Assertion Graphs for Verifying and Synthesizing Programs*

by

Tilak Agerwala

Jayadev Misra

TR 83 August 78

* This work was supported in part by the National Science Foundation under NSF Grant DCR75-09842 and in part by the Joint Services Electronics Program under contract AFOSR F44620-76-0089.

THE UNIVERSITY OF TEXAS AT AUSTIN

Key Words:

Assertion graphs, inductive assertions, program synthesis, program verification, sequential programs.

Assertion Graphs for Verifying and Synthesizing Programs

1. Introduction

A notion of assertion graphs is introduced in this paper which has been found to be a useful tool for understanding and verifying programs and synthesizing programs from their specifications.

A program can be viewed as a transition system operating on a set of states. Each state corresponds to distinct values of variables, registers, location counters, etc. Execution of a program statement results in transition from one state to another. Clearly, such a view of program execution is not very useful since the number of states is usually very large and sometimes potentially infinite.

Instead, the set of states is partitioned into a finite number of subsets such that the states inside a subset have similar properties with respect to the given program. Then it is unnecessary to identify or study each state individually; only the subsets are of interest. This is more tractable since the number of subsets is finite. Each subset is characterized by an assertion involving the variables of the program. This finite set of assertions constitutes a finite state machine where transitions among assertions are effected by execution of program statements. Once a reasonable choice of subsets (assertions) has been made, the transitions can be derived from the program text quite easily using a simulation algorithm. These transitions can be used to verify a program. An example of verification of a program from an informal description of an algorithm is given in section 2.

Section 3 illustrates how these ideas could be applied to systematic synthesis in a hierarchic manner similar to that of Dijkstra [1]. The final section considers some other uses of the approach mainly as a tool for program understanding and synthesis of programs from informal descriptions of algorithms.

The ideas in this paper were motivated by the works of Floyd [2] and Hoare [4] on verification, Henderson [3] on program testing and Dijkstra [1] on constructive program synthesis.

2. The Finite Assertion Technique

It is possible to verify a program using assertion graphs. The technique consists of two distinct phases: model construction and finite simulation. A model of the system is constructed from an intuitive understanding of the algorithm. This model consists of a finite set of assertions; each assertion being a proposition over program variables and their initial values. The assertions are so chosen that they capture the relevant behavior of the program and at every point during program exeuction, one (or more) of these assertions are believed to be true. In terms of system states this is equivalent to obtaining, from the potentially infinite set of states, a finite set of subsets each representing a particular system behavior. The assertion is merely a representation of this behavior.

For instance, consider the problem of merging two sorted arrays A[1..m] and B[1..n] with the result being in C[1..m+n]. The algorithm maintains three pointers i, j, k which are indices to A, B, C respectively. Initially i=j=k=1. At each step A[i] and B[j] are compared and the smaller one becomes C[k]; proper indices are then updated. This continues until either A or B is exhausted, in which case the other array is copied into C.

This description of the problem enables us to construct a model:

Assertion 1: A1 : $1 \leq i \leq m$, $1 \leq j \leq n$, $A[i] \leq B[j]$.
Assertion 2: A2 : $1 \leq i \leq m$, $1 \leq j \leq n$, $A[i] > B[j]$.
Assertion 3: A3 : $1 \leq i \leq m$, $j > n$.
Assertion 4: A4 : $i > m$, $1 \leq j \leq n$.
Assertion 5: A5 : $i > m$, $j > n$.

Furthermore the following assertion is true in all states and hence is called a global assertion.

"k=i+j-1, C[1] to C[k-1] is sorted and consists of the elements A[1] to A[i-1] and B[1] to B[j-1]. A, B are sorted and are unchanged."

Clearly, the global assertion is violated during the time when i or j has been updated and k is yet to be updated. Considering such a set of operations as a single operation, it may then be postulated that at each instant during program execution, one or more of the given assertions are true along with the global assertion.

The next step is to verify a given program with respect to the postulated model. Verification is obtained through a method of "simulation." It is assumed that the semantics of each simple statement in the program is known in the sense that given an entry assertion and the statement, the resulting assertion(s) can be derived. In this sense, each statement moves the system from one assertion state to one or more assertion states. A simulation algorithm is presented later in the paper. This algorithm enables one to deduce systematically the transitions corresponsing to compound statements such as if-then-else, do-while, etc.. It then remains to be shown that starting from a set of initial states (implied by the input specification), the program always moves to some set of final states, where each final state satisfies the output specification. The method is informally presented below. Consider the following program for merging, with the input and output specifications shown within braces.

```
{i=1, j=1, k=1}
while i ≤ m  and  j ≤ n  do
        if   A[i] ≤ B[j]  then  C[k]:=A[i] ; k:=k+1 ; i:=i+1
                          else  C[k]:=B[j] ; k:=k+1 ; j:=j+1
        endif
enddo ;
while  i ≤ m   do  C[k]:=A[i] ; k:=k+1 ; i:=i+1  enddo ;
while  j ≤ n   do  C[k]:=B[j] ; k:=k+1 ; j:=j+1  enddo ;
{C[1..m+n] is sorted and consists of elements from A,B}.
```

Let S_1, S_2 denote the following groups of statements and B_1, B_2, B_3, and B_4 the following conditions.

S_1 : C[k]:=A[i] ; k:=k+1 ; i:=i+1

S_2 : C[k]:=B[j] ; k:=k+1 ; j:=j+1

B_1 : $i \le m$ and $j \le n$

B_2 : $A[i] \le B[j]$

B_3 : $i \le m$

B_4 : $j \le n$

Then the structure of the program is as shown below.

```
while $B_1$ do if $B_2$ then $S_1$ else $S_2$ endif enddo ;
while $B_3$ do $S_1$ ;
while $B_4$ do $S_2$ ;
```

The set of initial assertion states is $\{A_1, A_2\}$. B_2 is true in A1 and false in A2. Execution of S_1 in A1 leads to either A1 or A2 or A4; execution of S_2 in A2 leads to either A1 or A2 or A3. This is shown in Figure 1. Hence, it follows that starting from A1 or A2, execution of the statement "if B_2 then S_1 else S_2 endif" takes the program to A1, A2, A3, or A4. Since the statement is embedded inside a loop, it will possibly be executed many times. However, the loop exit condition holds for A3 and A4. Hence, it may be concluded that starting from A1 or A2, after any number of loop iterations, the program is in A1, A2, A3, or A4. If the loop ever terminates, the program must be in A3 or A4. As shown in Figure 2, "while B_3 do S_1" is applicable in A3 and "while B_4 do S_2" in A4. The simulation thus indicates that if the program terminates it is guaranteed to be in A5, which along with the global assertions implies the output condition. It remains to be shown that every loop terminates in order to prove total correctness.

The model and simulation algorithm (which derives the transitions from a given program text) are described below.

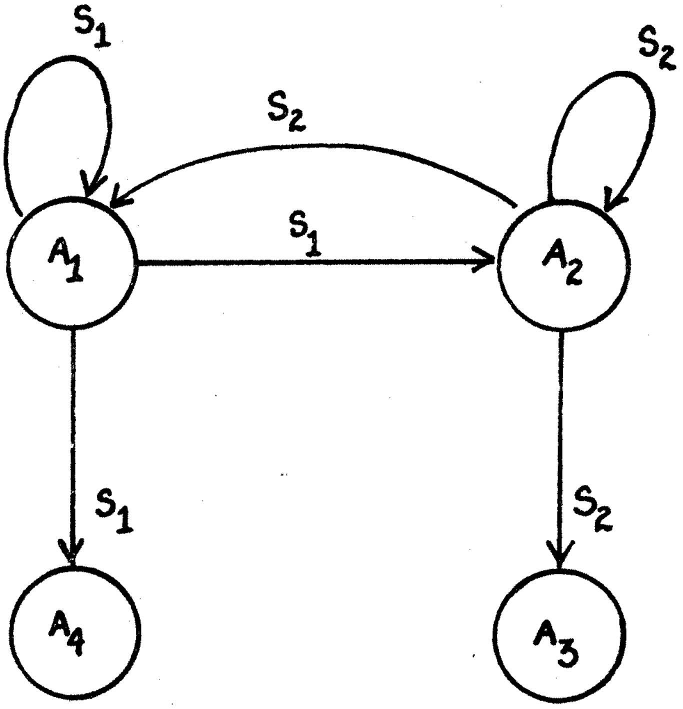


Figure 1

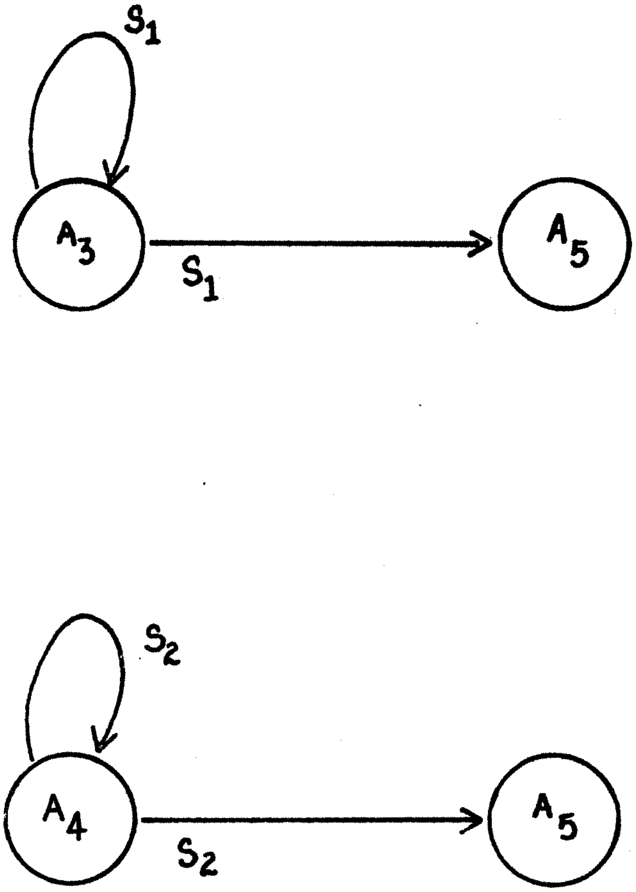


Figure 2

A model of the program is a finite set of assertions, M, on variable values. For any assertion $p \varepsilon M$ and statement S in the program, define S(p) to be a set of states, such that p{s}Q (using Hoare's notation [4]) and $Q = \bigvee_{q \varepsilon S(p)} q$. Thus, if any set of variable values satisfy p and S is executed then the resulting variable values always satisfy one (or more) q from S(p). In a transition diagram where each node represents a unique assertion in M, if $S(p) \subset M$ it would be represented as a set of arcs $A = \{<p,q> | q \varepsilon S(p)\}$. For a given M and $p \varepsilon M$, if there is no $S(p) \subset M$, then M is said to be incomplete with respect to S and p. This may happen if certain valid system states have been excluded from consideration in M. Furthermore, even if the model is complete there may be many sets $S(p) \subset M$, unless every pair of assertions belonging to M are disjoint (for all q_1, q_2, ε M there is no set of variable values satisfying $q_1 \wedge q_2$). We would normally be interested in a minimal S(p) (no subset of S(p) has the properties of S(p)). However, there might be many minimal sets when the states in the model are not disjoint. Any choice of S(p) will do if it leads to a proof using the simulation algorithm. We do not provide any insight into how a complete model is obtained or how S(p) is computed for simple statements. There is, of course, no algorithm to do this computation. However, all existing proof techniques have equivalent problems. Thus, the simulation algorithm works under the assumption that $S(p) \subset M$ can be computed given any simple statement S, a state p and a model M which is complete with respect to S and p.

The simulation algorithm given below computes S(p) for a compound statement S; in fact the algorithm computes a set of final assertions given a set of initial assertions.

If C is a set of assertions from a model M and B is some predicate define $\{C \wedge B\}$ to be a set Q such that $C \wedge B \Rightarrow \bigvee_{q \varepsilon Q} q$ and $Q \subset M$. If no such set

$\{C \wedge B\}$ exists, the model is incomplete with respect to C and B. Simulate (S,C,C*):

[The initial assertions are $C \subset M$; the program is S; the final assertions are C*.

$C^* = S(C) = \{q \mid p \varepsilon C \text{ and } q \varepsilon S(p)\}$.

The algorithm is applied recursively.]

```
Case S of
    S is a simple statement:  $C^* := \bigcup_{p \varepsilon C} S(p)$  [if $C = \emptyset$  then  $C^* = \emptyset$]
    $S = S_1; S_2$:          Simulate $(S_1, C, C_1^*)$ ;
                             Simulate $(S_2, C_1^*, C^*)$ ;
    S = if B then $S_1$ else $S_2$ endif:
                             Simulate $(S_1, \{C \wedge B\}, C_1^*)$ ;
                             Simulate $(S_2, \{C \wedge \neg B\}, C_2^*)$ ;
                             $C^* := C_1^* \cup C_2^*$
    S = while B do $S_1$ enddo:  [$C_i$ denotes the reachable states in i iterations
                                  or less]
                         $C_o := C$ ; $i := 0$,
                         repeat
                             Simulate $(S_1, \{C_i \wedge B\}, C')$ ;
                             $C_{i+1} = C_i \cup C'$ ; $i := i+1$
                         until $C_i = C_{i-1}$ ;
                         $C^* := \{C_i \wedge \neg B\}$ .
endcase
```

Theorem 1: The simulation algorithm terminates for any program S.

Proof: (outline)

1. By assumption, if S is simple S(p) can be computed. If S(p) is not a subset of M the model is incomplete and the algorithm fails.
2. Computation of $\{C \wedge B\}$ is finite: The computation is performed by enumerating subsets of M until one (say Q) is found such that $C \wedge B \Rightarrow \vee_{q \varepsilon Q}\, q$. If none is found, the model is incomplete and the algorithm fails.

3. (Simulate S, C, C*) where S is simple is clearly a finite computation by (1).

4. The loop corresponding to the "while-do-enddo" is executed a finite number of times: $|M|$ is finite. $|C_{i+1}| > |C_i|$ unless the loop terminates after the $(i+1)^{th}$ iteration. Hence the loop is executed at most $|M|$ times.

The correctness of the algorithm is straightforward and a formal proof of a program is obtained by:

(i) identifying a model;

(ii) applying the simulation algorithm to the set of initial assertions C, assuming that some process or human supplies S(p), for any simple statement S;

(iii) verifying that for every p in the resulting C^*, p implies the output condition.

If, during simulation the model is found to be incomplete, the whole process is repeated with an augmented model.

Proof of termination of the program involves displaying a function W from the program variables to a well founded set such that its value decreases with each transition [2].

It is easy to see that a proof by the proposed scheme can be transformed to one by inductive assertion and vice versa. The simulation algorithm returns C^* which can be written as a disjunction of the assertions in that subset. This corresponds to the output assertion of the program in an inductive assertion proof. In the case of a loop, while B do S_1 enddo, the simulation algorithm finds a certain C_i, such that $C_i = C_{i-1}$. C_i would correspond to the loop invariant since every further execution of S_1 would keep the system in the closed set of states C_i. Similarly the necessary assertions for the conditional statement can be found.

Conversely, the assertion used in a proof by the inductive assertion technique can be taken to constitute the model and the simulation algorithm applied to that model.

3. Program Synthesis

Given a problem with a finite number of states and certain well defined allowable transitions, it is straightforward to find a sequence of transitions leading from some initial state to a final state. Examples are puzzles such as the problem of missionaries and cannibals who want to cross a river or the problem of getting a certain amount of milk in a bottle, given certain specified amounts in other bottles.

In a general programming context the number of program states is usually infinite and the transitions are not well defined. Human resourcefulness is required in determining reasonable intermediate assertion states. Typically some input assertion I and output assertion 0 are given; the problem is to come up with a program S such that

$$I \xrightarrow{S} 0$$

If there is a statement in the underlying language or machine to effect this transition then the problem of synthesis is complete; otherwise one or more intermediate states and some transitions among them must be postulated. This process is repeated as long as some transition is not directly realizable. It may often be necessary to backtrack if an unwise choice was made previously. This notion of decomposition and refinement is due to Dijkstra [1] and Wirth [6].

This basic approach to program synthesis is utilized in conjunction with the finite assertion approach to illustrate the usefulness of the latter as an aid for improving understandability and as a tool for systematization.

The goal is to synthesize a correct program which inserts a key K into an ascending sorted list kept in linked form. If P is a pointer to a list element then P.INFO, P.LINK denote, respectively, the key and link (next node address) of a particular node. R points to the first node in the list. For simplicity, let us assume that the list contains at least one item. The list representation has at least two nodes, the first one being a dummy. The list (a) is shown in Figure 3.

As the overall strategy it is decided to use two pointers P and Q such that P.LINK=Q, to search the list sequentially in order to insert K at the appropriate point. The first step in the synthesis is shown in Figure 4. Starting with P, Q undefined, a program S takes the system into the desired final state. The system could be in the final state initially if the key is already present.

As the next step, an intermediate state A3 (P, Q defined) is postulated. An initialization step takes the system from A1 to A3. The list is searched in state 3. An insertion operation can take the system to the desired final state. This level of refinement is shown in Figure 5. The initialization is P:=R, Q:= R.LINK.

At this stage, refinement of state A3 is attempted. Two cases are easily identified:

(i) $K > Q.INFO$

(ii) $P.INFO < K < Q.INFO$

Figure 6 illustrates the refinement. An arc from A31 to A2 is postulated.

The refinement of A31 is now attempted leading to two substates:

(i) A31 and $Q.LINK = \lambda$

(ii) A31 and $Q.LINK \neq \lambda$

This is illustrated in Figure 7. At this stage a well defined transition diagram has been obtained. The meanings of "search", "insert", etc. are still intuitive. A program can now be written as follows:

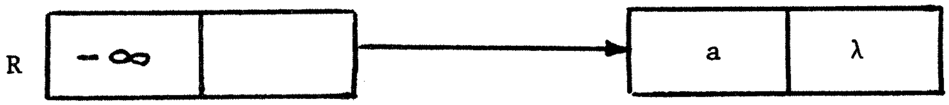
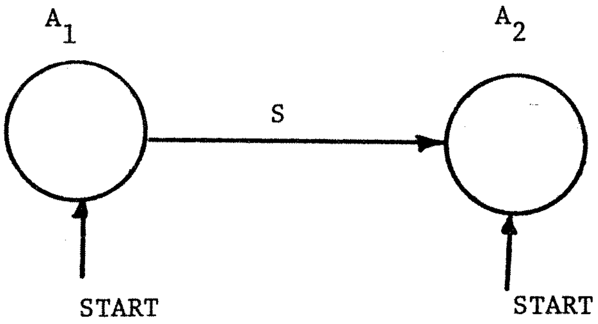


Figure 3

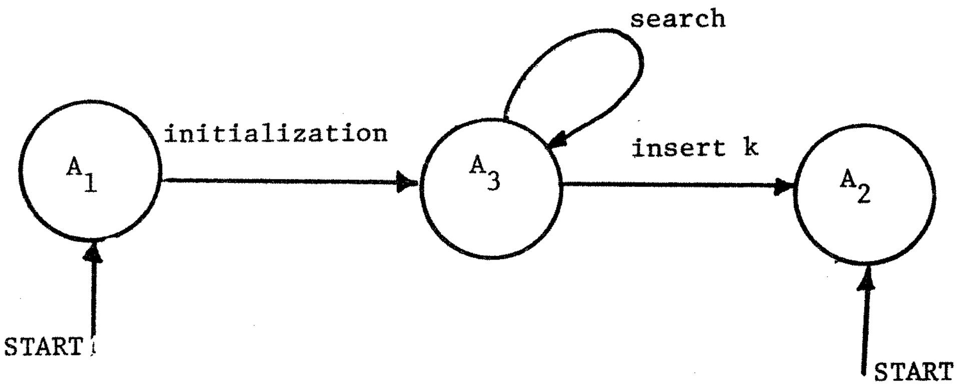


A_1 - P, Q undefined

A_2 - K is present in list in sorted order

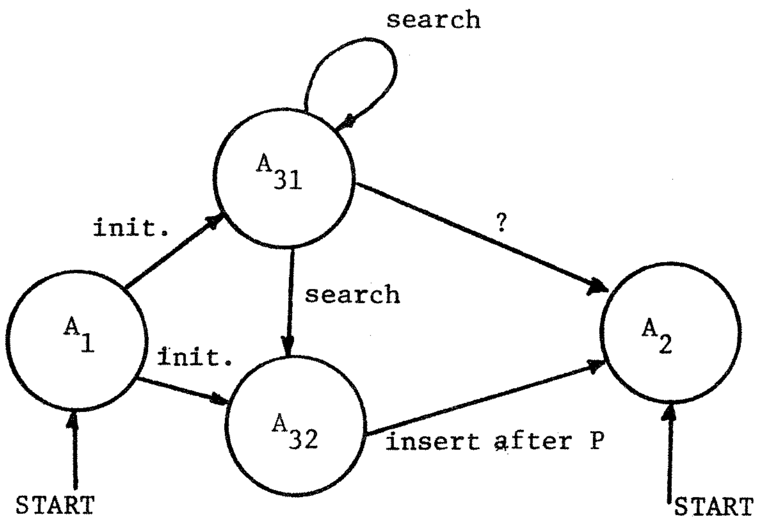
Global assertion: list contains orginal items in sorted order;
P, Q defined implies P . LINK = Q

Figure 4



A_3 - P, Q defined

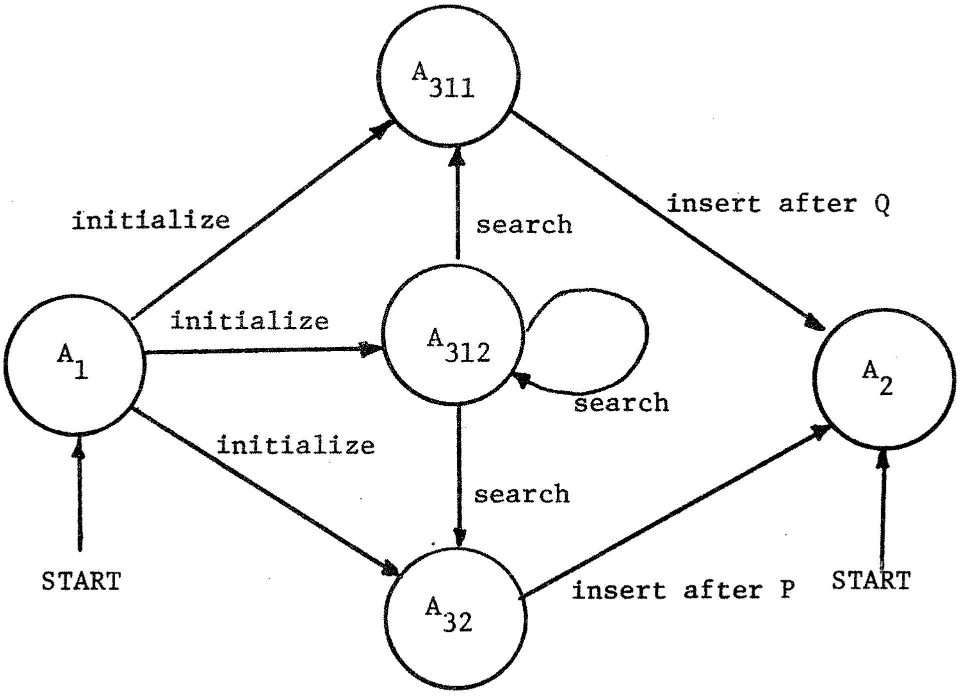
Figure 5



A_{31} - K > Q.INFO AND A3

A_{32} - P.INFO < K < Q.INFO AND A3

Figure 6



A_{311} - Q.LINK = λ AND A_{31}

A_{312} - Q.LINK $\neq$ λ AND A_{31}

Figure 7

```
BEGIN

    P=R; Q=R.LINK

    while  P, Q, defined, $K > Q.INFO$, $Q.LINK \neq \lambda$  do "search" enddo

    if     P, Q defined, $K > Q.INFO$, $Q.LINK = \lambda$ then "insert after Q" endif

    if     P, Q, defined, $P.INFO < K < Q.INFO$ then "insert after P"  endif

END
```

Refinement of the various operations can now be attempted.

```
Search

     P:=Q ; Q:=Q.LINK

Insert after Q

     obtain a node x from free list
     X.LINK:=Q.LINK
     Q.LINK:=x

     x.INFO:=K
```

Final program:

```
BEGIN
     P, Q:=R, R.LINK ;
     while  K > Q.INFO and Q.LINK $\neq \lambda$  do
            P:=Q ; Q:=Q.LINK
     enddo
     if     K > Q.INFO and Q.LINK $= \lambda$  then
            obtain x from free list
            X.LINK:=Q.LINK
            Q.LINK:=x
            x.INFO:=K
     endif
     if     P.INFO < K < Q.INFO  then
            obtain x from free list
            x.LINK:=Q
            x.INFO:=K
            P.LINK:=x
     endif
END
```

In fact, according to the analysis procedure presented earlier, this program has been proven to be partially correct.

4. Discussion

We have found that the notion of assertion graphs is extremely useful in understanding a program starting from an informal description of its underlying algorithm. Typically, one creates a mental model of program execution and identifies the program parts with the parts of the model. The assertion graph makes the model explicit; hence various program parts may either be simulated to derive the transitions or verified with respect to a given set of transitions. Algorithm understanding and program understanding are separated to a great extent. Once a reasonable set of assertions have been created, little intuition goes into applying the simulation algorithm. The assertions have a global scope and may be appreciated even in the absence of a program. Secondly, the transition graph (and the assertions) are fairly robust with respect to minor implementation changes, for instance in flow of control. The set of assertions remain intact as they usually depend on the global nature of the algorithm. Since the transitions can be obtained systematically, the major intuitive parts of the process do not have to be repeated.

The notion of viewing a program statement as a transformer of the state of a computation is due to Dijkstra. The synthesis approach presented here is not too different from that given in [1]. Using assertion graphs we have explicitly separated assertions about a program from statements in the program and this yields the advantages mentioned above.

To conclude this discussion, we point out that the concept of assertion graphs can be applied to parallel programs as well. The approach of Keller [7] illustrates this point. A global assertion over data variables (corresponding to a single assertion state in our scheme) is chosen and proved to be an invariant by induction on the number of statements executed. The technique could be extended by allowing multiple assertion states.

References

1. Dijkstra, E. W., A Discipline of Programming, Prentice Hall, 1976.

2. Floyd, R. W., Assigning Meaning to Programs, Proc. of Symp. on Applied Mathematics, Vol. 19, J. T. Schwartz (ed.), American Math. Society, Providence, R.I., 1967, 19-32.

3. Henderson, P., Finite State Modeling in Program Development, Proc. of the Intl. Conf. on Reliable Software, 21-23 April 1975, Los Angeles.

4. Hoare, C. A. R., An Axiomatic Basis of Computer Programming, CACM 12, 10 (Oct. 1969), 576-580, 583.

5. King, J. C., A New Approach to Program Testing, Proc. of the Intl Conf. on Reliable Software, 21-23 April 1975, Los Angeles.

6. Wirth, N., Program Development by Stepwise Refinement, CACM 14, 4 (April 1971), 221-227.

7. R. M. Keller, "Formal Verification of Parallel Programs," CACM Vol. 19, No. 7, July 1976, pp 371-384.