A synthesizer of inductive assertions

by STEVEN M. GERMAN and BEN WEGBREIT
Xerox Palo Alto Research Center
Palo Alto, California

INTRODUCTION

Mechanical verification of program correctness is desirable and possible.[6] Given a program with complete, correct predicate specifications* on the input, output, and each loop, verification of the output predicate is a mechanical process (cf. References 5 and 15 for surveys).

It is necessary and natural for a programmer to supply input and output assertions. However, completely specified inductive assertions on loops are redundant. Writing such redundant loop assertions is a tedious and error-prone task for the programmer, and is therefore an obstacle to the practical use of program verifiers. This paper describes a prototype system, Vista, which provides assistance in synthesizing correct inductive assertions. Given only the source program and input/output assertions, it is able to generate a useful class of assertions totally automatically. For a larger class, it is able to extend partial inductive assertions to form complete assertions, from which it proves program correctness.**

There has been a substantial amount of work on program verifiers (e.g., References 3, 8, 11, 14, 16, 19) and the synthesis of inductive assertions (e.g., References 2, 4, 9, 12, 13, 18, 21). Vista is one of the first implementations of techniques for assertion synthesis. This is an interim report on the current state of the system. In the course of the implementation, several new techniques were conceived and a number of previously published methods[21] were better understood. The second section of this paper presents a set of examples showing the range of Vista and the techniques it employs. The third section discusses its implementation and current limitations. Throughout, we place particular emphasis on general methods which may be of interest to researchers constructing similar systems.

Before turning to specifics, three general comments seem in order.

(1) Even in the current prototype state, the power of individual modules is somewhat impressive. For example, Vista can generate the inductive assertions for the first seven examples used by King[14] given only the input/output assertions, while it can take the incomplete inductive assertion supplied for Example 9 in Reference 14 and extend it to the complete assertion from which it proves program correctness. Similarly, Vista can generate about half of the inductive assertions needed to verify the median-finding program Find.[10]

(2) The power of Vista is due in no small measure to the theorem prover it employs. This is a component of the program verifier Pivot[3] which is the work of L. P. Deutsch. Vista is built on Pivot, employing Pivot's internal routines to carry out theorem proving, derive logical consequences, maintain data bases, and a variety of similar tasks. Salient points of this relationship are discussed in the section on implementation.

(3) Vista is not yet a complete, integrated, or fully automatic system. It presently consists of a number of loosely-coupled specialist modules each of which applies a particular technique of assertion synthesis and operates completely automatically once invoked. The specialist modules communicate by storing and retrieving information from a global data base associated with the program being verified. A control program invokes the specialist modules as required. However, this is the weakest portion of the current system: While the control program is currently able to automatically handle simple cases such as the first seven examples of Reference 14, more complex examples are done in interactive mode in which the user invokes the specialist modules. The shortcoming of the control program range from straightforward issues in programming to some ill-understood problems of heuristic control. The overall control structure, both present and planned, is discussed in a later section.

EXAMPLES

Vista uses four principal methods to obtain inductive assertions:

(1) symbolic evaluation in a weak interpretation,

* The predicate specification is correct if each assertion is a valid consequence of the input assertion and processing done by the program. The specification is complete if each assertion can be proved true given that all immediately preceding assertions are true.

** Throughout this paper and throughout the system we tacitly assume that the program does conform to its specifications and that formal verification of this conformation is the only issue at hand. The case of incorrect programs which must be debugged from the specifications, or the case of correct programs in which some of the internal inductive assertions are wrong, present additional problems. We would contend that these problems are best addressed after the simpler case of correct programs is better understood.

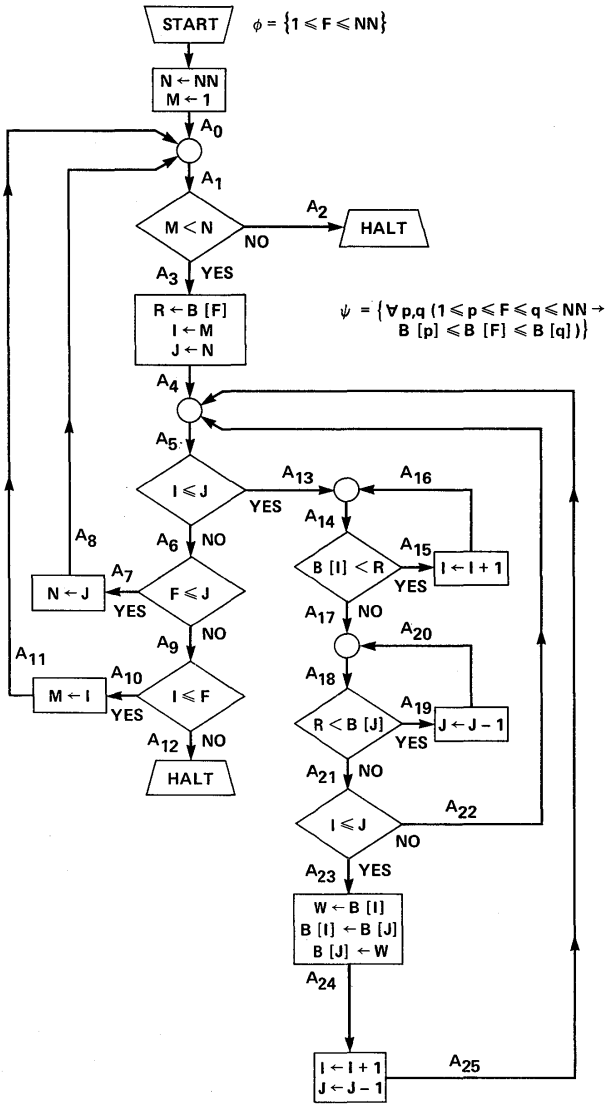


Figure 1—FIND

(2) combining output assertions with loop exit information to obtain trial loop assertions, and generalizing these where necessary,
(3) propagating valid assertions forward through the program, modifying them as required by the program transformations, producing assertions downstream,
(4) extracting information from proofs that fail in order to determine how assertions should be strengthened.

The first attempts to pick up obvious simple facts totally automatically. The others attempt to extend existing assertions—either to some other part of the program or by finding supporting lemmas needed to establish their correctness. We assume here that the programmer has supplied the essential ideas but may have omitted supporting details. The extent to which these details can be numerous, complex, and even subtle will be evident in the examples which follow.

We discuss these methods in turn, beginning each discussion with an explanation of the idea followed by an example to show how it is used in practice. Each example thus illustrates an invocation of one or more of Vista's specialist modules.

Notation. Throughout, a simple flowchart language is used. The input predicate is denoted by Φ; the output predicate by Ψ. Predicate P_i holds at arc A_i. To abbreviate a commonly used phrase, we say an assertion is *valid* if its verification conditions can be proved using assertions known to be correct.

Weak interpretation

In many programs, there is a useful amount of information that is obvious upon inspection. For example, suppose I and J are two integer variables. I is initialized to $J-1$ on entry to some loop, I is decremented on each path through the loop, J is unchanged, and the loop is exited when I turns negative. It follows that inside* the loop $0 \leq I < J$. As a second example, consider what would occur if the body of the loop were modified by the addition of a conditionally executed branch containing the assignment $J \leftarrow I \leftarrow J-I$; in this case, $0 \leq I \leq J$ would hold on the modified loop. As a third example, suppose the assignment $K \leftarrow B[I]$ was added to the body of the loop; in this case, it would follow that $K = B[I]$ and that assignments to $B[J+1]$ cannot affect this relation.

Weak interpretation attempts to derive simple facts of this sort; specifically, it considers only simple linear equalities or inequalities relating two variables. Its operation entails executing the computation steps symbolically, keeping track of the symbolic values of variables and of simple relations between them. When going repeatedly around a loop on which values are changed, it attempts to find a general expression which subsumes the particular cases. Thus, if an arc is reached with $I = N$ and then with $I = N-1$, the simple relation $I \leq N$ is tried. The outcome of weak interpretation is almost never enough to fully verify the program, but it does provide a set of valid relations which therefore need not be explicitly specified by the programmer.

For example, Figure 1 shows a program due to Hoare[10] for finding that element of an array $B[1:NN]$ whose value is $F-$th in order of magnitude. Weak interpretation results in the following valid assertions:

at arc A_1: $1 \leq M \leq F \leq N \leq NN$

at arc A_5: $M \leq I \ \& \ J \leq N \ \& \ 1 \leq M \leq F \leq N \leq NN$

at arc A_{21}: $B[J] \leq R \leq B[I] \ \& \ M \leq I \ \& \ J \leq N$
$\& \ 1 \leq M \leq F \leq N \leq NN$

at arc A_{24}: $B[I] \leq R \leq B[J] \ \& \ M \leq I \ \& \ J \leq N$
$\& \ 1 \leq M \leq F \leq N \leq NN$

* In this and in subsequent examples, we are deliberately informal, in order to present the main ideas as directly as possible. A precise statement would take into account the relative locations of the arc at which the assertion is formed and the points at which I is decremented.

Note the interdependence between the asignment $J \leftarrow N$ at A_3/A_4, the decrementing of J, the assertion $J \leq N$ at A_5, the test $F \leq J$, the assignment $N \leftarrow J$, and the assertion $F \leq N$ at A_1. Note also how $B[J] \leq R \leq B[I]$ becomes $B[I] \leq R \leq B[J]$ after the exchange.

Using loop exit tests and generalization

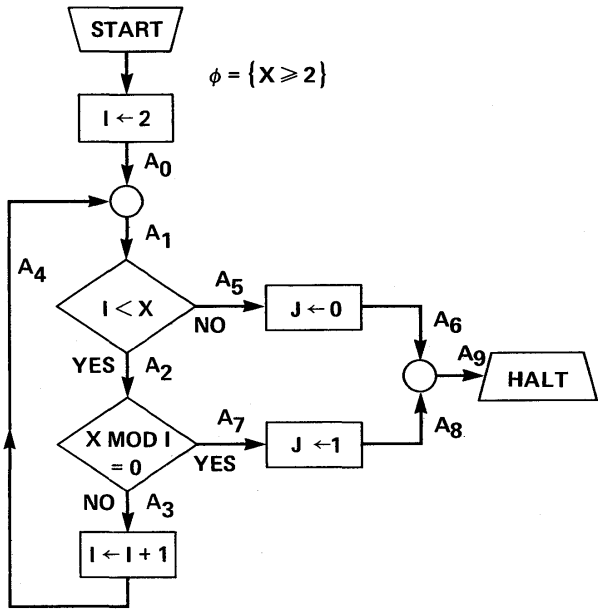
Suppose a loop is exited when some test D is true and that outside the loop some assertion P is to hold. Since P is to hold outside the loop, the assertion $\{D \rightarrow P\}$ must be true inside the loop, just before the exit test. $\{D \rightarrow P\}$ is the *weakest* inductive assertion at this point, in that any complete assertion must imply it. It may itself be a complete inductive assertion, in which case the theorem prover finds* it to be valid; or it may be incomplete, in which case it is found to be unprovable. If incomplete, it must be strengthened. Often, the forms of P and D suggest appropriate generalizations.** Suppose P asserts that some predicate Q is true of each j in the range $1 \leq j \leq N$ and D asserts that the loop is exited when $I \geq N$; one might hypothesize that inside the loop Q is true of each j in the sub-range from 1 to I, e.g., because the loop is counting up on I. This generalization, $\forall j (1 \leq j \leq I \rightarrow Q(j))$ is therefore tried as an inductive assertion.

As an example, consider the program in Figure 2 which tests whether X is prime. It sets the flag J to 0 if X is prime and to 1 otherwise. Vista can verify the program given only the input, Φ, and output, Ψ, assertions as shown. Combining Ψ with the loop exit information and simplifying, it obtains the weakest loop assertion: $\{I < X \vee \forall k (2 \leq k < X \rightarrow X \text{ MOD } k \neq 0)\}$. This is tested for validity, found to be unprovable, and then strengthened. The result, $\{\forall k (2 \leq k < I \rightarrow X \text{ MOD } k \neq 0\}$, is found to be a valid assertion. Further, it validates the output assertion, thus proving the program correct.

Predicate propagation

Whenever an assertion is known to be valid, it is useful to propagate it forward in the program, deriving the strongest consequences of the assertion downstream. Note that the verification condition[6] for a path deals only with the path's processing and the assertions at the head and end of the path, totally ignoring the rest of the program. Consequences of a valid predicate propagated downstream to a cutpoint may add new information which is absent in the inductive assertion at that cutpoint. On the path for which the cutpoint is the head, the new information may render valid a verification condition which would be invalid without the new information. Predicate propagation has three levels of sophistication.

Assertions which continue to hold, e.g., because they



$\psi = \{[J = 0 \rightarrow \forall k\ (2 \leq k < X \rightarrow X \text{ MOD } k \neq 0)] \wedge [J = 1 \rightarrow X \text{ MOD } I = 0 \wedge I < X]\}$

Figure 2—Testing for prime

involve variables unchanged on some path, are discovered. This is primarily a notational convenience: an assertion which holds over some large program region and is needed as a lemma in many loop verifications need be stated only once, at the head of the region.

Assertions are modified on passing through decisions and assignments to produce their consequences. On passing through a decision D in the Yes direction the clause D is added, and conversly for the No direction. On passing through an assignment $V \leftarrow E$, the clause $V = E$ is added and all uses of the old value of V are systematically eliminated. For example, if $B[J] \leq R$ is an assertion, then after passing through the test $R \leq S$, we have $\{B[J] \leq R \leq S\}$; after the assignment $R \leftarrow C$, we have $\{B[J] \leq S \ \& \ R = C\}$; further, after the assignment $J \leftarrow J - 1$, we have $\{B[J+1] \leq S \ \& \ R = C.\}$

When a junction is encountered, the assertion becomes a *trial* assertion for testing on the junction output arc. If it is valid there, a possibly useful new fact has been discovered. It may be unprovable, in which case a generally weaker assertion is formed by taking the disjunction of known assertions on all inputs to the junction and tested as a new trial assertion on the junction output arc.

A complete example of predicate propagation may be helpful. Figure 3 shows a simple sort program, Example 9 of Reference 14. Arc A_5 is tagged by hand with an incomplete assertion: that the portion of the array from $B[1]$ to $B[I]$ is sorted and that X is no larger than any element in the portion of B from $B[I]$ to $B[J-1]$. A complete assertion

* Throughout this paper, we assume that the theorem prover is able to confirm all valid theorems presented to it by Vista. The theorem prover is, of course, not complete over the integers with exponentiation—that is impossible by undecidability arguments. However, in practice, it turns out that the theorem prover can handle all the theorems we require.

** This is discussed in more detail in References 7 and 21.

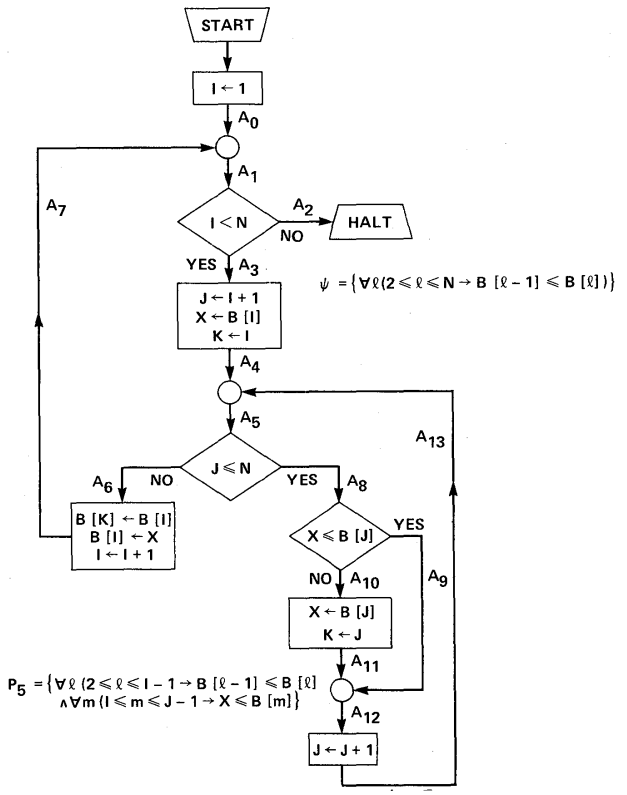


Figure 3—Sort by successively finding the smallest

would also include the key fact that no element in the sorted portion exceeds any element in the unsorted portion; lacking this, verification cannot proceed. Vista discovers the incompleteness when checking the inductive assertions for validity. The assertion that $B[1{:}I]$ is sorted is found to be initially true but not provable for subsequent passes around the loop $A_5A_6 \ldots A_4A_5$. However, the second assertion, $\forall m\,(I \leq m \leq J-1 \rightarrow \mathbf{X} \leq B[m])$ is found to be valid and is so recorded; Vista then propagates this valid assertion forward. The path which turns out to be interesting is $A_5A_6A_7A_1A_3A_4A_5$. Going along it produces the valid assertion at arc A_4: $\{I=1 \vee \forall m(I-1 \leq m \leq N \rightarrow B[I-1] \leq B[m])\}$ which may be read as: either this is the first time through the loop or $B[I-1]$ is the smallest element in the subarray between $B[I-1]$ and $B[N]$. This is the missing key fact. It is propagated forward to arc A_5 where it becomes a trial assertion, is checked for validity, and is found to be valid. Using weak interpretation as discussed previously, Vista also generates the additional assertions at A_5: $\{I \leq K < J \leq N+1$ & $I < N$ & $X = B[K]\}$. With these two sets of auxiliary assertions, the output predicate is then validated, thus proving the program correct.

Extracting information from unsuccessful proofs

When proving mathematical theorems, if some approach fails it is often useful to analyze the cause of failure and modify the approach to fix up the fault. This idea carries over to program verification. Suppose a loop* contains a trial assertion P—either specified by the programmer or generated as a trial assertion by one of the methods discussed previously—which cannot be verified around the loop. More precisely, let D be the set of conjuncts added because of decisions and let $F(\xi)$ be the transformation to the state vector ξ caused by loop assignments; loop verification requires that $\{\forall\xi(P(\xi)$ & $D(\xi) \rightarrow P(F(\xi))\}$ and this may be unprovable. Under the assumption that P is correct but incomplete, it follows that $P(F(\xi))$ must be true; hence, it is necessary to find an additional condition C such that $\{C(\xi)$ & $P(\xi)$ & $D(\xi)\}$ imply $\{C(F(\xi))$ & $P(F(\xi))\}$. Often it turns out that the verification condition fails only for certain identifiable cases and these cases may be used to construct the additional condition C. An example will best explain this.

Consider the program of Figure 4 which sorts an array B by straight insertion. At the start of the J-th pass, the

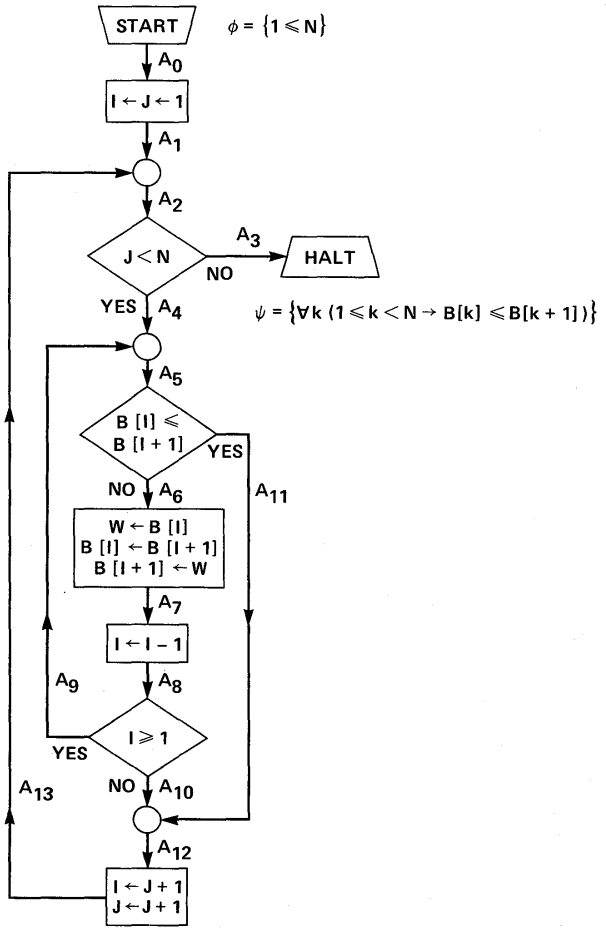


Figure 4—Straight insertion sort

* We discuss the application of assertion correction to a closed path; however, the idea can be used on any path.

subarray $B[1{:}J]$ has been sorted; the J-th pass inserts the element $B[J+1]$ in its correct position within the sorted subarray, thus sorting the subarray $B[1{:}J+1]$. Suppose Φ and Ψ are supplied as shown, along with the partial inductive assertion on arc A_5, that $P_5 = \{\forall k (1 \leq k \leq J \ \& \ k \neq I \rightarrow B[k] \leq B[k+1])\}$. Weak interpretation as discussed earlier adds $\{1 \leq I \leq J < N\}$. Vista attempts to prove P_5 is valid and finds that the proof fails for only one special case on the loop $A_5A_6A_7A_8A_9A_5$: if $B[I] > B[I+1]$ then after the exchange of $B[I]$ with $B[I+1]$ there is no way to show that $B[k] \leq B[k+1]$ when $k = I+1$. This would be true if and only if $B[I] \leq B[I+2]$ before the exchange and this has not yet been asserted. However, if P_5 is correct, this *must* be valid. Further, since $1 \leq k \leq J$ it follows that $I+1 = k \leq J$, i.e., $I < J$. This fact is included as part of the case information, resulting in a condition. Vista's analysis continues with path $A_5A_6A_7A_8A_{10}A_{12}A_{13}A_2$ on which the proof fails in two cases, yielding other conditions. The conjunction of the conditions simplifies to $\{I < J \ \& \ B[I] > B[I+1] \rightarrow B[I] \leq B[I+2]\}$. This may be read as: even if $B[I]$ is out of order, it is still smaller than $B[I+2]$. With this conjoined, P_5 is valid and validates Ψ, thus proving the program correct.

IMPLEMENTATION

In the interest of brevity, we confine our discussion to a few major points of the implementation. Further detail, treatment of other points and discussions of certain aspects of the overall organization can be found in References 7, 20 and 21. In particular, the reader will find there discussions of the theory and logical basis of the techniques presented below.

System structure

The major modules of Vista are (1) the weak interpreter, (2) trial generalization generator, (3) predicate propagator and (4) assertion correction and predicate initialization mechanism. Thus the example in the section *weak interpretation* represents a single call to the weak interpreter. The section *using loop exit tests* is handled by the control program, using the predicate initialization mechanism and the generalization generator. The section *predicate propagation* represents manual invocation of the weak interpreter followed by the predicate propagation mechanism. The section *extracting information from unsuccessful proofs* describes a manual invocation of the assertion corrector. It is indicative of the current status of Vista that when modules are invoked manually on the program in Figure 4. the complete assertions as both A_2 and A_5 are produced without any programmer-supplied inductive assertions, but that the control program is currently unable to duplicate this.

The current control program is applicable only to simple programs. We now describe a more general control strategy, which at present, would have to be executed interactively. The first step in assertion synthesis is to evaluate the entire program with the weak interpreter. This is a good start, because the weak interpreter produces assertions of guaranteed correctness that are useful during later phases of assertion synthesis and it does not itself require information other than the input assertion. Next, the strategy is to work on one cutpoint at a time, starting from those closest to the output assertion and working backwards. At each cutpoint, we attempt to convert the information available from weak interpretation and programmer-supplied assertions into a complete assertion. The predicate initialization mechanism is used to generate the weakest possible assertion. Then, as in the example involving loop exit tests, a number of possible generalizations of the weakest assertion are formed and tested for validity. Note that in multiple-loop programs, it is not possible to test for correctness until all cutpoints have complete assertions; at intermediate phases of assertion synthesis, loops without assertions are approximated by finite expansions of their bodies. If none of the trial assertions are correct and complete, the theorem prover is used to discard non-invariant clauses and form the strongest correct but incomplete assertion from the trial assertions. This predicate is propagated along closed paths back to the cutpoint. The result of predicate propagation is tested for new invariants and, if any are discovered, new generalizations are formed and tested. The discovery of a new invariant is a valuable gain in information. Therefore, we employ the new fact as quickly as possible. Otherwise the assertion correction mechanism is used to produce a new predicate to generalize and test. Synthesis proceeds in this fashion until a complete assertion is found.

Weak interpretation and predicate propagation

Weak interpretation and predicate propagation both attempt to derive assertions directly from the program structure, using similar processing steps. We discuss predicate propagation first and then explain how weak interpretation differs.

Predicate propagation starts with a known valid assertion P_0. Following the flowchart, processing at each node transforms a predicate P at the node input arc to obtain consequent predicate(s) at the output arc(s). Predicates are always expressed using current values of variables.

A decision node D conjoins to P the clauses D and $\sim D$ on its Yes and No output arcs respectively. It is frequently the case that the new clause could, if we wished, be combined with existing clauses of P to generate additional clauses. For example, after adding $S \leq T$ to $\{R \leq S \ \& \ T \leq U\}$ we could obtain the logically valid additional clauses $\{R \leq T \ \& \ R \leq U \ \& \ S \leq U\}$. The set of such additional clauses can, in general, be large—filled with redundant information. Hence, it is not a good practice to carry out the expansion. Instead, only simplification is performed. For example, if $P_1 = \{A \leq B \ \& \ C = D+1 \ \& \ F[B] \leq E\}$ is the predicate on the input arc, then the predicate on the Yes output arc of the decision $B = C$ is $P_2 = \{A \leq B \ \& \ C = D+1 \ \& \ F[B] \leq E \ \& \ B = C\}$.

Next, consider an assignment $X \leftarrow G(X, Y)$ where $R(X, Y)$ is the known predicate before the assignment. The *strongest*

predicate after the assignment is: $\{\exists X_0(R(X_0, Y) \& X = G(X_0, Y))\}$. However, this is too strong. Since it contains a new variable X_0 and since predicates are always expressed using current values of variables, this is unsuited to further propagation. Instead of using it directly, an attempt is made to find a logical consequent in which X_0 does not appear. This may be done in one of two ways depending on the form of the right hand side of the assignment:

(1) An invertible assignment is one which may be inverted to obtain the old value of the assigned variable as a function of its new value, i.e., $X_0 = H(X, Y)$ for some appropriate function H. Using this, X_0 may be directly eliminated from the strongest predicate, resulting in $\{R(H(X, Y), Y)\}$. For example, consider the above P_2 followed by the assignment $B \leftarrow B+2 \cdot E$. This may be inverted to obtain the old value of B as a function of its new value, i.e., $B_{\text{old}} = B_{\text{new}} - 2 \cdot E$. Substituting the right hand side, $B-2 \cdot E$, for the left, B, in P_2 expreses the relations using the new value of B. Thus, the predicate on the output arc of the assignment is $P_3 = \{A \leq B-2 \cdot E \ \& \ C = D+1 \ \& \ F[B-2 \cdot E] \leq E \ \& \ B-2 \cdot E = C\}$.

(2) A non-invertible assignment is more complex. Clauses which depend on the changed variable are deleted. However, the consequences of these clauses may not depend on the changed variable and these consequences may be invariants; hence, these consequences are derived and added to the data base before deletion. For example, if the above P_2 were followed by $B \leftarrow F[D]$, then the clauses $A \leq B$, $F[B] \leq E$, and $B = C$ would not be valid on the output arc; however, $A \leq C$, and $F[C] \leq E$ which are consequences *are* still valid, so these are generated and kept. As a final step, an equality expressing the new value of the changed variable is conjoined. For this example, the result is $P_4 = \{A \leq C \ \& \ F[C] \leq E \ \& \ C = D+1 \ \& \ B = F[D]\}$.

A junction node is the critical step in predicate propagation since it causes confluence of control and with it the need to find an assertion on the output that is valid for *each* of several paths leading to it. Consider a junction with n input arcs on which prior processing has resulted in the valid assertions $P_1, \ldots, P_n$. Let P_{n+1} be the valid assertion obtained on the output arc from prior processing (e.g., the starting predicate).

From $P_1, \ldots, P_{n+1}$, a new assertion on the output arc, P_{out}, is obtained. Each P_i is written as a set of clauses $P_i = C_i^1 \ \& \ C_i^2 \ \& \ \ldots \ \& \ C_i^{ki}$ and each conjunct C_i^j is tested on the implication $(P_1 \vee \ldots \vee P_{n+1} \rightarrow C_i^j)$. If this is provable, then C_i^j is a conjunct of P_{out}. Clauses which fail this simple test are tried as loop assertions and tested for validity on entry to and around the loop. Clauses which prove to be valid are added to P_{out}. Let P_i' be the residue after removing from P_i the clauses thus found to be in P_{out}. To propagate this residue, $(P_i' \vee \ldots \vee P_{n+1}')$ is formed and added as a conjunct of P_{out}. Finally, the new output assertion, P_{out}, is compared with the previous output assertion, P_{n+1}. Any clause in P_{out} not in P_{n+1} represents a new valid assertion and may be propagated forward.

Weak interpretation differs from predicate propagation in two respects. First, there is no known valid assertion to start with. We take as starting predicate P_0 on a loop the disjunction of the results of propagating the input assertion along all acyclic paths from the start node to the entrance of the loop. This is usually *not* a loop invariant; however, it can be used as a trial predicate and from it the weak interpreter attempts to construct a suitable generalization which *is* an invariant. Only simple linear relations between two variables are considered; hence, finding suitable generalizations by using domain-specific heuristics is often possible.* Secondly, the implementation differs in that weak interpretation requires only very simple deductions, rather than the full power of the theorem prover required for predicate propagation.

With the understanding that only those clauses expressing simple linear relations are carried, decision and assignment nodes are treated as in predicate propagation. When processing a junction, the first step is as in predicate propagation: Each conjunct C_i^j of each P_i is tested in $(P_1 \vee \ldots \vee P_{n+1} \rightarrow C_i^j)$ and successful conjuncts are included in the output predicate P_{out}. Generalization occurs when handling the conjuncts which fail this test. For example, suppose $n=2$, P_1 includes $I=J$, and P_2 includes $I=J+1$. The disjunction $\{I=J \vee I=J+1\}$ would be a logical consequence but probably not a loop invariant. Weak interpretation forms $\{I \leq J+1 \ \& \ I \geq J\}$ which is logically equivalent but is in a form more suggestive of generalization. This is in conjunctive form and is propagated around the loop again. If one of these conjuncts is an invariant then it will be implied by the result of further propagation and so it is kept, while the non-invariant conjunct is not thus implied and so is dropped. In this way, special cases comprising the starting trial predicate are progressively replaced with trial generalizations, some of which fail and are dropped while others are invariants and remain.

Constructing and correcting inductive assertions

Vista's abilities to construct inductive assertions from output predicates and to correct assertions from proofs that fail both depend heavily on a close working relationship between the trial assertion generator and Pivot, the theorem prover. In particular, Pivot's theorem proving process is structured to leave a record of the proof that is meaningful in relation to the program. At all major steps, Pivot records what actions it is taking and its reasons for taking them. Thus, when a proof fails, a body of useful information is produced for analysis by the trial assertion generator.

Vista's predicate *construction* mechanism forms the initial trial assertion inside a loop. Assume that a predicate P_0 must be shown to hold outside a loop, that P_I is known to be true inside (e.g., from weak interpretation), and that C is the computation in exiting the loop. Vista proceeds by asking Pivot to prove $(P_I \ \& \ C \rightarrow P_0)$. Usually, P_I will be insufficient

* In order to keep this discussion reasonably short, we are somewhat imprecise in several places. In particular, the method is only correct under suitable restrictions. See [20, 21] for a discussion of these restrictions and proofs of correctness.

for the proof to succeed; however, in the process of theorem proving, Pivot will find (implicitly) and simplify the additional facts that are needed inside the loop. For example, consider a loop with exit test $D=\{A[I]\leq Y\}$ for which the assertion $P_0=\{\exists v(Y\leq v\leq A[I])\ \&\ Q(v)\}$ must hold after exit. Assume the path that exits from the loop contains the assignment $A[J]\leftarrow X$. Then if it is known by weak interpretation that $\{I=J\}$ inside the loop, Vista will form the trial assertion $\{A[I]\leq Y\rightarrow[\exists v(Y\leq v\leq X)\ \&\ Q(v)]\}$ because $\{A[I]\leq Y\}$ is the exit condition and after the assignment $A[J]\leftarrow X$ the value of $A[I]$ in P_0 will be X.

When *correcting* assertions after proofs which fail, Vista's methods are similar but include special procedures for proofs involving simple arithmetic relations. One very useful method is to deduce a new invariant equality when the proof of a trial equality fails. For example, if Vista is considering the trial assertion $\{X=A^N\ \&\ N\geq 0\ \&\ M\geq 0\}$ on a loop with the assignments $X\leftarrow X\cdot B$, $N\leftarrow N+M$, then the clauses $N\geq 0$, $M\geq 0$ may be proved invariant, while $X=A^N$ will be found to be unprovable. Vista substitutes the values of variables after following the path around the loop into $X=A^N$, producing $X\cdot B=A^{N+M}$. It then uses a simple equation solver to find that this new equality would be satisfied if $A^N\cdot B=A^{N+M}$. If this new condition is satisfied upon entrance to the loop, it will be an invariant, and will allow $X=A^N$ to be proven invariant. Hence, it is conjoined to the trial assertion and a new invariant assertion is produced.

Vista's method for deducing a new invariant equality takes the original clause that could not be proven $(e=0)$, and the value of the clause at the ene of the path $(e'=0)$ and gives them to a simple equation solver which forms new equations by eliminating variables. If either equation allows direct solution for a variable, the value is substituted in the other equation yielding a new equality. A variable V which cannot be solved directly may sometimes be eliminated by finding a pair of arithmetic terms α, β such that all terms in $\alpha e+\beta e'=0$ which contain V are cancelled. All new equations formed by elimination of one or more variables are then tested for validity.

We now consider the example of an earlier section in more detail to show how Pivot is used to determine exactly why a trial assertion is insufficient. The path around the inner loop of Figure 4 switches $B[I]$ with $B[I+1]$ and decrements I. Consider the steps involved in trying to validate the partial inductive assertion $\{\forall k(1\leq k\leq J\ \&\ k\neq I\rightarrow B[k]\leq B[k+1])\}$, specifically in trying to show that if the assertion is true on one pass through the inner loop then it will still be true on the next pass. Pivot forms cases by considering the possible values for k. For values of k other than $I-1$, I, and $I+1$, $B[k]$ and $B[k+1]$ are unchanged. With $k=I-1$ the assertion will be vacuously satisfied on the next pass through the loop. When $k=I$ the goal is established directly. The only remaining value of k is $I+1$. Pivot decides to prove the path by forming two cases: $k\neq I+1$ and $k=I+1$. The proof of the first case succeeds. The second proof fails, leaving a record that when $B[I]>B[I+1]$ and $I\geq 2$ (so that the inner loop will be followed) and $1\leq k\leq J$ (so that k is the domain of the goal quantifier) and $k=I+1$ (for a case restriction) then the goal clause $B[I]\leq B[I+2]$ cannot be established.

Vista's assertion correcting mechanism examines Pivot's records and finds that the goal was created by instantiating the quantified trial assertion with $k=I+1$. Vista then forms a new universally quantified assertion in which the body is the original goal clause, in which k is the bound variable and in which the domain is the conjunction of the old domain and case restrictions on k. This quantifier is then simplified by Pivot's expression routines. Since the bound variable can be eliminated, the new quantifier reduces to $I<J\rightarrow B[I]\leq B[I+2]$. Finally, Vista retrieves the decision information $I\geq 2$ and $B[I]>B[I+1]$ and forms the additional condition $\{I\geq 2\ \&\ B[I]>B[I+1]\ \&\ I<J\rightarrow B[I]\leq B[I+2]\}$. Analysis of the other path that fails produces similar results applicable to that path. Together with the loop-path result, these simplify to produce the new assertion $\{I\geq 1\ \&\ I<J\ \&\ B[I]>B[I+1]\rightarrow B[I]\leq B[I+2]\}$ and allow the program to be proven correct.

CONCLUSION

Where do we go from here? Given the current state of Vista, what steps should be taken next and what prospects can be seen for production program verifiers in day-to-day use carrying out assertion synthesis?

There is currently substantial research, ongoing at various laboratories, investigating other methods for synthesizing inductive assertions. Particularly promising is the use of difference equations relating the values of program loop variables, whose solution may be used to obtain invariants.[4,12,13] We have implemented a package which finds inductive assertions in this way and are currently integrating it into Vista. Symbolic evaluation of the program with simplification and pattern matching has been studied[2,9,18] as another means for finding invariants. The system of Boyer and Moore[1] for proving theorems about pure Lisp functions tries to generalize theorems containing common subexpressions; in so doing, it uses methods related to the idea of strengthening partially specified assertions.

We suggest that an intermediate-to-expert level of competence is necessary if a verifier is to be of real use in verifying production programs. In particular, the assertions supplied by the programmer should be no more extensive than those used to explain the workings of a program to an experienced programmer. To achieve this level of competence, all the above methods are required. To a large extent, these methods are both non-overlapping and complementary. For example: difference equations are a well-understood means for obtaining equality invariants, but relatively useless for inequalities and disequalities (e.g., $a\neq b$); weak interpretation is well-suited for simple relations (both equalities and inequalities) but unable to produce bounded universally quantified assertions; examining the causes of failure in order to patch up an assertion which fails is very powerful, but only for trial assertions which are sufficiently close to the final invariant. Further, it is frequently the case that a relatively simple fact found by one method unblocks some other methods,

allowing a powerful line of attack to proceed. For these reasons, it seems most profitable to couple several good limited approaches of problem solution, rather than attempting to rely exclusively on one.

It must be emphasized that research in assertion synthesis is still in its infancy. Assertion synthesis at the level we believe desirable is still a distant goal. Programs of realistic size and complexity present a range of problems for which we have, as yet, no good solutions. Programs containing errors and the attendant problems of reconciling programs with their specifications offer further, and still harder, challenges.

Two related issues requiring further investigation are worth noting:

(1) The language for specifying assertions should be improved to facilitate specification of necessary assertions by the programmer (e.g., c.f. Reference 17). With the exception of Reference 1, there has been little work involving verification of programs containing assertions with programmer-defined recursive predicates. Also, it is difficult to express assertions about programs which manipulate list structure destructively. Studies in these area are being carried out and progress may be expected. It is then necessary to discover associated techniques for understanding, generalizing, and synthesizing assertions in these richer linguistic spaces.

(2) The theorem prover remains a fundamental module of any assertion synthesizing system. Improvements in domain-oriented theorem proving are therefore essential. A particular need which arises in assertion synthesis is the ability to efficiently check the validity of a formula and a number of slightly varied formulas (e.g., obtained by the deletion of conjuncts in the hypothesis of an implication). One would like a theorem prover to be able to simply extend, where possible, its proof of one formula when trying to prove a variant.

Substantial progress has been made in recent years toward the construction of production program verifiers; much remains to be done. The synthesis of inductive assertions is only one component, but an important one. Vista demonstrates that assertion synthesis is possible and can achieve significant performance.

ACKNOWLEDGMENTS

We wish to thank L. Peter Deutsch for his assistance in the use of his program verifier Pivot.

REFERENCES

1. Boyer, R. and J. Moore, "Proving Theorems about LISP functions," *Proc. 3rd Internat. Joint Conf. on Artificial Intelligence*, Aug. 1973, pp. 486-493.
2. Cooper, D. C., "Programs for Mechanical Program Verification," in *Machine Intelligence 6*, B. Meltzer and D. Michie (Eds.), American Elsevier, New York, 1971, pp. 43-59.
3. Deutsch, L. P., *An Interactive Program Verifier*, Ph.D. Thesis, Dept. of Computer Science, U. of California at Berkeley, 1973.
4. Elspas, B., *The Semiautomatic Generaton of Inductive Assertions for Proving Program Correctness*, SRI Project 2686, Stanford Research Institute, July 1974.
5. Elspas, B., K. L. Levitt, R. J. Waldinger, and A. Waksman, "An Assessment of Techniques for Proving Programs Correct," *Computing Surveys*, 4, 2, June 1972, pp. 97-147.
6. Floyd, R., "Assigning Meanings to Programs," in *Proc. of a Symposium in Applied Mathematics*, Vol. 19, J. T. Schwartz (Ed.) AMS, 1967, pp. 19-32.
7. German, S. M., *A Program Verifier that Generates Inductive Assertions*, Technical Report TR 19-74, Center for Research in Computing Technology, Harvard U., Aug. 1974.
8. Good, D. I., "Provable Programs and Processors," *AFIPS Conference Proceedings*, Vol. 43, *1974 National Computer Conf.*, pp. 357-363.
9. Greif, I. and R. Waldinger, "A More Mechanical Heuristic Approach to Program Verification," in *Internat. Symp. on Programming*, Paris, April, 1974, pp. 83-90.
10. Hoare, C. A. R., Proof of a Program: FIND," *C. ACM*, 14, 1, Jan. 1971, pp. 39-45.
11. Igarashi, S., R. L. London, and D. C. Luckham, *Automatic Program Verification I: AIM-200*, CS-73-365, Computer Science Dept., Stanford U., May 1973.
12. Katz, S. M. and Z. Manna, "A Heuristic Approach to Program Verification," *Proc. 3rd Internat. Joint Conf. on Artificial Intell.*, Aug. 1973, pp. 500-512.
13. Katz, S. and Z. Manna, *Logical Analysis of Programs*, Dept. of Applied Mathematics, Weizmann Inst. of Science, Rehovot, Israel, July 1974.
14. King, J., *A Program Verifier*, Ph.D. Thesis, Computer Science Dept., Carnegie-Mellon U., 1969.
15. London, R. L., "The Current State of Proving Programs Correct," *Proc. ACM 25th Ann. Conf.*, 1972, pp. 39-46.
16. London, R. L. and D. R. Musser, "The Application of a Symbolic Mathematical System to Program Verification," to appear in *ACM 74*.
17. Marmier, E., "A program verifier for PASCAL," *Proc. IFIP Congress 74*, North-Holland Publishing Co., pp. 177-181.
18. Moriconi, M., *Semiautomatic Synthesis of Inductive Predicates*, ATP-16, Depts. of Mathematics and Computer Sciences, U. of Texas at Austin, June 1974.
19. Waldinger, R. J. and K. N. Levitt, *Reasoning About Programs*, Artificial Intell. Center, Technical Note 86, Stanford Research Inst., Oct. 1973.
20. Wegbreit, B., *Property Extraction in Well-Founded Property Sets*, Center for Research in Computing Technology, Harvard U., Feb. 1973.
21. Wegbreit, B., "The Synthesis of Loop Predicates," *C. ACM*, 17, 2 (Feb. 1974), pp. 102-112.