November 1970

ON PROGRAM SYNTHESIS AND PROGRAM VERIFICATION

by

Zohar Manna
Computer Science Department
Stanford University

Richard J. Waldinger
Artificial Intelligence Group
Stanford Research Institute

Paper presented at the 4th Hawaii International Conference on Systems Science, Honolulu, Hawaii, January 12-14, 1971.

Artificial Intelligence Group

Technical Note 52

SRI Projects 8721 and 8550

The research reported herein was sponsored in part by the Air Force Systems Command, USAF, Department of Defense, through the Air Force Cambridge Research Laboratories, Office of Aerospace Research, under Contract F19628-70-C-0246, and by the National Aeronautics and Space Administration under Contract NASW-2086.

ON PROGRAM SYNTHESIS AND PROGRAM VERIFICATION

Zohar Manna
Computer Science Dept.
Stanford University

and

Richard J. Waldinger
Artificial Intelligence Group
Stanford Research Institute

Abstract

Certain similarities between program verification and program synthesis are pointed out. The analogy is illustrated using a "bubble-sort" program.

Recent work has shown that automatic deductive methods may be applied to the problems of program verification [1] and program synthesis [2]. As it turns out, these techniques are closely related. We demonstrate this relation using a particular program.

VERIFICATION

Consider the following program for "bubble-sorting" an array a of n+1 real numbers a[0],...,a[n]. (Ignore for a moment the attached assertions.) The operation a ← exchange(a,i,j) has the effect of exchanging the contents of a[i] and a[j].

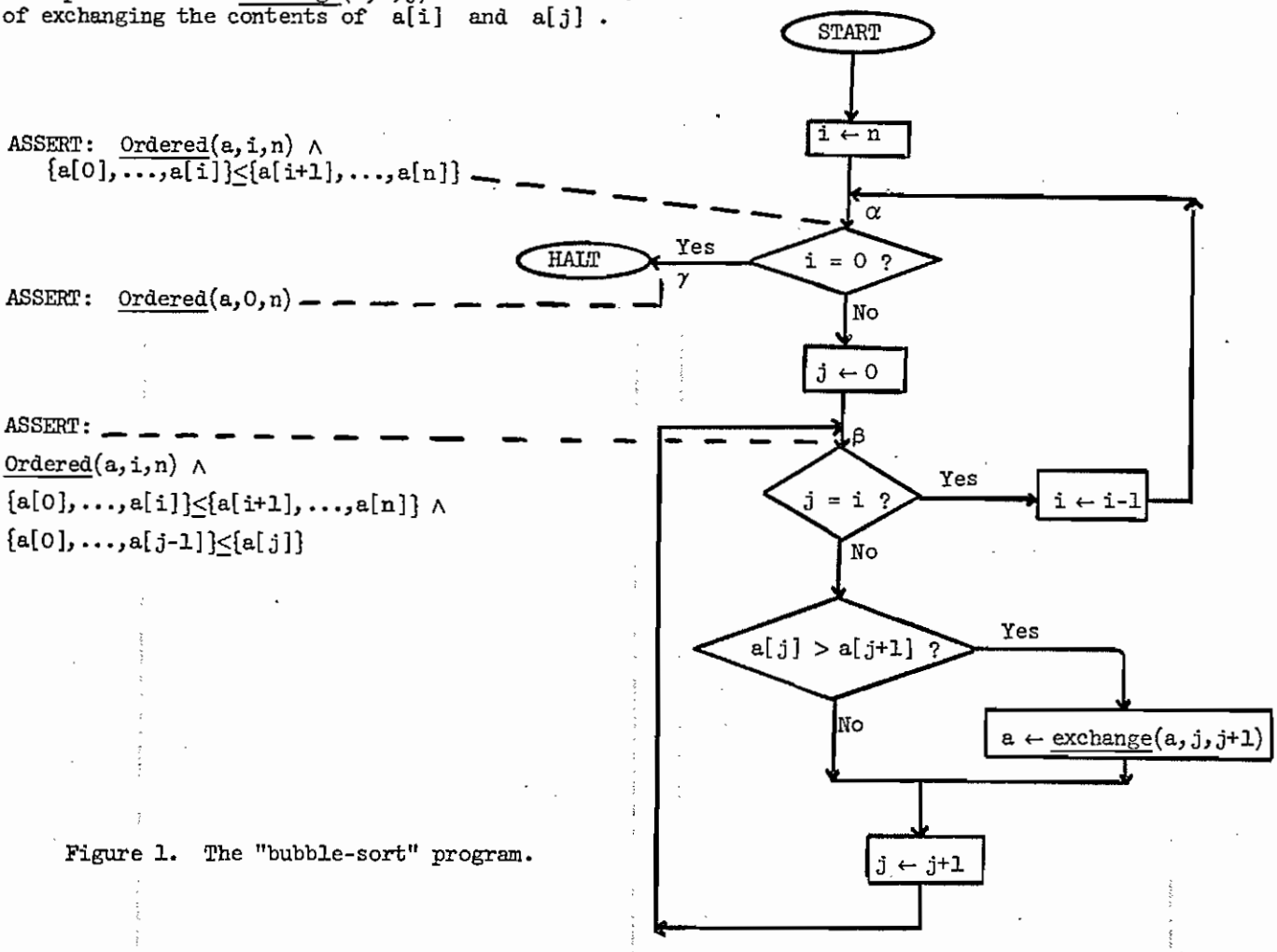

Figure 1. The "bubble-sort" program.

We wish to prove that this program is correct, and that it always terminates. To say that this program is correct is to say that when it halts, (i) the elements of the array a are the same as those of the initial array, but (ii) that they are in increasing order. It is clear that (i) holds, since $\underline{\text{exchange}}(a,i,j)$ is the only operation applied to a, and exchange leaves the contents of the array a unchanged except for order. Therefore we shall concentrate on the establishment of (ii). Later we shall show that the program terminates.

Following Floyd [1], we will attach the assertion $\underline{\text{Ordered}}(a,0,n)$ to the exit γ of the program.

The predicate $\underline{\text{Ordered}}(a,k,\ell)$ is taken to mean that the elements $a[k],a[k+1],\ldots,a[\ell]$ are in increasing order. (This is considered to be vacuously true if $k \geq \ell$.) Floyd's method requires that we affix assertions to certain intermediate points in the program, at least one point within each loop. These assertions describe the situation when control passes through those points. For example, to point α we attach the assertion

$$\underline{\text{Ordered}}(a,i,n) \wedge \{a[0],\ldots,a[i]\} \leq \{a[i+1],\ldots,a[n]\} .$$

The expression $\{a[k],\ldots,a[\ell]\}$ represents the set $\{a[m] \mid k \leq m \leq \ell\}$ (note that this set is empty if $k > \ell$). Furthermore, for any two sets of real numbers S and T, $S \leq T$ means that every element of S is less than or equal to any element of T (which is vacuously true if either S or T is empty).

In order to demonstrate the correctness of the program, we have simply to prove that when control passes through one of the labeled points, the values currently assigned to the variables satisfy the corresponding assertion, assuming that the assertion corresponding to the previous point was satisfied. This implies that if control reaches the exit the corresponding assertion will be satisfied, establishing the correctness of the program.

We have not yet discussed the termination of the program. We do this using the notion of the "well-ordered" set [1]. For this program we consider the set of pairs of non-negative integers well-ordered (lexicographically) as follows

$$(i_1,j_1) \prec (i_2,j_2)$$

if and only if

$$\text{either} \quad i_1 \leq i_2$$

$$\text{or} \quad i_1 = i_2 \quad \text{and} \quad j_1 \leq j_2 .$$

There are no infinite sequences of pairs of non-negative integers that are strictly decreasing under the above order. In our bubble-sort program, the quantities i and $i-j$ are non-negative whenever control passes through point β. Furthermore, consider the sequence of pairs of non-negative integers constructed as follows: whenever control passes through point β the current value of $(i,i-j)$ is added to the sequence. Then it can be shown that this sequence is strictly decreasing under the lexicographic order. Since this sequence must be finite, control can only pass through β finitely often; hence the program must terminate.

SYNTHESIS

To provide a basis for comparison, let us illustrate a synthesis process to construct a bubble-sort program automatically. We are given the input-output relation denoted by

$$a^* \approx a \wedge \underline{\text{Ordered}}(a^*,0,n) ,$$

where a is the input vector, a^* is the output vector, and $a^* \approx a$ means that a and a^* are the same vectors up to reordering. In general, if we wish to construct a program satisfying an input-output relation $R(x,y)$, with input x and output y, we can ask the synthesis system to find a constructive proof of the theorem

$$(\forall x)(\exists y)R(x,y) .$$

It then extracts a program that satisfies the above relation, and is thereby guaranteed to terminate and be correct.

In this case, the theorem to be proved is

$$(\forall n)(\forall a)(\exists a^*)[a^* \approx a \wedge \underline{\text{Ordered}}(a^*,0,n)] .$$

If the system is given this information alone, it will produce a sort program, but we have no way of controlling the sorting method it will use. Therefore, in order to direct the synthesis procedure to yield a bubble-sort program, and also to facilitate the search for a proof, we give the theorem-prover some additional information: it should use "going-down" induction [2] with the hypothesis

$$(\exists a^*)[\{a^*[0],\ldots,a^*[i]\} \leq \{a^*[i+1],\ldots,a^*[n]\} \wedge a^* \approx a \wedge \underline{\text{Ordered}}(a^*,i,n)] .$$

Then the theorem-prover proves two lemmas

I. (Initial step)

$$(\exists i)(\exists a^*)[\{a^*[0],\ldots,a^*[i]\} \leq \{a^*[i+1],\ldots,a^*[n]\} \wedge a^* \approx a \wedge \underline{\text{Ordered}}(a^*,i,n)] .$$

II. (Inductive step)

$$(\forall i)[i \neq 0 \wedge (\exists a^*)[\{a^*[0],\ldots,a^*[i]\} \leq \{a^*[i+1],\ldots,a^*[n]\} \wedge a^* \approx a \wedge \underline{\text{Ordered}}(a^*,i,n)]] \supset (\exists a^*)[\{a^*[0],\ldots,a^*[i-1]\} \leq \{a^*[i],\ldots,a^*[n]\} \wedge a^* \approx a \wedge \underline{\text{Ordered}}(a^*,i-1,n)] .$$

If the system succeeds in proving both these lemmas, it can conclude

$$(\exists a^*)[\{a^*[0]\} \leq \{a^*[1],\ldots,a^*[n]\} \wedge a^* \approx a \wedge \underline{\text{Ordered}}(a^*,0,n)]$$

which implies the desired result.

The proof of Lemma I is trivial, taking i to be n and a^* to be a.

In order to prove Lemma II, the system finds it suffices to show

$$\{a^*[0],\ldots,a^*[i-1]\} \leq \{a^*[i]\}$$

for any a^* satisfying the antecedent of the implication. Failing to establish this directly, it applies induction again; this time it uses "going-up" induction with the hypothesis

$$(\exists a^*)[\{a^*[0],\ldots,a^*[i-1]\} \leq \{a^*[i],\ldots,a^*[n]\}$$
$$\wedge\ a^* \approx a \wedge \underline{\text{Ordered}}(a^*,i-1,n)$$
$$\wedge\ \{a^*[0],\ldots,a^*[j-1]\} \leq \{a^*[j]\}]$$

where j, $j \leq i$, is the induction variable.

In applying the principle we derive two more lemmas to be proved: the proof of the first is trivial, but the proof of the second requires case analysis and gives rise to the program segment illustrated in Figure 2.

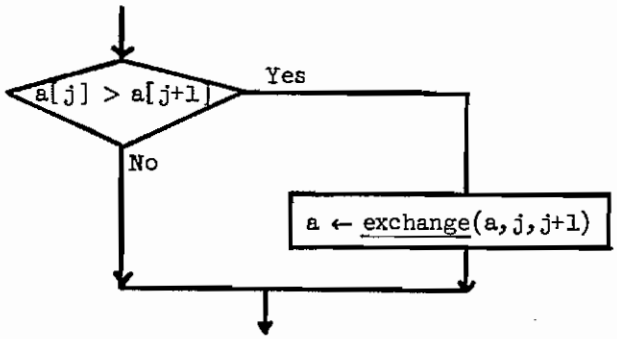

Figure 2. A segment of the bubble sort program

Then from the induction principles used, and the above segment, the synthesizer will construct the program illustrated in Figure 1.

CONCLUSION

The parallel between the analysis and synthesis methods is striking. The well-ordering used in proving termination corresponds precisely to the induction principles used in the synthesis proof. Furthermore, the two assertions associated with arcs α and β respectively in the correctness proof are essentially the same as the two induction hypotheses used in the synthesis proof. In fact, if the proofs are examined in detail, one finds that the same axioms and rules of inference are used in each proof. However, the synthesis proof requires much more ingenuity from the theorem-prover, as is to be expected.

References

[1] R. W. Floyd, "Assigning Meaning to Programs," in Proceedings of Symposia in Applied Mathematics, American Mathematical Society, Vol. 19, pp. 19-32.

[2] Z. Manna and R. J. Waldinger, "Towards Automatic Programming Synthesis," C.ACM. To appear.

Acknowledgment

The research reported herein was sponsored in part by the Air Force Systems Command, USAF, Department of Defense through the Air Force Cambridge Research Laboratories, Office of Aerospace Research, under Contract No. F19628-70-C-0246, and by the Advanced Research Projects Agency of the Department of Defense and the National Aeronautics and Space Administration under Contract No. NASW-2086 (at Stanford Research Institute).