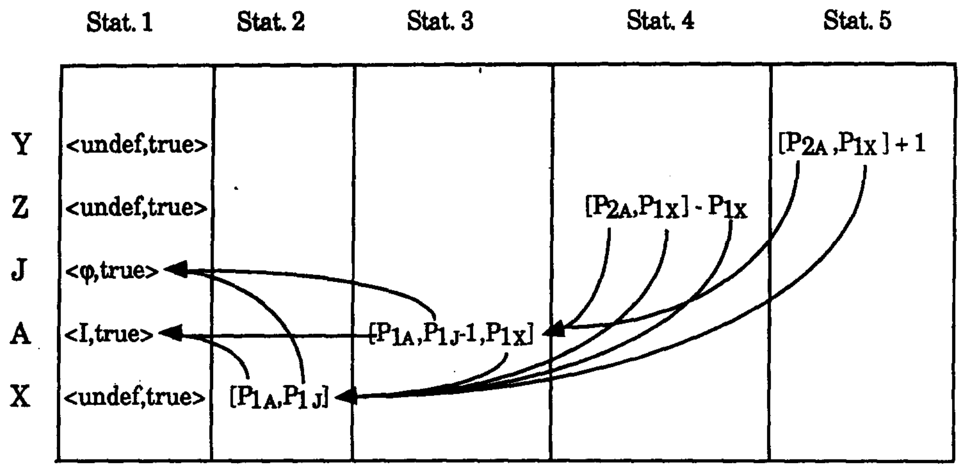

Fig. 1. The evolution of the program state.

3.3. *Derivation of canonical forms*

Whenever a decision point is reached, the evaluation of the condition requires the canonical form of the referenced variables. Let us consider the following non-canonical value-set of a simple variable:

$$[\{\langle\alpha_1,P_1(I)\rangle,\ldots,\langle\alpha_m,P_m(I)\rangle\},\{\langle\beta_1,Q_1\rangle,\ldots,\langle\beta_n,Q_n\rangle\}]$$

The corresponding canonical form is:

$$\{\langle\alpha_1,P_1(\beta_1)\wedge Q_1\vee\ldots\vee P_1(\beta_n)\wedge Q_n\rangle,\ldots,\langle\alpha_m P_m(\beta_1)\wedge Q_1\vee\ldots\vee P_m(\beta_n)\wedge Q_{ln}\rangle\}$$

Analogously, denoting the non-canonical value set of an indexed variable by:

$$[\{\langle\alpha_1,P_1(I)\rangle,\ldots,\langle\alpha_m,P_m(I)\rangle\},\{\langle\beta_1,Q_1\rangle,\ldots,\langle\beta_n,Q_n\rangle\},\{\langle\chi_1,R_1\rangle,\ldots,\langle\chi_t,R_t\rangle\}]$$

its canonical form is given by:

$$\{\langle\alpha_1,P_1(I)\wedge I\neq\beta_1\wedge\ldots\wedge I\neq\beta_n\rangle,\ldots,\langle\alpha_m,P_m(I)\wedge I\neq B_1\wedge\ldots\wedge I\neq\beta_n\rangle\}\oplus$$
$$\{\langle\chi_1,R_1\wedge(I=\beta_1\vee\ldots\vee I=\beta_n)\rangle,\ldots,\langle\chi_t,R_t\wedge(I=\beta_1\vee\ldots\vee I=\beta_n)\rangle\}$$

For example, the evaluation of the decision point (line 5) of the previous example requires the canonical form of the value-set of Z. The non-canonical form is:

$$Z\text{: } [P_{2A},P_{1X}]-P_{1X}$$

First, we should compute the canonical form of P_{1X} and P_{2A}:

$$P_{1X}\text{: } [P_{1A},P_{1J}]=[\{\langle I,\text{true}\rangle\},\{\langle\varphi,\text{true}\rangle\}]=\{\langle\varphi,\text{true}\rangle\},$$
$$P_{2A}\text{: } [[P_{1A},P_{1J}-1,P_{1X}],P_{1J}+1,P_{1X}+1]=$$
$$[[P_{1A},\{\langle\varphi-1,\text{true}\rangle\},\{\langle\varphi,\text{true}\rangle\}],\{\langle\varphi+1,\text{true}\rangle\},\{\langle\varphi+1,\text{true}\rangle\}]=$$
$$[\{\langle I,I\neq\varphi-1\rangle,\langle\varphi,I=\varphi-1\rangle\},\{\langle\varphi+1,\text{true}\rangle\},\{\langle\varphi+1,\text{true}\rangle\}]=$$
$$\{\langle I,I\neq\varphi-1\wedge I\neq\varphi+1\rangle,\langle\varphi,I=\varphi-1\wedge I\neq\varphi+1\rangle,\langle\varphi+1,I=\varphi+1\rangle\}$$

Thus, the canonical form of the value-set of Z is:

$$Z\text{: } [\{\langle I,I\neq\varphi-1\wedge I\neq\varphi+1\rangle,\langle\varphi,I=\varphi-1\wedge I\neq\varphi+1\rangle,\langle\varphi+1,I=\varphi+1\rangle\},\langle\varphi,\text{true}\rangle]-$$
$$\{\langle\varphi,\text{true}\rangle\}=\{\langle 0,\text{true}\rangle\}$$

and the program state at line 5 becomes:

State: $\{\langle J,\{\langle\varphi,\text{true}\rangle\}\rangle,\langle X,P_{1X}\rangle,\langle Y,[P_{2A},P_{1X}]+1\rangle,\langle Z,\{\langle 0,\text{true}\rangle\}\rangle,\langle A,P_{2a}\rangle\}$,
Picture: $P_{1X}=\{\langle\varphi,\text{true}\rangle\}$,
$P_{2A}=\{\langle I,I\neq\varphi-1\wedge I\neq\varphi+1\rangle,\langle\varphi,I=\varphi-1\wedge I\neq\varphi+1\rangle,\langle\varphi+1,I=\varphi+1\rangle\}$

Notice that pictures P_{1J} and P_{1A} were discarded since no more reference to them exists.

Below, we show in what cases it is possible to evaluate an expression without computing its canonical value-set. Let us consider the following fragment of code:

State: $\{\langle A,\mathscr{A}\rangle,\langle B,\mathscr{B}\rangle,\langle C,\mathscr{C}\rangle,\langle D,\mathscr{D}\rangle\}$,
A(C):=B;
if A(D) . . .

When the decision statement is reached the program state is:

State: $\{\langle A,[\mathscr{A},\mathscr{B},\mathscr{C}]\rangle,\langle B,\mathscr{B}\rangle,\langle C,\mathscr{C}\rangle,\langle D,\mathscr{D}\rangle\}$,

The value of A(D) is:

$[[\mathscr{A},\mathscr{B},\mathscr{C}],\mathscr{D}]$

Three different cases should be considered to compute the canonical form of A(D):

(1) C and D have the same value*: whenever this happens the value of A(D) is equal to $\mathscr{B}$;
(2) C and D haven't the same value†: the assignment A(C):=B has not affected the value of A(D). Thus A(D) is equal to $[\mathscr{A},\mathscr{D}]$;
(3) C and D may or may not the same value‡: in such a case it is necessary to compute the canonical form of $[[\mathscr{A},\mathscr{B},\mathscr{C}],\mathscr{D}]$.

For example, let us consider the value-set of Z: $[P_{2A},P_{1X}]-P_{1X}$. To compute such value-set it is necessary to evaluate the Xth element of A: $[P_{2A},P_{1X}]$.

$[[[P_{1A},\{\langle\varphi-1,true\rangle\},\{\langle\varphi,true\rangle\}],\{\langle\varphi+1,true\rangle\},\{\langle\varphi+1,true\rangle\}],\{\langle\varphi,true\rangle\}]$

Since φ does not equal $\varphi+1$, the value of A(X) is given by:

$[[P_{1A},\{\langle\varphi-1,true\rangle\},\{\langle\varphi,true\rangle\}],\{\langle\varphi,true\rangle\}]$

Since φ does not equal $\varphi-1$, the value of A(X) is given by:

$[P_{1A},\{\langle\varphi,true\rangle\}]\rightarrow[\{\langle I,true\rangle\},\{\langle\varphi,true\rangle\}]\rightarrow\{\langle\varphi,true\rangle\}$

Thus, the value-set of Z is:

$\{\langle\varphi,true\rangle\}-\{\langle\varphi,true\rangle\}=\{\langle 0,true\rangle\}$

and the program state holding at line 5 is:

State: $\{\langle J,P_{1J}\rangle,\langle X,P_{1X}\rangle,\langle Y,[P_{2A},P_{1X}]+1\rangle,\langle Z,\{\langle 0,true\rangle\}\rangle,\langle A,P_{2A}\rangle\}$, where
$P_{1J}=\{\langle\varphi,true\rangle\}$, $P_{1X}=\{\langle\varphi,true\rangle\}$,
$P_{2A}=[[P_{1A},\{\langle\varphi-1,true\rangle\},\{\langle\varphi,true\rangle\}],\{\langle\varphi+1,true\rangle\},\{\langle\varphi+1,true\rangle\}]$
$P_{1A}=\{\langle I,true\rangle\}$

3.4. The evaluation of decision points

Let C be a boolean expression governing a decision point. We denote by PC the value of the path condition and by ν: $\{\langle\alpha_1,P_1\rangle,\ldots,\langle\alpha_n,P_n\rangle\}$ the value-set of C.

As discussed in Section 2.1.2, the **then** branch of a conditional statement is executed if PC implies $\bigvee_{j=1}^{n}\alpha_j\wedge P_j$. This can be rewritten as:

$$PC\Rightarrow\bigvee_{j=1}^{n}\alpha_j\wedge P_j\equiv\bigvee_{j=1}^{n}PC\Rightarrow\alpha_j\wedge P_j\equiv\bigvee_{j=1}^{n}(PC\Rightarrow\alpha_j)\wedge(PC\Rightarrow P_j)$$

Therefore, we should evaluate $PC\Rightarrow\alpha_j$ to determine whether α_j satisfies C, and $PC\Rightarrow P_j$ to determine whether α_j is a feasible value under the current assumptions. At this purpose, we

*Given two value-sets: $\mathscr{A}$:$\{\langle\alpha_1,P_1\rangle,\ldots,\langle\alpha_m,P_m\rangle\}$ and $\mathscr{B}$:$\{\langle\beta_1,Q_1\rangle,\ldots,\langle\beta_n,Q_n\rangle\}$, we say that $\mathscr{A}$ is equal to $\mathscr{B}$ iff $n=m\wedge\forall i|1\leqslant i\leqslant n(\alpha i=\beta i)$.

†Given two value-sets $\mathscr{A}$:$\{\langle\alpha_1,P_1\rangle,\ldots,\langle\alpha_m,P_m\rangle\}$ and $\mathscr{B}$:$\{\langle\beta_1,Q_1\rangle,\ldots,\langle\beta_n,Q_n\rangle\}$, we say that $\mathscr{A}$ is not equal to $\mathscr{B}$ iff $\forall i,j|1\leqslant j\leqslant n\wedge 1\leqslant i\leqslant n(\alpha j\neq\beta i)$.

‡Given two value-sets $\mathscr{A}$ and $\mathscr{B}$ we say that $\mathscr{A}$ and $\mathscr{B}$ may or may not be equal iff they are neither equal nor not equal.

introduce two vectors, called the *value_vector* (V_v) and the *index_vector* (I_v), made up of the results of such evaluations. The former is defined as follows:

$$V_v = [v_1 \ldots v_n], \quad \text{where} \quad \begin{array}{l} v_j = \text{true iff } PC \Rightarrow \alpha_j \text{ or} \\ v_j = \text{false iff } PC \Rightarrow \neg\alpha_j \text{ or} \\ v_j = \alpha_j \text{ otherwise.} \end{array}$$

while the latter is:

$$I_v = [i_1 \ldots i_n], \quad \text{where} \quad \begin{array}{l} i_j = \text{true iff } PC \Rightarrow P_j \text{ or} \\ i_j = \text{false iff } PC \Rightarrow \neg P_j \text{ or} \\ i_j = P_j \text{ otherwise.} \end{array}$$

Since P_j terms are mutually exclusive I_v can contain at most one *true*. As a consequence, if an i_j equals *true* all the others must equal *false* and should not be evaluated. If an i_j evaluates to *false* then the evaluation of the corresponding v_j is not needed.

The expression $\bigvee_{j=1}^{n} (v_j \wedge i_j)$ represents the condition that must be added to PC to satisfy the condition of the decision statement. In the same way the expression $\bigvee_{j=1}^{n} (\neg v_j \wedge i_j)$ represents the condition that must be added to PC to satisfy the negation of the condition of the decision statement.

3.5. *Symbolic execution algorithms*

In what follows we describe the algorithms used to perform an assignment statement and provide the value-set of an expression in canonical form. We denote by New_def(X) a boolean stating whether variable X was defined but not used, and by Simple(X) a boolean stating whether variable X is simple.

The following algorithm describes the execution of the assignment $X := Exp(Y_1, \ldots, Y_n)$ (represented in a semi-formal notation):

```
Perform(X:=Exp(Y_1, ..., Y_n)) is
  for each Yi in Exp do
    if New_def(Yi) then
      New_def(Yi):=false;
      Add_picture(Yi,VSet(Yi));
    end if;
  end for;
  Update_program_state(X,Exp(Vset(Y1), ..., Vset(Yn)));
  New_def(X):=true;
end Perform;
```

where Add_picture(Y,P) inserts the picture P of X in a suitable data structure. VSet(X) returns the Value_set currently associated to X in the program state:

```
VSet(X) is
  if Simple(X) then
    return(The value-set associated to X in Program State)
  else
    Let Val_J be the value-set associated to the index of X;
    return([The value-set associated to X in Program State, Val_J]);
  end if;
end Vset;
```

Finally, Update_program_state(X,V), updates the program state with the new Value_set V.

```
Update_program_state(X, VSet) is
  if Simple(X) then
    Substitute(X, Vset);
  else
    Let VSet_J be the value-set of the index of X;
    Old_Vset_X:=VSet(X);
    Substitute(X,[Old_Vset_X,VSet_J,Vset]);
 end if;
 end Update_program_state
```

where Substitute(X,Vset) replaces the current value-set of X with Vset. The following two algorithms return the actual value of a value_set V of simple and indexed variables, respectively.

```
 EvaluateS(V) is     -- For simple variables
   case form(V) in
     1) return(V);                         -- {⟨value,Scond⟩$^+$}
     2) return(Eval(CV,EvaluateS(SV)));    -- [CV,SV]
   end case
   end EvaluateS;

Eval(CV,SV) is      -- SV is in canonical form
  case form (CV) in
    1) return(eval(CV,SV)); -- The eval function is the one defined in Section 2.1.1
    2) if Equal_Vset(EvaluateS(I),SV)  -- CV = [CV1,I,E]
         then return(E);
         elsif Different_Vset(EvaluateS(I),SV)
         then return(Evaluate_S[(CV1,SV]);
         else return(eval(EvaluateC(CV),SV));
       end if
    end case
  end Eval

EvaluateC(V) is     -- For indexed variables
  case form(V) in
    1) return(V);                          -- {⟨value,Ccond⟩$^+$}
    2) Index_VSet:=EvaluateS(SV1);
       return((EvaluateC(CV)/Index_VSet)$\oplus$
         (EvaluateS(SV2)$\otimes$Index_VSet));  -- [CV,SV1,SV2]
  end case
end EvaluateC;
```

where operator $\oplus$ is defined in Section 2.1 and /, $\otimes$ allow to derive the value-set $\mathscr{A}'$ and $\mathscr{E}'$ of Section 2.1, respectively.

Finally, given a value-set v: $\{\langle\alpha_1,P_1\rangle,\ldots,\langle\alpha_n,P_n\rangle\}$ of the expression of a decision point, the condition that must be added to PC to execute the **then** branch, if any, is computed in the following way:

```
 True_cond(V) is     -- For simple variables
   Result:=false;
   for each Pi in V do
     if PC $\Rightarrow$ Pi then return(Cond($\alpha$i));
     elsif PC $\Rightarrow$ $\neg$Pi then continue;
     else Result:=Result $\vee$ (Pi $\wedge$ Cond($\alpha$i));
     end if
   end for
   return(Result);
 end True_cond
```

```
Cond(C) is
  if PC ⇒ C then return(true);
  elsif PC ⇒ ¬C then return(false);
  else return(C);
  end if
end Cond
```

4. TWO EXAMPLES

This section discusses two examples of symbolic execution with indexed variables. Both programs sort an array, but they are based on different algorithms. The former is symbolically tested, while the latter is formally verified to be correct with respect to its assertions.

4.1. An example of application of symbolic execution to testing

The procedure SORT_1 takes an input an array of integer B of length N, and procedures as output an array C which is a sorted copy of B.

We symbolically execute this procedure under the initial assumption that the input array is already sorted, but in reverse order. All variables, but B and I (input parameters) have an undefined initial value. Thus, the initial symbolic state is:

State: $\{\langle A,\{\langle \text{undef},\text{true}\rangle\}\rangle,\langle B,\{\langle \beta.I,\text{true}\rangle\}\rangle,\langle C,\{\langle \text{undef},\text{true}\rangle\}\rangle,$
$\langle N,\{\langle \eta,\text{true}\rangle\}\rangle,\langle X,\{\langle \text{undef},\text{true}\rangle\}\rangle,\langle J,\{\langle \text{undef},\text{true}\rangle\}\rangle\}$
PC: $\forall I \in (1 \ldots \eta - 1)(\beta.I \rangle \beta.I\rangle + 1)$

```
type INTEGER_ARRAY is array (INTEGER range<>) of INTEGER;
procedure SORT_1(N: in POSITIVE; B: in INTEGER_ARRAY;
                 C: out INTEGER_ARRAY) is

X, J: INTEGER;
A:  INTEGER_ARRAY(0...N);
0   begin
1     for I in 1...N loop
2       A(I):=B(I);
3     end loop;
4     for I in 2...N loop
5       X:=A(I);
6       A(0):=X;
7       J:=I − 1;
8       while X < A(J) loop
9         A(J + 1):=A(J);
10        J:=J − 1;
11      end loop;
12      A(J + 1):=X;
13    end loop;
14    for I in 1···N loop
15      C(I):=A(I);
16    end loop;
17  end SORT_1;
```

At line 4 the symbolic state becomes:

State: $\{\langle A,\{\langle \text{undef},I = 0\rangle,\langle \beta.I,I \geqslant 1 \wedge I \leqslant \eta\rangle\}\rangle,\langle B,\{\langle \beta.I,\text{true}\rangle\}\rangle,\langle C,\{\langle \text{undef},\text{true}\rangle\}\rangle,$
$\langle N,\{\langle \eta,\text{true}\rangle\}\rangle,\langle X,\{\langle \text{undef},\text{true}\rangle\}\rangle,\langle J,\{\langle \text{undef},\text{true}\rangle\}\rangle\}$
PC: $\forall I \in (1 \ldots \eta)(\beta.I > \beta.I + 1)$

A new pair representing the index of the **for** loop of line 4 ($\langle I,\{\langle 2,\text{true}\rangle\}\rangle$) is added to State. To execute the loop it is necessary to determine whether under the current assumptions the value of

I is less or equal than the value of N, i.e. PC $\Rightarrow 2 \leqslant \eta$. Since the answer is *neither*, the loop is executed under the assumption $2 \leqslant \eta$. Thus, the symbolic state becomes:

State: $\{\langle A,\{\langle undef, I=0\rangle,\langle \beta.I, I \geqslant 1 \wedge I \leqslant \eta\rangle\}\rangle,\langle B,\{\langle \beta.I,true\rangle\}\rangle,\langle C,\{\langle undef,true\rangle\}\rangle,$
$\langle N,\{\langle \eta,true\rangle\}\rangle,\langle X,\{\langle undef,true\rangle\}\rangle,\langle J,\{\langle undef,true\rangle\}\rangle,\langle I,\{\langle 2,true\rangle\}\rangle\}$
PC: $\forall I \in (1 \ldots \eta)(\beta.I > \beta.I + 1) \wedge 2 \leqslant \eta$

Line 5 contains an assignment referring to A. Therefore the pictures of A and I are built. The symbolic state is:

State: $\{\langle A,P_{IA}\rangle,\langle B,\{\langle \beta.I,true\rangle\}\rangle,\langle C,\{\langle undef,true\rangle\}\rangle$
$\langle N,\{\langle \eta,true\rangle\}\rangle,\langle X,[P_{IA},P_{II}]\rangle,\langle J,\{\langle undef,true\rangle\}\rangle,\langle I,P_{II}\rangle\}$
PC: $\forall I \in (1 \ldots \eta)(\beta.I > \beta.I + 1) \wedge 2 \leqslant \eta$
Picture: $P_{IA} = \{\langle undef, I=0\rangle,\langle \beta.I, I \geqslant 1 \wedge I \leqslant \eta\rangle\}$
$P_{II} = \{\langle 2,true\rangle\}$

Lines 6 and 7 contain two further assignments. At line 8 the symbolic state is:

State: $\{\langle A,[P_{IA}\{\langle 0,true\rangle\},P_{IX}]\rangle,\langle B,\{\langle \beta.I,true\rangle\}\rangle,\langle C,\{\langle undef,true\rangle\}\rangle,$
$\langle N,\{\langle \eta,true\rangle\}\rangle,\langle X,[P_{IA},P_{II}]\rangle,\langle J,P_{II}-1\rangle,\langle I,P_{II}\rangle\}$
PC: $\forall I \in (1 \cdots \eta)(\beta.I \rangle \beta.I + 1) \wedge 2 \leqslant \eta$

Picture: $P_{IA} = \{\langle undef, I=0\rangle,\langle \beta.I, I \geqslant 1 \wedge I \leqslant \eta\rangle\}$
$P_{II} = \{\langle 2,true\rangle\}$
$P_{IX} = [P_{IA},P_{II}]$

Figure 2 depicts the evolution of the program state by showing the value-sets holding at line 4, 5 and 8.

The evaluation of the condition at line 8 involves variables X, J and A. Thus, it is necessary to compute the canonical form of the value-sets of such variables.

X: $[P_{IA},P_{II}] = [\{\langle undef, I=0\rangle,\langle \beta.I, I \geqslant 1 \wedge I \leqslant \eta\rangle\},\{\langle 2,true\rangle\}] = \{\langle \beta.2,true\rangle\}\rangle$

A: $[P_{IA},\{\langle 0,true\rangle\},P_{IX}] = [\{\langle undef, I=0\rangle,\langle \beta.I, I \geqslant 1 \wedge I \leqslant \eta\rangle\},\{\langle 0,true\rangle\},\{\langle \beta.2,true\{\langle] =$
$\{\langle \beta.2, I=0\rangle,\langle \beta.I, I \geqslant 1 \wedge I \leqslant \eta\rangle\}$

J: $P_{II} - 1 = \{\langle 1,true\rangle\}$

The value vector and the index vector are respectively;

$V_v = [PC \Rightarrow (\beta.2 < \beta.2), PC \Rightarrow (\beta.2 < \beta.1)] \rightarrow [false, true]$
$I_v = [PC \Rightarrow true \wedge 1 = 0, PC \Rightarrow true \wedge 1 \geqslant 1 \wedge 1 \leqslant \eta] \rightarrow [false, true]$

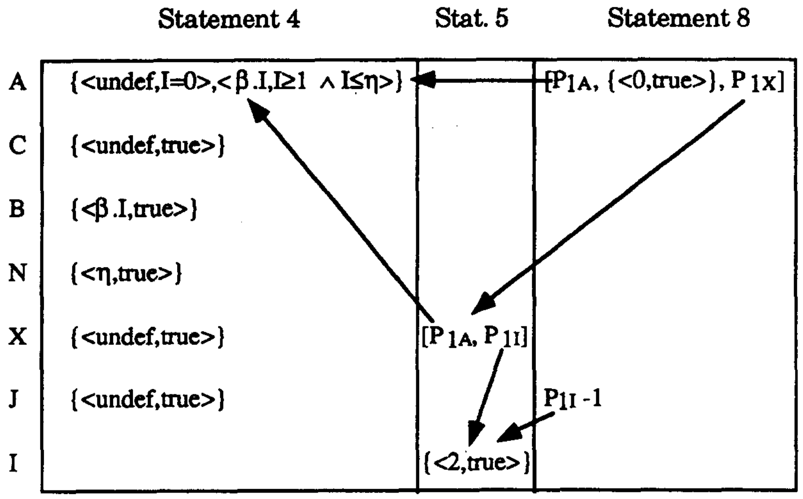

Fig. 2. The evolution of the program state until line 8.

Since $V_v \times I_v^T$ equals *true*, the loop is entered with the following symbolic state:

State: $\{\langle A,\{\langle\beta.2,I=0\rangle,\langle\beta.I,I \geqslant 1 \wedge I \leqslant \eta\rangle\}\rangle,\langle B,\{\langle\beta.I,true\rangle\}\rangle,\langle C,\{\langle undef,true\rangle\}\rangle,$
$\langle N,\{\langle\eta,true\rangle\}\rangle,\langle X,\{\langle\beta.2,true\rangle\}\rangle,\langle J,\{\langle 1,true\rangle\}\rangle,\langle I,P_{II}\rangle\}$
PC: $\forall I \in (1 \ldots \eta)(\beta.I > \beta.I + 1) \wedge 2 \leqslant \eta$
Picture: $P_{II} = \{\langle 2,true\rangle\}$

When the execution reaches lines 8 for the second time the symbolic state is (see also Fig. 3):

State: $\{\langle A,[P_{2A},P_{IJ}+1,[P_{2A},P_{IJ}]]\rangle,\langle B,\{\langle\beta.I,true\rangle\}\rangle,\langle C,\{\langle undef,true\rangle\}\rangle,$
$\langle N,\{\langle\eta,true\rangle\}\rangle,\langle X,\{\langle\beta.2,true\rangle\}\rangle,\langle J,P_{IJ}-1\rangle,\langle I,P_{II}\rangle\}$
PC: $\forall I \in (1 \ldots \eta)(\beta.I > \beta.I + 1) \wedge 2 \leqslant \eta$

Picture: $P_{II} = \{\langle 2,true\rangle\}$
$P_{2A} = \{\langle\beta.2,I=0\rangle,\langle\beta.I,I \geqslant 1 \wedge I \leqslant \eta\rangle\}$
$P_{IJ} = \{\langle 1,true\rangle\} - 1$

Once again the canonical form of the value-set of A and J has to be computed:

A: $[P_{2A},P_{IJ}+1,[P_{2A},P_{IJ}]] =$
$[\{\langle\beta.2,I=0\rangle,\langle\beta.I,I \geqslant 1 \wedge I \leqslant \eta\rangle\},\{\langle 1,true\rangle\}+1,[\{\langle\beta.2,I=0\rangle,$
$\langle\beta.I,I \geqslant 1 \wedge I \leqslant \eta\rangle\},\{\langle 1,true\rangle\}]] =$
$[\{\langle\beta.2,I=0\rangle\langle\beta.I,I \geqslant 1 \wedge I \leqslant \eta\rangle\},\{\langle 1,true\rangle\}+1,\{\langle\beta.1,true\rangle\}] =$
$\{\langle\beta.2,I=0\rangle,\langle\beta.1,I=2\rangle,\langle\beta.I,I \geqslant 1 \wedge I \leqslant \eta \wedge I \neq 2\rangle\}$

J: $\{\langle 1,true\rangle\} - 1 = \{\langle 0,true\rangle\}$

The vectors V_v and I_v are:

$V_v = [PC \Rightarrow (\beta.2 < \beta.2), PC \Rightarrow (\beta.2 < \beta.1), PC \Rightarrow (\beta.2 < \beta.0]) \rightarrow [false,true,PC \Rightarrow (\beta.2 < \beta.0)]$
$I_v \ = [PC \Rightarrow true \wedge 0 = 0, PC \Rightarrow true \wedge 0 = 2, PC \Rightarrow true \wedge 0 \geqslant 1 \wedge 0 \leqslant \eta] \rightarrow [true,false,false]$

This time $V_v \times I_v^T$ equals *false* and thus, the execution leaves the **while** loop.

At line 4 the symbolic state is:

State: $\{\langle A,[P_{3A},P_{2J}+1,P_{2X}]\rangle,\langle B,\{\langle\beta.I,true\rangle\}\rangle,\langle C,\{\langle undef,true\rangle\}\rangle,\langle N,\{\langle\eta,true\rangle\}\rangle,$
$\langle X,\{\langle\beta.2,true\rangle\}\rangle,\langle J,\{\langle 0,true\rangle\}\rangle,\langle I,P_{II}+1\rangle\}$
PC: $\forall I \in (1 \ldots \eta)(\beta.I > \beta.I + 1) \wedge 2 \leqslant \eta$

Picture: $P_{II} = \{\langle 2,true\rangle\}$
$P_{3A} = \{\langle\beta.2,I=0\rangle,\langle\beta.1,I=2\rangle,\langle\beta.I,I \geqslant 1 \wedge I \leqslant \eta \wedge I \neq 2\rangle\}$

The value of PC does not imply the condition of the **for** loop, nor the opposite; thus, we assume that $3 \leqslant \eta$ and we enter the loop once more.

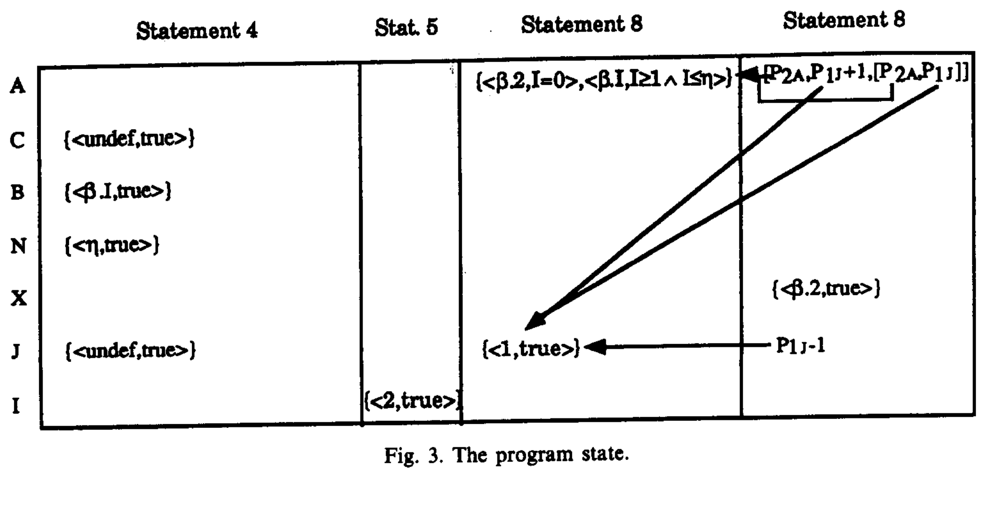

Fig. 3. The program state.

The symbolic state at line 8 is:

State: $\{\langle A,[P_{4A},\{\langle 0,true\rangle\},P_{3X}]\rangle,\langle B,\{\langle\beta.I,true\rangle\}\rangle,\langle C,\{\langle undef,true\rangle\}\rangle,$
$\langle N,\{\langle\eta,true\rangle\}\rangle,\langle X,P_{3X}\rangle,\langle J,P_{2I}-1\rangle,\langle I,P_{2I}\rangle\}$

PC: $\forall I\in(1\ldots\eta)(\beta.I>\beta.I+1)\wedge 3\leqslant\eta$

Picture: $P_{2I}=\{\langle 3,true\rangle\}$
$P_{3A}=\{\langle\beta.2,I=0\rangle,\langle\beta.1,I=2\rangle,\langle\beta.I,I\geqslant 1\wedge I\leqslant\eta\wedge I\neq 2\rangle\}$
$P_{2J}=\{\langle 0,true\rangle\}$
$P_{2X}=\{\langle\beta.2,true\rangle\}$
$P_{4A}=[P_{3A},P_{2J}+1,P_{2X}]$
$P_{3X}=[P_{4A},P_{1I}+1]$

The loop condition is **true** and the loop is entered. After the execution of lines 9 and 10 the symbolic state is:

State: $\{\langle A,[P_{5A},P_{3J}+1,[P_{5A},P_{3J}]]\rangle,\langle B,\{\langle\beta.I,true\rangle\}\rangle,\langle C,\{\langle undef,true\rangle\}\rangle,$
$\{N,\{\langle\eta,true\rangle\}\rangle,\langle X,\{\langle\beta.3,true\rangle\}\rangle,\langle J,P_{3J}-1\rangle,\langle I,P_{2I}\rangle\}$

PC: $\forall I\in(1\ldots\eta)(\beta.I>\beta.I+1)\wedge 3\leqslant\eta$

Picture: $P_{2I}=\{\langle 3,true\rangle\}$
$P_{3J}=\{\langle 2,true\rangle\}$
$P_{5A}=\{\langle\beta.3,I=0\rangle,\langle\beta.2,I=1\rangle,\langle\beta.1,I=2\rangle,\langle\beta.I,I\geqslant 1\wedge I\leqslant\eta\wedge I\neq 1\wedge I\neq 2\rangle\}$

The condition of loop at line 8 is *true*; thus, at line 11 the symbolic state is:

State: $\{\langle A,[P_{6A},P_{4J}+1,[P_{6A},P_{4J}]]\rangle,\langle B,\{\langle\beta.I,true\rangle\}\rangle,\langle C,\{\langle undef,true\rangle\}\rangle,$
$\langle N,\{\langle\eta,true\rangle\}\rangle,\langle X,\{\langle\beta.3,true\rangle\}\rangle,\langle J,P_{4J}-1\rangle,\langle I,P_{2I}\rangle\}$

PC: $\forall I\in(1\ldots\eta)(\beta.I>\beta.I+1)\wedge 3\leqslant\eta$

Picture: $P_{2I}=\{\langle 3,true\rangle\}$
$P_{4J}=\{\langle 1,true\rangle\}$
$P_{6A}=\{\langle\beta.3,I=0\rangle,\langle\beta.2,I=1\rangle,\langle\beta.1,I=2\vee I=3\rangle,$
$\langle\beta.I,I\geqslant 1\wedge I\leqslant\eta\wedge I\neq 1\wedge I\neq 2\wedge I\neq 3\rangle\}$

The **while** loop is left, the assignment at line 12 is executed and the symbolic state at line 4 becomes:

State: $\{\langle A,[P_{7A},P_{5J}+1,P_{4X}]\rangle,\langle B,\{\langle\beta.I,true\rangle\}\rangle,\langle C,\{\langle undef,true\rangle\}\rangle,\langle N,\{\langle\eta,true\rangle\}\rangle,$
$\langle X,P_{4X}\rangle,\langle J,P_{5J}\rangle,\langle I,P_{2I}+1\rangle\}$

PC: $\forall I\in(1\ldots\eta)(\beta.I>\beta.I+1)\wedge 3\leqslant\eta$

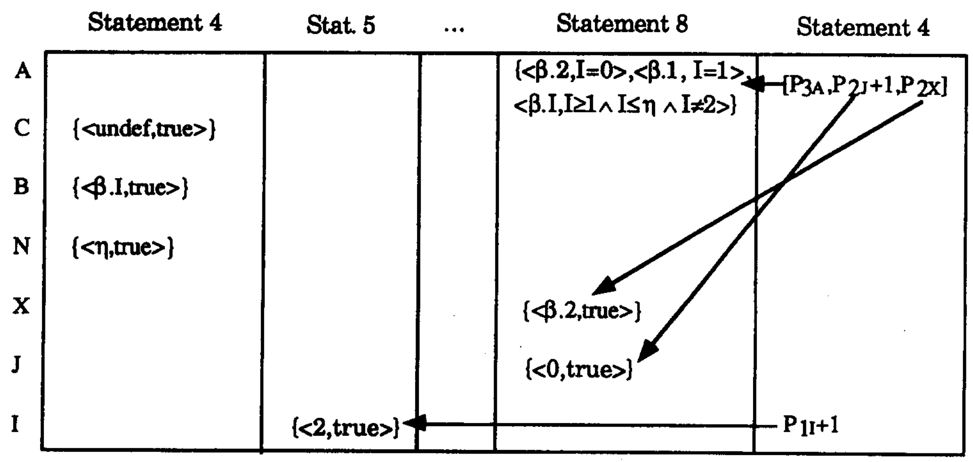

Fig. 4. The evolution of the program state.

Picture: $P_{2I} = \{\langle 3,\text{true}\rangle\}$
$P_{5J} = \{\langle 0,\text{true}\rangle\}$
$P_{4X} = \{\langle \beta.3,\text{true}\rangle\}$
$P_{7A} = \{\langle \beta.3, I = 0\rangle, \langle \beta.2, I = 1 \vee I = 2\rangle, \langle \beta.1, I = 3\rangle,$
$\langle \beta.I, I \geqslant 1 \wedge I \leqslant \eta \wedge I \neq 1 \wedge I \neq 2 \wedge I \neq 3\rangle\}$

The condition of the **for** loop is evaluated, and assuming that $4 > \eta$ the loop is left.

Thus, after the execution of the loop at lines 14–16 the symbolic state is:

State: $\{\langle A, P_{8A}\rangle, \langle B, \{\langle \beta.I,\text{true}\rangle\}\rangle, \langle C, \{\langle \beta.3, I = 1\rangle, \langle \beta.2, I = 2\rangle, \langle \beta.1, I = 3\rangle,$
$\langle \text{undef}, I \neq 1 \wedge I \neq 2 \wedge I \neq 3\rangle\}\rangle, \langle N, \{\langle \eta,\text{true}\rangle\}\rangle, \langle X, P_{4X}\rangle, \langle J, P_{5J}\rangle\}$
PC: $\forall I \in (1 \ldots \eta)(\beta.I > \beta.I + 1) \wedge 3 \leqslant \eta \wedge 4 > \eta$

Picture: $P_{5J} = \{\langle 0,\text{true}\rangle\}$
$P_{4X} = \{\langle \beta.3,\text{true}\rangle\}$
$P_{8A} = \{\langle \beta.3, I = 0\rangle, \langle \beta.3, I = 1\rangle, \langle \beta.2, I = 2\rangle, \langle \beta.1, I = 3\rangle,$
$\langle \beta.I, I \geqslant 1 \wedge I \leqslant \eta \wedge I \neq 1 \wedge I \neq 2 \wedge I \neq 3\rangle\}$

As expected, the array C contains the first three elements of B, but in reverse order.

4.2. An example of program verification

The procedure SORT_2 takes as input an array A, and an integer N representing its length, and sorts A. The Ada code contains, as comments, some first order predicates used to prove the (partial) correctness [16] of the procedure. Assuming that the *entry* clause holds before the execution of SORT_2 the program is correct if the *exit* clause after the execution does. The *assert* clauses represent loop invariants.

```
 0. type INTEGER_ARRAY is array <> of INTEGER;
 1. procedure SORT_2 (A: in out INTEGER_ARRAY; N: in POSITIVE) is
 2. TEMP: INTEGER;
 3. M: POSITIVE;
 4.   --: entry(N ≥ 1)
 5.   --: exit (FORALL K(1 <= K and K <= N - 1 IMPLIES A(K) <= A(K + 1)))
 6. begin
 7.   --: assert (FORALL K(1 <= K and K <= I - 1 IMPLIES A(K) <= A(K + 1)))
 8.   for I in 1 ... N - 1 loop
 9.     M:=I;
10.     --: assert (FORALL T(I + 1 <= T and T <= J - 1 IMPLIES A(I) <= A(T))
11.     for J in I + 1 ... N loop
12.       if A(J) < A(I) then
13.         M:=J;
14.         TEMP:=A(I);
15.         A(I):=A(M);
16.         A(M):=TEMP;
17.       end if;
18.     end loop;
19.   end loop;
20. end SORT_2;
```

The verification is divided into several steps to deal with loops as discussed in [7], [6]. Each step verifies the correctness of a sub-path of the program. By denoting with En, Ex, A1 and A2 the entry, exit and the two assertion clauses, respectively, the following five steps must be performed:

Step 1 Assuming that En holds, A1 should hold before the execution of the extern loop.
Step 2 Assuming that A1 holds and the (extern) loop condition is true, A1 should hold after the execution of the loop body. The inner loop is not executed, instead its effect is represented by A2.

Step 3 Assuming that A1 holds and the (extern) loop condition is false, EX should hold.
Step 4 Assuming that A1 holds and the (extern) loop condition is true, A2 should hold.
Step 5 Assuming that A2 holds and the (inner) loop condition is true, A2 should hold after the execution of the (inner) loop body.

In what follows we discuss the verification step by step:

Step 1. The initial symbolic state is:

State: $\{\langle A,\{\langle \alpha.I,\text{true}\rangle\}\rangle,\langle N,\{\langle \eta,\text{true}\rangle\}\rangle,\langle \text{TEMP},\{\langle \text{undef.true}\rangle\}\rangle$
$\langle M,\{\langle \text{undef},\text{true}\rangle\}\rangle\}$
PC: $1 \leqslant \eta$

When evaluating the **for** loop (line 8), a new pair is added to State to represent variable I. Thus, the symbolic state at line 8 is*:

State: $\{\langle A,\{\langle \alpha.I,\text{true}\rangle\}\rangle,\langle N,\{\langle \eta,\text{true}\rangle\}\rangle,\langle \text{TEMP},\{\langle \text{undef},\text{true}\rangle\}\rangle,$
$\langle M,\{\langle \text{undef},\text{true}\rangle\}\rangle,\langle I,\{\langle 1,\text{true}\rangle\}\rangle\}$
PC: $1 \leqslant \eta$

PC should imply the assert statement at line 7:

$$1 \leqslant \eta \Rightarrow \forall K(1 \leqslant K \wedge K \leqslant 0 \Rightarrow \alpha.K \leqslant \alpha.K + 1)$$

The above implication can be rewritten as:

$$1 \leqslant \eta \Rightarrow \forall K(\text{false} \Rightarrow \alpha.K \leqslant \alpha.K + 1)$$

Thus, path 6–7–8 is correct.

Step 2. Under the assumption that the value of I is less or equal than $\eta - 1$ and greater or equal than 1, the loop is executed; moreover we assume that A1 holds. Thus, the loop body is executed starting from the following symbolic state:

State: $\{\langle A,\{\langle \alpha.I,\text{true}\rangle\}\rangle,\langle N,\{\langle \eta,\text{true}\rangle\}\rangle,\langle \text{TEMP},\{\langle \text{undef},\text{true}\rangle\}\rangle,$
$\langle M,\{\langle \text{undef},\text{true}\rangle\}\rangle,\langle I,\{\langle \iota,\text{true}\rangle\}\rangle\}$
PC: $1 \leqslant \iota \wedge \iota \leqslant \eta \wedge \forall K(1 \leqslant K \wedge K \leqslant \iota - 1 \Rightarrow \alpha.K \leqslant \alpha.K + 1)$

The assignment at line 9 is executed and then a new pair is added to State to represent J. At line 11 the symbolic state is:

State: $\{\langle A,\{\langle \alpha.I,\text{true}\rangle\}\rangle,\langle N,\{\langle \eta,\text{true}\rangle\}\rangle,\langle \text{TEMP},\{\langle \text{undef},\text{true}\rangle\}\rangle,\langle M,\{\langle \iota,\text{true}\rangle\}\rangle,$
$\langle I,\{\langle \iota,\text{true}\rangle\}\rangle,\langle J,\{\langle \iota + 1,\text{true}\rangle\}\rangle\}$
PC: $1 \leqslant \iota \wedge \iota \leqslant \eta \wedge \forall K(1 \leqslant K \wedge K \leqslant \iota - 1 \Rightarrow \alpha.K \leqslant \alpha.K + 1)$

We assume the inner loop is correct. Thus, we have the invariant of line 10 to represent its effect. The symbolic state at line 19 is:

State: $\{\langle A,\{\langle \alpha.I,\text{true}\rangle\}\rangle,\langle N,\{\langle \eta,\text{true}\rangle\}\rangle,\langle \text{TEMP},\{\langle \tau,\text{true}\rangle\}\rangle,\langle M,\{\langle \iota,\text{true}\rangle\}\rangle,$
$\langle I,\{\langle \iota + 1,\text{true}\rangle\}\rangle\}$
PC: $1 \leqslant \iota \wedge \iota \leqslant \eta \wedge \forall K(1 \leqslant K \wedge K \leqslant \iota - 1 \Rightarrow \alpha.K \leqslant \alpha.K + 1) \wedge \forall T(\iota + 1 \leqslant T \wedge T \leqslant \eta$
$\Rightarrow \alpha.\iota \leqslant \alpha T)$

To prove that the extern loop is correct PC should imply A1, i.e.

$$1 \leqslant \iota \wedge \iota \leqslant \eta \wedge \forall K(1 \leqslant K \wedge K \leqslant \iota - 1 \Rightarrow \alpha.K \leqslant \alpha.K + 1) \wedge \forall T(\iota + 1 \leqslant T \wedge T \leqslant \eta \Rightarrow \alpha.\iota \leqslant \alpha.T) \Rightarrow \forall K(1 \leqslant K \wedge K \leqslant \iota \Rightarrow \alpha.K \leqslant \alpha.K + 1)$$

*The statement **for** I **in** H . . K **loop** S **end loop** is viewed as:

```
I:=H;
while I <= K loop
  S;
  I:=I + 1;
end loop;
```

The above implication holds: the first invariant on the left hand side implies the right hand side for all values of K, but ι; hence, we can rewrite the implication as follows:

$$1 \leqslant \iota \cap \iota \leqslant \eta \wedge \forall T(\iota + 1 \leqslant T \wedge T \leqslant \eta \Rightarrow \alpha.\iota \leqslant \alpha.T) \Rightarrow \alpha.\iota \leqslant \alpha.\iota + 1$$

which is true. Thus, the extern loop is correct, if the inner one is.

Step 3. To prove program correctness we must prove that PC at the end of the execution implies the exit clause. The symbolic state at the end of the execution is:

State: $\{\langle A,\{\langle \alpha.I,\text{true}\rangle\}\rangle,\langle N,\{\langle \eta,\text{true}\rangle\}\rangle,\langle TEMP,\{\langle \tau,\text{true}\rangle\}\rangle,\langle M,\{\langle \mu,\text{true}\rangle\}\rangle,$
$\langle I,\{\langle \iota,\text{true}\rangle\}\rangle\}$
PC: $\iota > \eta \wedge \forall K(1 \leqslant K \wedge K \leqslant \iota - 1 \Rightarrow \alpha.K \leqslant \alpha.K + 1)$

while the exit clause is:

$$\forall K(1 \leqslant K \wedge K \leqslant \eta - 1 \Rightarrow \alpha.K \leqslant \alpha.K + 1)$$

Thus, the program is correct if the inner loop is.

Step 4. The verification of the inner loop starts from the symbolic state of line 11:

State: $\{\langle A,\{\langle \alpha.I,\text{true}\rangle\}\rangle,\langle N,\{\langle \eta,\text{true}\rangle\}\rangle,\langle TEMP,\{\langle \text{undef},\text{true}\rangle\}\rangle,\langle M,\{\langle \iota,\text{true}\rangle\}\rangle,$
$\langle I,\{\langle \iota,\text{true}\rangle\}\rangle,\langle J,\{\langle \iota + 1,\text{true}\rangle\}\rangle\}$
PC: $1 \leqslant \iota \wedge \iota \leqslant \eta \wedge \forall K(1 \leqslant K \wedge K \leqslant \iota - 1 \Rightarrow \alpha.K \leqslant \alpha.K + 1)$

PC trivially implies A2:

$$1 \leqslant \iota \wedge \iota \leqslant \eta \wedge \forall K(1 \leqslant K \wedge K \leqslant \iota - 1 \Rightarrow \alpha.K \leqslant \alpha.K + 1) \Rightarrow \forall T(\iota + 1 \leqslant T \wedge T \leqslant \iota \Rightarrow \alpha.\iota \leqslant \alpha.T)$$

Step 5. Assuming that J is less than or equal than η the inner loop is executed. Moreover A2 is assumed to hold. Thus, the symbolic state is:

State: $\{\langle A,\{\langle \alpha.I,\text{true}\rangle\}\rangle,\langle N,\{\langle \eta,\text{true}\rangle\}\rangle,\langle TEMP,\{\langle \text{undef},\text{true}\rangle\}\rangle,\langle M,\{\langle \iota,\text{true}\rangle\}\rangle,$
$\langle I,\{\langle \iota,\text{true}\rangle\}\rangle,\langle J,\{\langle \varphi,\text{true}\rangle\}\rangle\}$
PC: $1 \leqslant \iota \wedge \iota \leqslant \eta \wedge \iota + 1 \leqslant \varphi \wedge \varphi \leqslant \eta - 1 \wedge \forall T(\iota + 1 \leqslant T \wedge T \leqslant \varphi - 1 \Rightarrow \alpha.\iota \leqslant \alpha.T)$

The evaluation of the decision point (line 11) leads to the following implication:

$$PC \Rightarrow \alpha.\varphi < \alpha.\iota$$

The expression is neither *true* nor *false*. As a consequence, we should consider both cases:

(i) $\alpha.\varphi < \alpha.\iota$ is *false*.

No statement is executed and thus, the symbolic state at the end of the inner loop is:

State: $\{\langle A,\{\langle \alpha.I,\text{true}\rangle\}\rangle,\langle N,\{\langle \eta,\text{true}\rangle\}\rangle,\langle TEMP,\{\langle \text{undef},\text{true}\rangle\}\rangle,\langle M,\{\langle \iota,\text{true}\rangle\}\rangle,$
$\langle I,\{\langle \iota,\text{true}\rangle\}\rangle,\langle J,\{\langle \varphi + 1,\text{true}\rangle\}\rangle\}$
PC: $1 \leqslant \iota \wedge \iota \leqslant \eta \wedge \iota + 1 \leqslant \varphi \wedge \varphi \leqslant \eta - 1 \wedge \forall T(\iota + 1 \leqslant T \wedge T \leqslant \varphi - 1$
$\Rightarrow \alpha.\iota \leqslant \alpha.T) \wedge \alpha.\varphi \geqslant \alpha.\iota$

At the end of the inner loop we should prove that PC implies the assert statement of line 10:

$$1 \leqslant \iota \wedge \iota \leqslant \eta \wedge \iota + 1 \leqslant \varphi \wedge \varphi \leqslant \eta - 1 \wedge \forall T(\iota + 1 \leqslant T \wedge T \leqslant \varphi - 1 \Rightarrow \alpha.\iota \leqslant \alpha.T) \wedge \alpha.\varphi \geqslant \alpha.\iota \Rightarrow$$
$$\Rightarrow \forall T(\iota + 1 \leqslant T \wedge T \leqslant \varphi \Rightarrow \alpha.\iota \leqslant \alpha.T)$$

The implication can be rewritten as:

$$1 \leqslant \iota \wedge \iota \leqslant \eta \wedge \iota + 1 \leqslant \varphi \wedge \varphi \leqslant \eta - 1 \wedge \forall T(\iota + 1 \leqslant T \wedge T \leqslant \varphi - 1 \Rightarrow \alpha.\iota \leqslant \alpha.T) \wedge \alpha.\varphi \geqslant \alpha.\iota \Rightarrow$$
$$\Rightarrow \alpha.\iota \leqslant \alpha.\varphi$$

which is true.

(ii) $\alpha.\varphi < \alpha.\iota$ is *true*.

State: $\{\langle A,\{\langle \alpha.I,\text{true}\rangle\}\rangle,\langle N,\{\langle \eta,\text{true}\rangle\}\rangle,\langle TEMP,\{\langle \text{undef},\text{true}\rangle\}\rangle,\langle M,\{\langle \iota,\text{true}\rangle\}\rangle,$
$\langle I,\{\langle \iota,\text{true}\rangle\}\rangle,\langle J,\{\langle \varphi,\text{true}\rangle\}\rangle\}$

PC: $1 \leqslant \imath \wedge \imath \leqslant \eta \wedge \imath + 1 \leqslant \varphi \wedge \varphi \leqslant \eta - 1 \wedge \forall T(\imath + 1 \leqslant T \wedge T \leqslant \varphi - 1$
$\Rightarrow \alpha.\imath \leqslant \alpha.T) \wedge \alpha.\varphi < \alpha.\imath$

After the execution of line 13 the symbolic state is:

State: $\{\langle A,\{\langle \alpha.I,true\rangle\}\rangle,\langle N,\{\langle \eta,true\rangle\}\rangle,\langle TEMP,\{\langle undef,true\rangle\}\rangle,\langle M,\rangle\}\varphi,true\rangle\}\rangle,$
$\langle I,\{\langle \imath,true\rangle\}\rangle,\langle J,\{\langle \varphi,true\rangle\}\rangle\}$
PC: $1 \leqslant \imath \wedge \imath \leqslant \eta \wedge \imath + 1 \leqslant \varphi \wedge \varphi \leqslant \eta - 1 \wedge \forall T(\imath + 1 \leqslant T \wedge T \leqslant \varphi - 1$
$\Rightarrow \alpha.\imath \leqslant \alpha.T) \wedge \alpha.\varphi < \alpha.\imath$

After the execution of line 14 the symbolic state is:

State: $\{\langle A,P_{IA}\rangle,\langle N,\{\langle \eta,true\rangle\}\rangle,\langle TEMP,[P_{IA},P_{II}]\rangle,\langle M,\{\langle \varphi,true\rangle\}\rangle,\langle I,P_{II}\rangle,$
$\langle J,\{\langle \varphi,true\rangle\}\rangle\}$
PC: $1 \leqslant \imath \wedge \imath \leqslant \eta \wedge \imath + 1 \leqslant \varphi \wedge \varphi \leqslant \eta - 1 \wedge \forall T(\imath + 1 \leqslant T \wedge T \leqslant \varphi - 1$
$\Rightarrow \alpha.\imath \leqslant \alpha.T) \wedge \alpha.\varphi \langle \alpha.\imath$

Picture: $P_{IA} = \{\langle \alpha.I,true\rangle\}$
$P_{II} = \{\langle \imath,true\rangle\}$

After the execution of line 15 the symbolic state is:

State: $\{\langle A,[P_{IA}, P_{II},[P_{IA}, P_{IM}]]\rangle,\langle N,\{\langle \eta,true\rangle\}\rangle,\langle TEMP,[P_{IA}, P_{II}]\rangle,\langle M,P_{IM}\rangle,$
$\langle I,P_{II}\rangle,\langle J,\{\langle \varphi,true\rangle\}\rangle\}$
PC: $1 \leqslant \imath \wedge \imath \leqslant \eta \wedge \imath + 1 \leqslant \varphi \wedge \varphi \leqslant \eta - 1 \wedge \forall T(\imath + 1 \leqslant T \wedge T \leqslant \varphi - 1$
$\Rightarrow \alpha.\imath \leqslant \alpha.T) \wedge \alpha.\varphi \langle \alpha.\imath$

Picture: $P_{IA} = \{\langle \alpha.I,true\rangle\}$
$P_{II} = \{\langle \imath,true\rangle\}$
$P_{IM} = \{\langle \varphi,true\rangle\}$

Finally, after the execution of line 16 the symbolic state is:

State: $\{\langle A,[[P_{IA}, P_{II},[P_{IA}, P_{IM}]], P_{IM},P_{ITEMP}]\rangle,\langle N,\{\langle \eta,true\rangle\}\rangle,\langle TEMP,P_{ITEMP}\rangle,$
$\langle M,P_{IM}\rangle,\langle I,P_{II}\rangle,\langle J,\{\langle \varphi,true\rangle\}\rangle\}$
PC: $1 \leqslant \imath \wedge \imath \leqslant \eta \wedge \imath + 1 \leqslant \varphi \wedge \varphi \leqslant \eta - 1 \wedge \forall T(\imath + 1 \leqslant T \wedge T \leqslant \varphi - 1$
$\Rightarrow \alpha.\imath \leqslant \alpha.T) \wedge \alpha.\varphi < \alpha.\imath$

Picture: $P_{IA} = \{\langle \alpha.I,true\rangle\}$
$P_{II} = \{\langle \imath,true\rangle\}$
$P_{ITEMP} = [P_{IA},P_{II}]$
$P_{IM} = \{\langle \varphi,true\rangle\}$

The canonical form of the value-set of A is:

$$\{\langle \alpha.\varphi,I = \imath\rangle,\langle \alpha.\imath,I = \varphi\rangle,\langle \alpha.I, I \neq \imath \wedge I \neq \varphi\rangle\}$$

The inner loop is correct if PC implies A2 at the end of the loop body:

$$1 \leqslant \imath \wedge \imath \leqslant \eta \wedge \imath + 1 \leqslant \varphi \wedge \varphi \leqslant \eta - 1 \wedge \forall T(\imath + 1 \leqslant T \wedge T \leqslant \varphi - 1$$
$$\Rightarrow \alpha.\imath \leqslant \alpha.T) \wedge \alpha.\varphi < \alpha.\imath \Rightarrow \forall T(\imath + 1 \leqslant T \wedge T \leqslant \varphi \Rightarrow \alpha.\varphi \leqslant \alpha.T)$$

which can be rewritten as:

$$1 \leqslant \imath \wedge \imath \leqslant \eta \wedge \imath + 1 \leqslant \varphi \wedge \varphi \leqslant \eta - 1 \wedge \forall T(\imath + 1 \leqslant T \wedge T \leqslant \varphi - 1$$
$$\Rightarrow \alpha.\imath \leqslant \alpha.T) \wedge \alpha.\varphi < \alpha.\imath \Rightarrow \forall T(\imath + 1 \leqslant T \wedge T \leqslant \varphi - 1 \Rightarrow \alpha.\varphi \leqslant \alpha.T) \wedge \alpha.\varphi \leqslant \alpha.\imath$$

Thus, the inner loop is correct with respect to its invariant and, as is the program.

5. CONCLUSIONS

Traditional approaches to symbolic execution do not provide an effective way to deal with indexed variables and dynamic data structures. In this paper we introduced a formal definition of symbolic execution involving simple and indexed variables. Each variable is represented by a set describing every value it can be bound to and under which constraints such bindings actually occur. Furthermore, we have discussed how to represent incrementally such sets to avoid an excessive growth both in time and space of the computational resources needed to perform symbolic execution. Finally, we have sketched the algorithms used when developing SYMBAD and SESADA.

6. REFERENCES

1. Clarke, L. A. and Richardson, D. J. Symbolic evaluations methods—implementations and applications. In *Computer Program Testing*. Chandrasekaran, B. and Radicchi, S. (Eds), pp. 65–102. Amsterdam: North Holland; 1981.
2. King, J. C. Symbolic execution and program testing. *Commun. ACM* **19:** 385–394; 1976.
3. Clarke, L. A System to generate test data and symbolically execute programs. *Trans. Softw. Engng.* **2:** 215–222; 1976.
4. Korel, B. Automated software test data generation. *Trans. Softw. Engng.* **16:** 870–879; 1990.
5. Ramamoorty, C., Ho, S. and Chen, W. On the automated generation of program test data. *Trans. Softw. Engng.* **2:** 293–300; 1976.
6. Kemmerer, R. and Eckmann, S. UNISEX a UNIx—based Symbolic EXecutor for Pascal. *Softw. Pract. Exper.* **15:** 439–457; 1985.
7. Hanter, S. L. and King, J. C. An introduction to proving the correctness of programs. *Comp. Surv.* **8:** 331–353; 1976.
8. Coen-Porisini, A., De Paoli, F., Ghezzi, C. and Mandrioli, D. Software Specialization via Symbolic Execution. *Trans. Softw. Eng.* **17**; 1991.
9. Ghezzi, C., Mandrioli, D. and Tecchio, A. Program simplification via symbolic execution. *Lect. Notes Comput. Sci.* **206:** 116–128; 1985.
10. Howden, W. Symbolic Testing and the DISSECT Symbolic Evaluation system. *Trans. Softw. Engng.* **4:** 266–278; 1977.
11. Coen-Porisini, A. and De Paoli, F. Symbad: a symbolic executor of sequential Ada programs. *Proceedings of the International Conference on Safety, Security and Reliability Related Computer for the 1990's—SafeComp '90*, pp. 105–111. London (UK), 1990.
12. Coen-Porisini, A. and De Paoli, F. SESADA, an environment for software specialization. *Third European Software Engineering Conference-ESEC '91*, Milano (Italy) 1991.
13. Gordon, R. *The denotational description of programming languages*. New York: Springer Verlag; 1975.
14. Stoy, J. E. *Denotational Semantics: the Scott-Strachey approach to Programming Language Theory*. Cambridge, MA: MIT; 1977.
15. Aho, A. V., Hopcroft, J. E. and Ullman, J. D. *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley; 1986.
16. Floyd, R. W. Assigning meaning to programs. *Proc. Symp. Appl. Mathematica* **19:** 19–32; 1967.

About the Author—ALBERTO COEN-PORISINI was born in 1961. He received a Laura degree in Electrical Engineering from the Politecnico di Milano, Italy in 1987. From 1988 to 1991 he was a Ph.D. student in Computer Science at the Dipartimento di Elettronica of the Politecnico di Milano. He is currently a candidate for a Ph.D. degree. His research interests are in software engineering—specification languages, program transformation, reusability—and programming languages.

About the Author—FLAVIO DE PAOLI received a Laurea degree in Electrical Engineering in 1985, and a Ph.D. in Computer Science in 1991 from the Politecnico di Milano, Milano, Italy. His research interests include software engineering, software reuse, software metrics, programming languages, distributed systems and conferencing systems.