Stanford Artificial Intelligence Laboratory
Memo AIM-281

June 1976

Computer Science Department
Report No. STAN-CS-76-558

Is "sometime" sometimes better than "always"? Intermittent assertions in proving program correctness

by

Zohar Manna
Artificial Intelligence Lab
Stanford University
Stanford, Ca.

Richard Waldinger
Artificial Intelligence Center
Stanford Research Institute
Menlo Park, Ca.

Research sponsored by

Advanced Research Projects Agency
ARPA Order No. 2494
and
National Science Foundation

COMPUTER SCIENCE DEPARTMENT
Stanford University

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: STAN-CS-76-558, AIM-281
2. GOVT ACCESSION NO.:
3. RECIPIENT'S CATALOG NUMBER:
4. TITLE (and Subtitle): IS "SOMETIME" SOMETIMES BETTER THAN "ALWAYS"? Intermittent assertions in proving program correctness.
5. TYPE OF REPORT & PERIOD COVERED: Technical rept.
6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): Zohar Manna [redacted] Richard Waldinger
8. CONTRACT OR GRANT NUMBER(s): DAHC15-73-C-0435, NSF-GJ-36146
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Artificial Intelligence Laboratory, Stanford University, Stanford, California 94305
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: ARPA Order-2494
11. CONTROLLING OFFICE NAME AND ADDRESS: Col. Dave Russell, Dep. Dir., ARPA, IPT, ARPA Headquarters, 1400 Wilson Blvd. Arlington, Virginia 22209
12. REPORT DATE: 16 June 1976
13. NUMBER OF PAGES: 40
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Philip Surra, ONR Representative, Durand Aeronautics Building Room 165, Stanford University, Stanford, California 94305 (12) 43p.
15. SECURITY CLASS. (of this report): 15
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:
16. DISTRIBUTION STATEMENT (of this Report): Releasable without limitations on dissemination.

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report):
18. SUPPLEMENTARY NOTES:
19. KEY WORDS (Continue on reverse side if necessary and identify by block number):
20. ABSTRACT (Continue on reverse side if necessary and identify by block number):

This paper explores a technique for proving the correctness and termination of programs simultaneously. This approach, which we call the *intermittent-assertion method*, involves documenting the program with assertions that must be true at some time when control is passing through the corresponding point, but that need not be true every time. The method, introduced by Knuth and further developed by Burstall, promises to provide a valuable complement to the more conventional methods. Next page →

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED 094120
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. Abstract (continued)

We first introduce and illustrate the technique with a number of examples. We then show that a correctness proof using the invariant assertion method or the subgoal induction method can always be expressed using intermittent assertions instead, but that the reverse is not always the case. The method can also be used just to prove termination, and any proof of termination using the conventional well-founded sets approach can be rephrased as a proof using intermittent assertions. Finally, we show how the method can be applied to prove the validity of program transformations and the correctness of continuously operating programs.

Manna & Waldinger

June 16, 1976

Is "sometime" sometimes better than "always"? Intermittent assertions in proving program correctness

Zohar Manna
Artificial Intelligence Lab
Stanford University
Stanford, Ca.

Richard Waldinger
Artificial Intelligence Center
Stanford Research Institute
Menlo Park, Ca.

Abstract

This paper explores a technique for proving the correctness and termination of programs simultaneously. This approach, which we call the *intermittent-assertion method*, involves documenting the program with assertions that must be true at some time when control is passing through the corresponding point, but that need not be true every time. The method, introduced by Knuth and further developed by Burstall, promises to provide a valuable complement to the more conventional methods.

We first introduce and illustrate the technique with a number of examples. We then show that a correctness proof using the invariant assertion method or the subgoal induction method can always be expressed using intermittent assertions instead, but that the reverse is not always the case. The method can also be used just to prove termination, and any proof of termination using the conventional well-founded sets approach can be rephrased as a proof using intermittent assertions. Finally, we show how the method can be applied to prove the validity of program transformations and the correctness of continuously operating programs.

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense under Contract DAHC15-73-C-0435, by the National Science Foundation under Grant GJ-36146 and by a grant from the United States-Israel Binational Science Foundation (BSF), Jerusalem, Israel.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, or the U.S. Government.

Table of Contents

I. Introduction

The most prevalent approach to prove that a program satisfies a given property has been the *invariant assertion method*, made known largely through the work of Floyd [1967] and Hoare [1969]. In this method, the program being studied is supplied with formal documentation in the form of comments, called *invariant assertions*, which express relationships between the different variables manipulated by the program. Such an invariant assertion is attached to a given point in the program with the understanding that the assertion is to hold every time control passes through the point.

Assuming that an appropriate invariant assertion, called the *input specification*, holds at the start of the program, the method allows us to prove that the other invariant assertions hold at the corresponding points in the program. In particular, we can prove that the *output specification*, the assertion associated to the program's exit, will hold whenever control reaches the exit. If this output specification reflects what the program is intended to achieve, we have succeeded in proving the correctness of the program.

It is in fact possible to prove that an invariant assertion holds at some point even though control never reaches that point, since then the assertion holds vacuously every time control passes through the point in question. In particular, using the invariant assertion method, one might prove that an output specification holds at the exit even though control never reaches that exit. If we manage to prove that a program's output specification holds, but neglect to show that the program terminates, we are said to have proved the program's *partial correctness*.

A separate proof, by a different method, is required to prove that the program does terminate. Typically, a termination proof is conducted by choosing a *well-founded set*, whose elements are ordered in such a way that no infinite decreasing sequences of elements exist. (The natural numbers under the regular less-than ordering, for example, constitute a well-founded set.) For some designated label within each loop of the program an expression involving the variables of the program is then selected whose value always belongs to the well-founded set. These expressions must be chosen so that each time control passes from one designated loop label to the next, the value of the expression corresponding to the second label is smaller than the value of the expression corresponding to the first label. Here, "smaller" means with respect to the *well-founded ordering*, the ordering of the chosen well-founded set. This establishes termination of the program, because if there were an infinite computation of the program, control would traverse an infinite sequence of loop labels; the successive values of the corresponding expressions would constitute an infinite decreasing sequence of elements of the well-founded set, thereby contradicting the defining property of the set. This *well-founded ordering method* constitutes the conventional way of proving the termination of a program (Floyd [1967]).

If a program both terminates and satisfies its output specification, that program is said to be *totally correct*.

Burstall [1974] advocated an approach whereby the total correctness of a program can be shown in a single proof (see also Schwarz [1975] and Topor [1976]). This technique, which he attributes to Knuth [1968] (Section 2.3.1), again involves affixing comments to points in the program but with the intention that *sometime* control will pass through the point and satisfy the attached assertion. Consequently, control may pass through a point many times without satisfying the assertion, but control must pass through the point at least once with the assertion satisfied; therefore we call these comments *intermittent assertions*. If we prove the output specification as an intermittent assertion at the program's exit, we have simultaneously shown that the program must halt and satisfy the specification. This establishes the program's total correctness. Since the conventional approach requires two separate proofs to establish total correctness, the intermittent-assertion method invites further attention.

We will use the phrase

sometime Q at L

to denote that Q is an intermittent assertion at label L (that sometime control will pass through L with assertion Q satisfied). If the entrance of a program is labelled *start* and its exit is labelled *finish*, we can express its total correctness with respect to an input specification P and an output specification R by

Theorem: if sometime P at *start*
then sometime R at *finish*.

This theorem entails the termination as well as the partial correctness of the program, because it implies that control must eventually reach the programs exit, and satisfy the desired output specification.

If we are only interested whether the program terminates, but don't care if it satisfies any particular output specification, we can try to prove

Theorem: if sometime P at *start*
then sometime at *finish*.

The conclusion "sometime at *finish*" expresses that control must eventually reach the program's exit, but does not require that any relation be satisfied.

Generally, to prove the total correctness or termination theorem for a program, we must affix

intermittent assertions to some of the program's internal points, and supply lemmas to relate these assertions. Typically, we will need a lemma for each of the program's loops, to describe the intended behavior of that loop.

The proofs of the lemmas often involve *complete induction over a well-founded ordering* (see Manna [1974]). In proving such a lemma we assume that the lemma holds for all elements of the well-founded set smaller (in the ordering) than a given element, and show that the lemma then holds for the given element as well.

In this paper, we will first present and illustrate the intermittent assertions method with a variety of examples. We will then show that the conventional invariant-assertion method and the well-founded ordering method, in addition to the more recent subgoal induction method (Manna [1971], Morris and Wegbreit [1976]) for proving partial correctness, can be regarded as special cases of the intermittent-assertion method. On the other hand, we present proofs of correctness and termination using intermittent assertions that are markedly simpler than their conventional counterparts. Finally, we show that the method can also be applied to establish the validity of program transformations, and to prove the correctness of continuously operating programs, programs that are intended never to terminate.

II. The Intermittent-Assertion Method

Rather than present a formal definition of the intermittent-assertion method, we prefer to illuminate it by means of a sequence of examples. Each example has been selected to illustrate a different aspect of the method.

1. Finding a zero of an array

We first consider a simple program, as a vehicle for introducing the basic technique. The program is given an array $a[1{:}n]$ of numbers, where $n \geq 1$, and is intended to find the index of an array element equal to zero, if such an element exists, or to output zero otherwise. The program, which we will call *find-index*, is

```
        input(a[1:n])
start:  index ← n
more:   if a[index] = 0
        then success: output(index)
        else index ← index-1
             if index = 0
             then failure: output(index)
             else goto more.
```

This program counts down from n until it succeeds in finding a zero element or else reaches index zero without finding such an element. It has two exits, *success* and *failure*, corresponding to successful and unsuccessful termination. In talking about this program, the abbreviation "$a[u{:}v]$ has a zero" will denote that at least one of $a[u]$, $a[u+1]$, ..., $a[v]$ is zero. (If $v < u$, then $a[u{:}v]$ has no zeros.) The total correctness of the *find-index* program is then expressed by the following theorem.

Theorem: if sometime $n \geq 1$ at *start*
then if $a[1{:}n]$ has a zero
then sometime $a[index] = 0$ and $1 \leq index \leq n$ at *success*
else sometime $index = 0$ at *failure*.

This theorem states the termination in addition to the partial correctness of the program because it implies that control must eventually reach one of the program's exits (*success* or *failure*) and satisfy the appropriate output specification.

In order to apply the intermittent assertion method, we are in general required to supply a lemma for each of the program's loops to describe the behavior of that loop. For the single loop of the given program, it suffices to prove the following lemma.

Lemma: if sometime *index* = i and $1 \le i \le n$ at *more*
then if $a[1{:}i]$ has a zero
then sometime $a[index] = 0$ and $1 \le index \le i$ at *success*
else sometime *index* = 0 at *failure*.

The hypothesis *index* = i in the antecedent enables us to refer to the original value of *index* in the conclusion, even though the value of *index* may have changed subsequently.

Note that the conclusion of the lemma is the same as the conclusion of the total correctness theorem with n replaced by i. This lemma implies the theorem directly, for assume we enter the program at *start* with input $n \ge 1$. Then we reach *more* immediately with *index* = n. The lemma then tells us (taking i to be n), that the conclusion of the theorem will ultimately be satisfied.

The proof of the lemma is by complete induction on i over the nonnegative integers; in other words, we will assume the lemma holds whenever *index* = i', for any nonnegative integer i' such that $i' < i$, and show that it then holds when *index* = i.

Suppose that

sometime *index* = i, and $1 \le i \le n$ at *more*.

We distinguish between two cases: If $a[i] = 0$, then clearly $a[1{:}i]$ has a zero; but then we reach *success* with *index* satisfying the conditions of the lemma, that $a[index] = 0$ and $1 \le index \le i$. On the other hand, if $a[i] \neq 0$, then

(*) $a[1{:}i]$ has a zero <=> $a[1{:}i-1]$ has a zero

and *index* is reset to $i-1$. If $i-1 = 0$, then $a[1{:}i-1]$, and hence $a[1{:}i]$, has no zeros; but indeed in this case we reach the failure exit with *index* = $i-1 = 0$. Alternatively, if $i-1 \neq 0$, we return to *more* with *index* = $i-1$, where $1 \le i-1 < n$. Since $i-1 < i$, our induction hypothesis applies, with *index* = $i-1$, telling us that

if $a[1{:}i-1]$ has a zero
then sometime $a[index] = 0$ and $1 \le index \le i-1$ at *success*
else sometime *index* = 0 at *failure*.

However, $a[1{:}i-1]$ has a zero if and only if $a[1{:}i]$ has a zero (by *), and $index \le i-1$ implies $index \le i$. Consequently we have established the conclusion of the lemma, and completed the proof of the total correctness of the *find-index* program.

The chief difficulties in devising such a proof are formulating the lemmas and choosing the ordering on which to base the mathematical induction in the proof. The conventional methods, however, require comparable acts of ingenuity. For instance, suppose we want to apply the invariant assertion method to prove the partial correctness of the program with respect to the input specification

$$n \geq 1,$$

and the output specification

if $a[1{:}n]$ has a zero
then $a[index] = 0$ and $1 \leq index \leq n$
else $index = 0$,

associated with both the success and the failure exits. (Note that the invariant-assertion method does not allow us to distinguish between the roles of the two exits.) Then we must devise an invariant assertion at *more*, such as

$1 \leq index \leq n$ and
$a[index{+}1{:}n]$ has no zeroes,

or, alternatively,

$1 \leq index \leq n$ and
$a[1{:}n]$ has a zero <=> $a[1{:}index]$ has a zero.

Furthermore, if we wish to prove termination using the conventional method, we must devise a decreasing expression over an appropriate well-founded ordering: for instance, we can use the same expression (*index*) and the same well-founded set (the nonnegative integers) that we used as the basis for the complete induction under the intermittent-assertion method. We will make these connections more precise in Section III-3.

2. The greatest common divisor of two numbers

Because the *find-index* program involved only a single loop, a single lemma sufficed to prove its total correctness. The following program, which computes the greatest common divisor (*gcd*) of two positive integers, is introduced to show how the intermittent-assertion method is applied to a program with a more complex loop structure.

We define $gcd(x\ y)$, where x and y are positive integers, as the greatest integer that evenly divides both x and y, that is,

$$gcd(x\ y) = max\{u : u|x \text{ and } u|y\}.$$

For instance, $gcd(9\ 12) = 3$ and $gcd(12\ 25) = 1$.

The program is

```
        input(x y)
start:
more:   if x = y
        then finish: output(y)
        else reducex: if x > y
                      then  x ← x-y
                            goto reducex
             reducey: if x < y
                      then  y ← y-x
                            goto reducey
             goto more.
```

This program is based on the properties of the *gcd* that $gcd(x\ y) = y$ if $x=y$, $gcd(x\ y) = gcd(x-y\ y)$ if $x > y$, and $gcd(x\ y) = gcd(x\ y-x)$ if $x < y$.

We would like to use the intermittent-assertion method to prove the total correctness of the gcd program. The total correctness can be expressed as follows:

Theorem: if sometime $x = a$, $y = b$ and $a,b > 0$ at *start*
then sometime $y = max\{u : u|a \text{ and } u|b\}$ at *finish*

To prove this theorem, we need a lemma that describes the behavior of each of the program's loops. For instance, corresponding to the outer loop we have

Lemma 1 (outer loop):
if sometime $x = a_1$, $y = b_1$ and $x,y > 0$ at *more*
then sometime $x = y$ and $y = max\{u : u|a_1 \text{ and } u|b_1\}$ at *more*.

This lemma summarizes the effects of the program's outer loop.

Lemma 1 implies the theorem directly: We assume that initially $x = a$, $y = b$, and $a,b > 0$ at *start*; we immediately reach *more* again, with $x = a$, $y = b$, and $a,b > 0$. Lemma 1 implies that sometime we will return to *more*, with $x = y = max\{u : u|a \text{ and } u|b\}$. Since $x = y$, we will then reach *finish*, with $y = max\{u : u|a \text{ and } u|b\}$.

Because there are two inner loops within the program's outer loop, to prove Lemma 1 requires the following lemmas, each describing the behavior of the corresponding inner loop.

Lemma 2 (top inner loop):

if sometime $x = a_2$, $y = b_2$, $x > y$ and $x,y > 0$ at *reducex*
then sometime $x < a_2$, $y = b_2$, $x \le y$, $x,y > 0$ and
$\{u : u|x \text{ and } u|y\} = \{u : u|a_2 \text{ and } u|b_2\}$ at *reducex*.

Lemma 3 (bottom inner loop):

if sometime $x = a_3$, $y = b_3$, $x < y$ and $x,y > 0$ at *reducey*
then sometime $x = a_3$, $y < b_3$, $x \ge y$, $x,y > 0$ and
$\{u : u|x \text{ and } u|y\} = \{u : u|a_3 \text{ and } u|b_3\}$ at *reducey*.

Lemma 2 states that the top inner loop will reduce x until it is less than or equal to y but will not change y or the set of common divisors of x and y. Lemma 3 is similar, but the roles of x and y are reversed. Let us first prove Lemma 2

Proof of Lemma 2: This proof is by complete induction on a_2; in other words, we will assume that Lemma 2 holds whenever $x = a_2'$, for any non-negative integer a_2' such that $a_2' < a_2$, and show that the lemma then holds when $x = a_2$.

Suppose that

sometime $x = a_2$, $y = b_2$, $x > y$ and $x,y > 0$ at *reducex*.

Since $x > y$, x is reset to $x-y$, *i.e.* to a_2-b_2, which is still positive, and control returns to *reducex*. Therefore,

sometime $x = a_2-b_2$, $y = b_2$, $x,y > 0$ and
$\{u : u|x \text{ and } u|y\} = \{u : u|a_2 \text{ and } u|b_2\}$ at *reducex*.

The set of common divisors of x and y is unchanged because the common divisors of a_2-b_2 and b_2 are the same as those of a_2 and b_2. We must now distinguish between two cases.

Case $x \le y$: The conclusion of Lemma 2 is already satisfied, since $b_2 > 0$ and, therefore, $x = a_2-b_2 < a_2$.

Case $x > y$: Since $x = a_2-b_2 < a_2$, our induction hypothesis applies, telling us that

sometime $x < a_2-b_2$, $y = b_2$, $x \le y$, $x,y > 0$ and
$\{u : u|x \text{ and } u|y\} = \{u : u|a_2-b_2 \text{ and } u|b_2\}$ at *reducex*.

We can conclude that $x < a_2$ (because $b_2 > 0$) and that

$\{u : u|x \text{ and } u|y\} = \{u : u|a_2 \text{ and } u|b_2\}$

since the common divisors of a_2-b_2 and b_2 are the same as those of a_2 and b_2, thereby completing the proof of Lemma 2.

The proof of Lemma 3 is similar to that of Lemma 2, except that induction is on b_3 instead of a_2. It remains only to use both Lemmas in proving Lemma 1.

Proof of Lemma 1: The proof is by complete induction on b_1; thus, we will assume that Lemma 1 holds whenever $y = b_1'$, for any non-negative integer b_1' such that $b_1' < b_1$, and show the lemma then holds when $y = b_1$.

Assume that

sometime $x = a_1$, $y = b_1$, and $x,y > 0$ at *more*.

If $x = y$, the conclusion of the lemma,

sometime $x = y$ and $y = max\{u : u|a_1 \text{ and } u|b_1\}$ at *more*,

is already satisfied because the greatest common divisor of x and y is then the greatest divisor of y, namely y itself. If $x \neq y$, then either $x > y$ and we pass to *reducex*, or $x < y$ and we pass to *reducey*. We treat these two cases separately.

Case $x > y$ at *reducex*: Here, we know $x = a_1$, $y = b_1$, $x > y$ and $x,y > 0$. Thus, by Lemma 2, we have

sometime $x < a_1$, $y = b_1$, $x \leq y$, $x,y > 0$ and

(*) $\{u : u|x \text{ and } u|y\} = \{u : u|a_1 \text{ and } u|b_1\}$ at *reducex*.

If $x = y$ at this point, we are done: we pass directly to *more*, and y is already the greatest common divisor of x and y (by the argument given earlier in this proof) and therefore of a_1 and b_1, by (*). On the other hand, if $x < y$ we pass to *reducey*, and the situation is treated in the next case.

Case $x < y$ at *reducey*: Whether we have travelled around the top inner loop or reached this point directly, we know that $x \leq a_1$, $y = b_1$, $x < y$, $x,y > 0$, and $\{u : u|x \text{ and } u|y\} = \{u : u|a_1 \text{ and } u|b_1\}$. Lemma 3 implies that

sometime $x \leq a_1$, $y < b_1$, $x \geq y$, $x,y > 0$ and

(**) $\{u : u|x \text{ and } u|y\} = \{u : u|a_1 \text{ and } u|b_1\}$ at *reducey*.

The formula (**) holds because Lemma 3 tells us that the common divisors of x and y are still unchanged. We can then pass directly to *more* and, if $x = y$, the conclusions of the lemma are satisfied as before. On the other hand, if $x \neq y$, the value of y has been reduced, and therefore

the induction hypothesis applies. If we denote the current values of x and y by a_1' and b_1', the induction hypothesis implies that

sometime $x = y$ and $y = max\{u : u|a_1' \text{ and } u|b_1'\}$ at *more*,

or, applying (**), that

$y = max\{u : u|a_1 \text{ and } u|b_1\}$.

This satisfies the conclusion of Lemma 1.

This example shows how the use of the lemmas in proving the total correctness theorem reflects the loop structure of the program.

It is not difficult to prove the partial correctness of the above program using the conventional invariant assertion method. For instance, to prove that the program is partially correct with respect to the input specification

$x_0 > 0$ and $y_0 > 0$

and output specification

$y = gcd(x_0\ y_0)$

(where x_0 and y_0 are the initial values of x and y) we can use the same invariant assertion

$x,y > 0$ and $\{u : u|x \text{ and } u|y\} = \{u : u|x_0 \text{ and } y_0\}$

at each of the labels *more*, *reducex* and *reducey*.

In contrast, the termination of this program is awkward to prove by the conventional well-founded ordering method, because it is possible to pass from *more* to *reducex*, *reducex* to *reducey*, or from *reducey* to *more* without changing any of the program variables. One of the simplest proofs of the termination of the *gcd* program by this method involves taking the well-founded set to be the pairs of non-negative integers ordered by the regular lexicographic ordering; that is,

$(u_1\ v_1) < (u_2\ v_2)$

if and only if

$u_1 < u_2$ or
$u_1 = u_2$ and $v_1 < v_2$.

When the expressions corresponding to the loop labels are taken to be

$(x+y\ 2)$	at *more*,
if $x \neq y$ then $(x+y\ 1)$ else $(x+y\ 4)$	at *reducex*, and
if $x < y$ then $(x+y\ 0)$ else $(x+y\ 3)$	at *reducey*,

it can be shown that their successive values decrease as control passes from one loop label to the next (Katz and Manna [1975]). Although this method is effective, it is not the most natural in establishing the termination of the *gcd* program.

3. The Ackermann Function

Another example for which the termination proof is even more difficult to accomplish by the well-founded ordering method is an iterative program to compute the Ackermann function. This function, denoted by $A(x\ y)$, is defined recursively for non-negative integers x and y as

$$
\begin{aligned}
A(x\ y) \Leftarrow\ & \text{if } x = 0 \\
& \text{then } y+1 \\
& \text{else if } y = 0 \\
& \quad \text{then } A(x-1\ 1) \\
& \quad \text{else } A(x-1\ A(x\ y-1)).
\end{aligned}
$$

This function is of theoretical interest, in part because it value grow extremely quickly; for instance,

$$A(4\ 4) = 2^{2^{2^{2^{16}}}}$$

An iterative program to compute the same function is

```
        input(x0 y0)
start:  stack[1] ← x0
        stack[2] ← y0
        index ← 2
more:   if index = 1
        then finish: output(stack[1])
        else if stack[index-1] = 0
             then stack[index-1] ← stack[index]+1
                  index ← index-1
                  goto more
             else if stack[index] = 0
                  then stack[index-1] ← stack[index-1]-1
                       stack[index] ← 1
                       goto more
                  else stack[index+1] ← stack[index]-1
                       stack[index] ← stack[index-1]
                       stack[index-1] ← stack[index-1]-1
                       index ← index+1
                       goto more.
```

This iterative program represents a direct translation of the recursive definition; at each iteration it can be shown that

$$A(stack[1]\ A(stack[2]\ \ldots\ A(stack[index-1]\ stack[index]))) = A(x_0\ y_0)$$

at *more* and, when the program terminates, that

$$stack[1] = A(x_0\ y_0).$$

The only known proof of the termination of this program using the conventional well-founded ordering method (by A. Pnueli) is extremely complicated, and we challenge the intrepid reader to construct such a proof.

The intermittent assertion proof allows us to show total correctness of the same program in a markedly more simple fashion. The theorem that expresses the program's total correctness is

Theorem: if sometime $x_0, y_0 \geq 0$ at *start*
then sometime $stack[1] = A(x_0\ y_0)$ at *finish*.

In proving this theorem we will employ the following lemma,

Lemma: if sometime $index = i$, $i \geq 2$, $(stack[1]\ stack[2]\ \ldots\ stack[i-2]) = s$,
$stack[i-1] = a$ and $stack[i] = b$ at *more*,
then sometime $index = i-1$, $(stack[1]\ stack[2]\ \ldots\ stack[i-2]) = s$
and $stack[i-1] = A(a\ b)$ at *more*.

Here, s represents a tuple of stack elements. The expression $(stack[1]\ stack[2]\ \ldots\ stack[i-2]) = s$ is included in the hypothesis and the conclusion of the lemma to convey that the initial segment of the array, the first $i-2$ elements, are unchanged when we return to *more*.

It is straightforward to see that the Lemma implies the Theorem. For *index* is 2, $stack[1]$ is x_0, and $stack[2]$ is y_0 the first time we reach *more*. Then the lemma implies that eventually we will reach *more* again, with $index=1$ and $stack[1] = A(x_0\ y_0)$. Since $index = 1$ we then pass to *finish* with the desired output.

To prove the lemma let us suppose

sometime $index = i$, $i \geq 2$, $(stack[1]\ stack[2]\ \ldots\ stack[i-2]) = s$,
$stack[i-1] = a$ and $stack[i] = b$ at *more*.

Our proof will be by induction on the pair $(stack[index-1]\ stack[index])$ under the lexicographic ordering over the non-negative integers; in other words, we will assume the lemma holds whenever $stack[index-1] = a'$ and $stack[index] = b'$, where a' and b' are any non-negative integers such that $a' < a$, or $a' = a$ and $b' < b$, and show that it then holds when $stack[index-1]=a$ and $stack[index]=b$. The proof distinguishes between three cases, corresponding to the conditional tests in the recursive definition of the Ackermann function.

Case $a = 0$: Then $A(a\ b) = b+1$ by the recursive definition of the Ackermann function. But since $index \neq 1$, and $stack[index-1] = a = 0$, we return to *more* with $index = i-1$ and $stack[i-1] = b+1$, satisfying the conclusion of the lemma.

Case $a > 0$, $b = 0$: Here, $A(a\ b) = A(a-1\ 1)$ by the definition of the Ackermann function. Because $index \neq 1$, $stack[index-1] = a \neq 0$ and $stack[index] = b = 0$, we return to *more* with $index = i$, $stack[i-1] = a-1$, and $stack[i] = 1$. Since $stack[i-1] = a-1 < a$, we have

$$(stack[i-1]\ stack[i]) = (a-1\ 1) < (a\ 0),$$

and, therefore, the inductive hypothesis can be applied, to yield that

sometime $index = i-1$, $(stack[1]\ stack[2]\ \ldots\ stack[i-2]) = s$ and
$stack[i-1] = A(a-1\ 1)$ at *more*.

Because $A(a\ b) = A(a-1\ 1)$, the lemma is established in this case.

Case $a > 0$, $b > 0$: Then $A(a\ b) = A(a\text{-}1\ A(a\ b\text{-}1))$, by the recursive definition. Since $index \neq 1$, $stack[index\text{-}1] = a \neq 0$, and $stack[index] = b \neq 0$, we return to *more* with

$index = i+1$,
$stack[i-1] = a-1$,
$stack[i] = a$, and
$stack[i+1] = b-1$.

Because $index = i+1$ and $(stack[i]\ stack[i+1]) = (a\ b\text{-}1) < (a\ b)$, our induction hypothesis applies, yielding

sometime $index = i$, $(stack[1], stack[2]\ \ldots\ stack[i\text{-}2]) = s$,
$stack[i-1] = a-1$, and $stack[i] = A(a\ b\text{-}1)$.

Note that we could conclude that $stack[i-1] = a-1$ because the induction hypothesis, for $index = i+1$, states that the first $i-1$ array elements are unchanged. Since $index = i$ and $(stack[i-1]\ stack[i]) = (a\text{-}1\ A(a\ b\text{-}1)) < (a\ b)$, we can apply the induction hypothesis once more, to obtain that

sometime $index = i-1$, $(stack[1]\ stack[2]\ \ldots\ stack[i\text{-}2]) = s$,
and $stack[i-1] = A(a\text{-}1\ A(a\ b\text{-}1))$ at *more*,

which is the desired conclusion in this case.

This completes the proof of the total correctness of our Ackermann program. We believe it reflects our understanding of the way the program works.

4. Counting the tips of a tree

For all of the above examples (the *find-index*, *gcd*, and *Ackermann* programs) it is possible to find a simple partial correctness proof using invariant assertions. Thus, none of these examples illustrate a certain advantage of the intermittent-assertion method over the conventional methods for proving partial correctness. Our next example presents a situation in which a straightforward proof of total correctness using intermittent assertions is possible, but for which the proof of partial correctness with invariant assertions is peculiarly difficult.

This example, included in Burstall [1974], is an algorithm to count the *tips* of a binary tree, those nodes that have no descendents. A recursive definition of a function *tips(tree)* that counts the tips of a binary tree *tree* is

tips(tree) <= if *tree* is a tip
then 1
else *tips(left(tree))* + *tips(right(tree))*,

where *left*(*tree*) and *right*(*tree*) are the left and right subtrees of *tree* respectively.

An iterative program to count the tips of a binary tree *tree* is

```
        input(tree)
start:  stack ← ()
        count ← 0
more:   if tree is a tip
        then  count ← count+1
              teststack:  if stack = ()
                          then finish:  output(count)
                          else  (tree stack) ← (head(stack) tail(stack))
                                goto more
        else  (tree stack) ← (left(tree)  right(tree)· stack)
              goto more.
```

Here, the value of *stack* is a list. We denote the empty list by (). If *stack* is not empty, then *head*(*stack*) is its first element and *tail*(*stack*) is the list of its remaining elements. We denote by $x \cdot stack$ the list formed by adding the element x at the beginning of *stack*. The assignment (*tree stack*) ← (*head*(*stack*) *tail*(*stack*)) means that *tree* and *stack* are simultaneously set to the value of *head*(*stack*) and *tail*(*stack*), respectively.

This program operates by using the variable *count* to be the number of tips already counted, *tree* to be the subtree of the original tree whose tips are now being counted, and *stack* to be the list of subtrees whose tips have yet to be counted. Its partial correctness can be established taking the invariant

$$tips(tree_0) = count + tips(tree) + \sum_{s \in stack} tips(s) \text{ at } more,$$

where $tree_0$ is the given tree, and $\sum_{s \in stack} tips(s)$ is the sum of the tips of all the subtrees on *stack*.

To prove termination with the conventional well-founded ordering approach, we can show that the value of the expression

$$(tips(tree_0) - count \;\; tips(tree))$$

always decreases in the lexicographic ordering each time we return to *more*, but always remains in the set of pairs of nonnegative integers.

If we use intermittent assertions, we can express the total correctness of this program in the following theorem.

Theorem: if sometime *tree* = *t* at *start*
then sometime *count* = *tips*(*t*) at *finish*.

The proof of this theorem depends on the following lemma, that describes the behavior of the program in more detail.

Lemma: if sometime *tree* = *t*, *count* = *c* and *stack* = *s* at *more*
then sometime *count* = *c*+*tips*(*t*) and *stack* = *s* at *teststack*.

It is not difficult to see that this lemma implies the theorem. Suppose

sometime *tree* = *t* at *start*.

Then we set *stack* to (), *count* to 0, and reach *more*, so that

sometime *tree* = *t*, *count* = 0 and *stack* = () at *more*.

The lemma then tells us that

sometime *count* = 0+*tips*(*t*) and *stack* = () at *teststack*.

Because *stack* is (), we reach *finish*, so that

sometime *count* = *tips*(*t*) at *finish*,

and the theorem is thereby established.

The proof of the lemma is by complete induction on the structure of *tree*. We suppose that

sometime *tree* = *t*, *count* = *c*, and *stack* = *s* at *more*,

and we assume inductively that the lemma holds whenever *tree* = *t'*, where *t'* is any subtree of *t*. The proof distinguishes between two cases, depending on the current value *t* of *tree*.

Case *t* is a tip: Then *tips*(*t*) = 1 by the recursive definition of *tips*; *count* is then increased by 1, and we reach *teststack* with *count* = *c*+1 = *c*+*tips*(*t*), establishing the conclusion of the lemma.

Case *t* is not a tip: Then *tips*(*t*) = *tips*(*left*(*t*))+*right*(*left*(*t*)), by the recursive definition of *tips*. Since *t* is not a tip, we pass around the else branch of the loop and return to *more*, so that

sometime $tree = left(t)$, $count = c$ and $stack = right(t) \cdot s$ at *more*.

Because $left(t)$ is a subtree of t, our induction hypothesis can be applied to yield

sometime $count = c + tips(left(t))$ and $stack = right(t) \cdot s$ at *teststack*.

Since *stack* is nonempty (it contains $right(t)$), we pass around the loop once again, setting *tree* to $head(stack)$ and *stack* to $tail(stack)$, so that

sometime $tree = right(t)$, $count = c + tips(left(t))$ and $stack = s$ at *more*.

Since $right(t)$ is a subtree of t, we can apply our induction hypothesis a second time, obtaining

sometime $count = c + tips(left(t)) + tips(right(t))$ and $stack = s$ at *teststack*.

In other words, since $tips(t) = tips(left(t)) + tips(right(t))$,

sometime $count = c + tips(t)$, and $stack = s$ at *teststack*.

This is the desired conclusion of the lemma.

Part of the difficulty in applying the conventional invariant-assertion method here is that this iterative program is derived from a recursive definition, with a stack introduced to circumvent the need for recursion. Both the iterative and the recursive programs count the tips of every subtree of the given tree. For the recursive program, each recursive call counts the tips of a single subtree; however, for the iterative program, counting the tips of the subtree may involve traversing the loop many times. The intermittent-assertion method allows us to relate the point at which we are about to count the tips of a subtree t (at *more*) with the point at which we have completed the counting (at *teststack*), and to consider all the computation between these points as a single unit, which corresponds naturally to a single recursive call of $tips(t)$.

On the other hand, the invariant assertion method forces us to relate the point at which we are about to count the tips of the subtree t with the point after we have traversed the loop only once. There may be no natural connection between these points: for instance, in travelling around the loop we may have reset *tree* to be the head of the stack. Our invariant assertion must be exceptionally complete, and even include a description of the stack, so that we can relate these two points.

The intermittent-assertion method is particularly useful for such iterative programs as our *tips* program that are derived from recursive definitions. We will discuss this class of programs further in Section IV.

III. Relation to Conventional Proof Techniques

One question that naturally arises in presenting a new proof technique is its relationship to the more conventional methods? In the previous section we argued that, in some cases, the intermittent-assertion method enjoys certain advantages. In this section we will show that this method effectively can replace the others. In particular, we directly rephrase a partial-correctness proof using either of two standard techniques as a partial-correctness proof using intermittent assertions. Furthermore, any termination proof using the well-founded ordering method can be expressed using intermittent assertions instead. Therefore, we can always use the intermittent-assertion method in place of the established techniques.

For simplicity, we will employ a simple one-loop abstract program to illustrate these constructions,

```
        input(x_0)
start:  x ← f(x_0)
more:   if t(x)
        then   x ← h(x)
               finish: output(x)
        else   x ← g(x)
               goto more.
```

The results we present, however, apply to concrete programs with arbitrary loop structure.

For each computation, a program of the above form defines an implicit ordering on the possible values of the program's variable x: This is the ordering $\succ$ defined as the transitive closure of the relation

$$a \succ g(a),$$

where a and $g(a)$ are two successive values of x at the label *more* as control passes around the loop. We will call $\succ$ the *ordering induced by the computation*. If the computation terminates, this ordering is well-founded: because, if the ordering admitted an infinite decreasing sequence, the computation itself would have to be infinite.

The above ordering enables us to apply the intermittent-assertion method to prove partial correctness, even though the method is basically a total correctness technique. We include in the theorem an explicit assumption that the computation terminates. Since the output specification is only claimed to hold if the program halts, the theorem will express partial rather than total correctness. The proof of the corresponding lemma will involve a complete induction over the ordering induced by the computation, which is well-founded because the computation has been assumed to terminate.

Now let us see in more detail how an invariant-assertion proof of the partial correctness of our program can be rephrased using intermittent assertions.

1. Invariant-Assertion Method

Suppose that we have used the invariant assertion technique to prove that our program is partially correct with respect to some input specification $P(x_0)$ and output specification $R(x_0\ x)$. Then we have found an invariant assertion $Q(x_0\ x)$ with the property that every time we pass through *more*, $Q(x_0\ x)$ will be true for the current value of x, and we have proved the three conditions

(1) $P(x_0)$ => $Q(x_0\ f(x_0))$ for every x_0
(the invariant assertion holds on entry to the loop),

(2) $P(x_0)$ and $Q(x_0\ x)$ and $\neg t(x)$ => $Q(x_0\ g(x))$ for every x_0 and x
(the invariant assertion is maintained as control passes around the loop), and

(3) $P(x_0)$ and $Q(x_0\ x)$ and $t(x)$ => $R(x_0\ h(x))$ for every x_0 and x
(the invariant assertion implies the output specification after exit from the loop).

Conditions (1) and (2) tell us that $Q(x_0\ x)$ is indeed an invariant assertion of our program, and condition (3) says that if $Q(x_0\ x)$ holds when we exit from the loop, then the desired output specification will be satisfied. Together these conditions imply the partial correctness of our simple program.

If we can prove the three conditions necessary in establishing the partial correctness of the program using an invariant assertion, then we can use the intermittent-assertion method instead. The proof will require the same three conditions, and the ordering induced by the computation will serve as the basis for the complete induction. The theorem that expresses the partial correctness is as follows.

Theorem: if sometime $P(x_0)$ at *start*
and if the computation terminates
then sometime $R(x_0\ x)$ at *finish*.

We can always prove this theorem using a lemma of form

Lemma: if sometime $x = a$, $P(x_0)$ and $Q(x_0\ a)$ at *more*
and if the computation terminates
then sometime $Q(x_0\ x)$ and $t(x)$ at *more*.

First let us show that the lemma actually does imply the theorem. Assume that

sometime $P(x_0)$ at *start*.

Then we reach *more* at once with $x = f(x_0)$. The condition (1), which we used in proving the partial correctness of this program, says that

$$P(x_0) \Rightarrow Q(x_0\ f(x_0))$$

for all x_0. Since $P(x_0)$ now holds at *more*, we can conclude that $Q(x_0\ f(x_0))$ also holds at *more*. Because we also have assumed that the computation terminates, we can apply the lemma to yield

sometime $Q(x_0\ x)$ and $t(x)$ at *more*.

Our condition (3), however, states that

$$P(x_0) \text{ and } Q(x_0\ x) \text{ and } t(x) \Rightarrow R(x_0\ h(x))$$

for every x_0 and x. Since we know $P(x_0)$, $Q(x_0\ x)$ and $t(x)$ hold at *more*, we can deduce $R(x_0\ h(x))$ at *more* as well. Since $t(x)$ holds, we pass directly to *finish*, setting x to $h(x)$, so that now $R(x_0\ x)$ holds at *finish*, as we intended to prove.

To prove the lemma, suppose that

sometime $x = a$, $P(x_0)$ and $Q(x_0\ a)$ at *more*,

and that the computation terminates. The proof is by complete induction on a, using the ordering $\prec$ induced by the computation. Since we are assuming that the computation terminates, we know that $\prec$ is well-founded. We therefore assume inductively that the lemma holds whenever $x = a'$, for any a' such that $a' \prec a$, and show that it holds when $x = a$ as well. In the proof we distinguish between two cases, depending on whether $t(a)$ holds.

Case $t(a)$ holds: Because $x = a$, both $Q(x_0\ x)$ and $t(x)$ hold at *more*, and the conclusion of the lemma is already satisfied.

Case $\neg t(a)$ holds: Recall that condition (2) says that

$$P(x_0) \text{ and } Q(x_0\ x) \text{ and } \neg t(x) \Rightarrow Q(x_0\ g(x))$$

for every x_0 and x. Since $P(x_0)$, $Q(x_0\ a)$ and $\neg t(a)$ all hold at *more*, we can conclude that $Q(x_0\ g(a))$ holds at *more* as well. Since we have assumed $\neg t(a)$, we pass around the loop, returning to *more* with x reset to $g(a)$. But now, because

$$g(a) \prec a$$

in the ordering $\prec$ induced by the computation, our induction hypothesis applies with $x = g(a)$. This allows us to infer that

sometime $Q(x_0\ x)$ and $t(x)$ at *more*

which completes the proof of the lemma.

We have thus constructed a proof of the partial correctness of the program using intermittent assertions, assuming that we could prove its partial correctness with invariant assertions.

2. Subgoal induction method

The invariant-assertion approach always relates the current values of the program variables to their initial values. Another approach for proving partial correctness, the *subgoal induction method*, relates these variables to their ultimate values when the program halts. We will first present the method, and then show as before that if we have proved the partial correctness of a program using this method, then we can rephrase the same proof with intermittent assertions instead. Again, we will demonstrate the construction on the one-loop abstract program

```
        input(x0)
start:  x ← f(x0)
more:   if t(x)
        then  x ← h(x)
              finish:  output(x)
        else  x ← g(x)
              goto more.
```

To prove the partial correctness of this program with respect to the input specification $P(x_0)$ and output specification $R(x_0\ x)$ by the subgoal induction method, we need to find a *subgoal assertion* $Q(u\ v)$ at *more*: the intuitive meaning of this assertion is that $Q(u\ v)$ must hold whenever u is the current value of x at *more* and v is the ultimate value of x at *finish*. To establish the partial correctness of the program, we need to prove the following three conditions:

(1) $t(x)$ => $Q(x\ h(x))$ for every x
(the subgoal assertion holds when we are about to exit from the loop),

(2) $Q(g(x)\ z)$ and $\neg t(x)$ => $Q(x\ z)$ for every x and z
(if we are about to go around the loop, and if the subgoal assertion holds for the value of x after passing around the loop once more, then the subgoal assertion holds for the current value of x as well), and

(3) $P(x_0)$ and $Q(f(x_0)\ z)$ => $R(x_0\ z)$ for every x_0 and z
(if the subgoal assertion holds the first time we reach *more*, then the output specification is satisfied).

Subgoal induction works backward through the computation, whereas the invariant assertion method works forward. Condition (1) implies that the subgoal assertion applies to the current value of x when we are about to exit from the loop. Condition (2) says that if the subgoal assertion applies to the subsequent value of x, it applies to the current value of x as well. If the program terminates, therefore, conditions (1) and (2) together imply that the subgoal assertion applies to all values assumed by x within the loop. Finally, condition (3) states that, if the subgoal assertion holds for the initial value assumed by x within the loop, the desired output specification is satisfied. Combined, these three conditions establish the partial correctness of the program.

Because many readers may be unfamiliar with the subgoal induction method, we will illustrate it with a very simple program to compute the *gcd*:

```
        input(x0 y0)
start:  (x y) ← (x0 y0)
more:   if x=0
        then finish: output(y)
        else (x y) ← (rem(y x) x)
             goto more.
```

This program differs slightly from our abstract program in that it has two program variables, x and y.

Suppose we are to show that this program is partially correct with respect to the input specification

$P(x_0\ y_0)$: $x_0 > 0$ and $y_0 > 0$,

and the output specification

$$R(x_0\ y_0\ y):\quad y = max\{u : u|x_0 \text{ and } u|y_0\},$$

using subgoal induction. If we take the subgoal assertion to be

$$Q(x\ y\ z):\quad z = max\{u : u|x \text{ and } u|y\},$$

the three conditions to prove the partial correctness are then

(1) $x=0 \Rightarrow y = max\{u : u|x \text{ and } u|y\}$ for every x and y,

(2) $z = max\{u : u|rem(y\ x) \text{ and } u|x\}$ and $x\neq 0 \Rightarrow z = max\{u : u|x \text{ and } u|y\}$ for every x, y and z, and

(3) $x_0>0$ and $y_0>0$ and $z=max\{u : u|x_0 \text{ and } u|y_0\} \Rightarrow z=max\{u : u|x_0 \text{ and } u|y_0\}$ for every x_0, y_0 and z.

These three conditions are easily seen to be true, and the partial correctness is established.

Note that although the subgoal assertion for the above program is derived directly from the output specification, the invariant assertion for proving partial correctness of this program,

$$\{u : u|x \text{ and } u|y\} = \{u : u|x_0 \text{ and } u|y_0\},$$

requires some ingenuity to construct.

If we have used subgoal induction to prove partial correctness, we can construct an analogous proof with intermittent assertions instead, just as we did for the invariant-assertion method. The theorem that expresses the partial correctness of the program is again

Theorem: *if sometime* $P(x_0)$ *at start*
and if the computation terminates
then sometime $R(x_0\ x)$ *at finish.*

The lemma used to prove the theorem is different:

Lemma: *if sometime* $x = a$ *at more*
and if the computation terminates
then sometime $Q(a\ x)$ *at finish.*

The difference between the two lemmas reflects the difference in meaning between the invariant and subgoal assertions.

To see that the lemma implies the theorem, assume that

sometime $P(x_0)$ at *start*

and that the computation terminates; we then reach *more* immediately with $x = f(x_0)$. The lemma implies that

sometime $Q(f(x_0)\ x)$ at *finish*.

Since our condition (3) states that

$P(x_0)$ and $Q(f(x_0)\ z) \Rightarrow R(x_0\ z)$

for every x_0 and z, we can conclude that $R(x_0\ x)$ also holds at *finish*, as we intended to prove.

The proof of the lemma is again by complete induction on a, using the ordering $\prec$ induced by the computation, which is well-founded because the computation has been assumed to terminate. Suppose that

sometime $x = a$ at **more**.

We assume inductively that the lemma holds whenever $x = a'$, for any a' such that $a' \prec a$, and show that it holds when $x = a$. In the proof we distinguish between two cases, depending on whether or not $t(a)$ holds.

Case $t(a)$ holds: By condition (1),

$t(x) \Rightarrow Q(x\ h(x))$

for every x; therefore, $Q(a\ h(a))$ holds at *more*. Because $t(a)$ is true, x is set to $h(a)$ so that $Q(a\ x)$ holds, and control passes to *finish*, to satisfy the conclusion of the lemma.

Case $\neg t(a)$ holds: Here, the computation passes around the loop, setting x to $g(a)$ and returning to *more*. Since

$g(a) \prec a$

in the ordering $\prec$ induced by the computation, our induction hypothesis applies with $x = g(a)$, to yield that

sometime $Q(g(a)\ x)$ at *finish*.

Recall that condition (2) says that

$$Q(g(x)\ z) \text{ and } \neg t(x) \Rightarrow Q(x\ z)$$

for every x and z. Because $Q(g(a)\ x)$ now holds at *finish*, and we have assumed $\neg t(a)$, we can conclude that $Q(a\ x)$ at *finish*. This completes the proof of the lemma.

Again, although this demonstration was only conducted for our simple one-loop abstract program, it applies to programs with a more complex loop structure as well. This result allows the use of the intermittent-assertion method to mimic any proof of partial correctness by subgoal induction.

We have remarked that the invariant-assertion method relates the current values of the program variables to their initial values, whereas the subgoal-induction method relates the current values to their final values. The intermittent-assertion technique can imitate both of these methods because it can relate the values of the program variables at any two stages in the computation.

3. Well-founded ordering method

The above constructions enabled us to mirror conventional partial-correctness proofs using intermittent assertions. In fact, we can also use the intermittent-assertion method to express conventional termination proofs that use the well-founded set approach. We will again use the simple one-loop abstract program:

```
        input(x0)
start:  x ← f(x0)
more:   if t(x)
        then  x ← h(x)
              finish: output(x)
        else  x ← g(x)
              goto more.
```

To prove the termination of this program by the conventional method, we must find a well-founded set W with ordering $<$ and an expression $E(x)$ such that

(1) $E(a) \in W$ for every value a of x at *more*
(the value of the expression at *more* always belongs to the well-founded set), and

(2) $E(g(a)) < E(a)$, where a and $g(a)$ are any two successive values of x at *more*
(the value of the expression is always reduced in the well-founded ordering when control passes around the loop).

If these two conditions hold, the existence of an infinite computation of the program implies the existence of a corresponding infinite decreasing sequence of elements of the well-founded set, which violates the defining property of the set.

The theorem that expresses the termination of the program with intermittent assertions is

Theorem: if sometime $P(x_0)$ at *start*
then sometime at *finish*.

Because the conclusion of the theorem states only that we eventually reach *finish*, without making any claims about the values of the program variables at that time, it expresses only the termination of the program.

We can prove this theorem by establishing the lemma

Lemma: if sometime $P(x_0)$ and $x = a$ at *more*
then sometime $t(x)$ at *more*.

Of course this lemma implies the theorem, because if we reach *more* and $t(x)$ holds, then we pass directly to *finish*.

To prove the lemma we use complete induction over the same well-founded ordering that we employed to prove termination in the conventional proof. We assume that

sometime $P(x_0)$ and $x = a$ at *more*

and assume inductively that the lemma holds whenever $x = a'$, where $E(a') < E(a)$ and $E(a') \in W$. The proof considers two cases. If $t(a)$ holds, the conclusion of the lemma is already satisfied; otherwise, we pass around the loop, set x to $g(a)$, and return to *more*. Since $E(g(a)) \in W$ by condition (1), and $E(g(a)) < E(a)$ by condition (2), our induction hypothesis applies, to yield that

sometime $t(x)$ at *more*,

and the proof of the lemma is concluded.

With this section we conclude our discussion of how the intermittent-assertion method applies to proving the total correctness of programs. In the next two sections we deal with some less typical applications.

IV. Application: Validity of Transformations That Eliminate Recursion

In discussing the *tips* program (Section II-4) we remarked that part of the difficulty in proving the correctness of the program arose because the program was developed by introducing a stack to remove the recursion from the original definition. It has been argued (e.g. Knuth [1974], Burstall and Darlington [1975]) that, in such cases, we should first prove the correctness of the original recursive program, and then develop the more efficient iterative version by applying one or more *transformations* to the recursive one. These transformations are intended to increase the efficiency of the program (at the expense of clarity) while still satisfying its original specifications.

If we were applying this methodology in producing our *tips* program, therefore, we would first prove the correctness of the recursive version (a trivial task, since that version is completely transparent); we would then develop the iterative *tips* program by systematically transforming the recursive program – removing its recursion and introducing a stack instead. Consequently, the proof we presented in Section II would be completely unnecessary, since the program would have been produced by applying to a correct recursive program a sequence of transformations that are guaranteed not to change that program's specifications.

To realize such a plan, however, we must be certain that the transformations we use actually do produce a program equivalent to the original one; given the same input, the two programs must be guaranteed to return the same output. In other words, we must be certain that bugs cannot be introduced during the transformation process.

In this section we will illustrate how intermittent assertions can be employed to establish that such transformations really do preserve the equivalence of the transformed programs. We will present three transformations to demonstrate this application. Each transformation eliminates recursion in the original program, but each applies in different circumstances and uses a different device. The first transformation requires an extra variable to effect the recursion removal; the other two use a stack, represented as an array and list respectively.

1. Introducing a New Variable

Suppose we are given a recursive program of form

$$F(x) \Leftarrow \textbf{if } p(x) \textbf{ then } f(x) \textbf{ else } h(x\ F(g(x))).$$

If we know that

(1) $h(u\ h(v\ w)) = h(h(u\ v)\ w)$ for every u, v and w
(h is associative), and

(2) $h(e\ u) = u$ for every u
(e is a left identity of h),

then we can transform our program into an equivalent iterative program, of form

```
        input(x)
start:  z ← e
more:   if p(x)
        then  z ← h(z f(x))
              finish: output(z)
        else  (x z) ← (g(x) h(z x))
              goto more.
```

This program uses the additional variable z to accumulate the intermediate results that would be returned by the recursive calls of the original program. The validity of this transformation can be expressed by the following

Theorem: if sometime $x = a$ at *start*
and $F(a)$ is defined
then sometime $z = F(a)$ at *finish*.

The theorem contains the condition that the recursive function F be defined on a, (that the computation of $F(a)$ will terminate) this condition is necessary for, otherwise, the iterative program will never terminate, and therefore control will never reach *finish* at all.

The theorem can be proved using

Lemma: if sometime $x = a$ and $z = b$ at *more*
and $F(a)$ is defined
then sometime $z = h(b\ F(a))$ at *finish*.

The lemma implies the theorem, since if we enter the program with input $x = a$, where $F(a)$ is defined, then we reach *more* with $x = a$ and $z = e$. By the lemma, therefore, we will eventually reach *finish* with $z = h(e\ F(a))$, but then $z = F(a)$ since e is a left identity of h (condition (2)).

To prove the lemma, suppose

sometime $x = a$ and $z = b$ at *more*,

where $F(a)$ is defined. The proof employs complete induction on a, over the *ordering* $\prec$ *induced by the recursive computation* of $F(a)$. (This is the ordering such that $g(d) \prec d$, where d and $g(d)$ are the arguments to two successive calls to F in the computation of $F(a)$. It can be shown to be well-founded, since the recursive computation of $F(a)$ has been assumed to terminate.) We will assume inductively that the lemma holds whenever $x = a'$, for any a' such that $a' \prec a$, and show that it holds when $x=a$ as well.

We distinguish between two cases, depending on the truth value of $p(a)$.

Case $p(a)$ is true: Then $F(a) = f(a)$, by the definition of F. Since $p(a)$ is true, we follow the then branch of the program and arrive at *finish* with $z = h(b\ f(a))$, i.e., $z = h(b\ F(a))$, which is the desired conclusion.

Case $p(a)$ is false: Then $F(a) = h(a\ F(g(a)))$ by the definition of F. Since $p(a)$ is false, we follow the else branch around the loop, so that

sometime $x = g(a)$ and $z = h(b\ a)$ at *more*.

Since $x = g(a)$ and $g(a) \prec a$, we obtain by our induction hypothesis that

sometime $z = h(h(b\ a)\ F(g(a)))$ at *finish*,

or, by the associativity of h (condition (1)),

sometime $z = h(b\ h(a\ F(g(a))))$ at *finish*.

But $F(a) = h(a\ F(g(a)))$, so that we have

sometime $z = h(b\ F(a))$ at *finish*,

which is the desired result.

In the above argument we established that the iterative program will terminate whenever the original recursive program does and that the two programs will then return the same value. It is still conceivable, however, that the iterative program may terminate and return some value even though the recursive program does not. In other words, we have shown that the iterative program computes an *extension* of the function computed by our recursive program, rather that the exact same function.

In fact, the iterative and recursive programs do compute the same function; i.e. they are *equivalent*. To show that the recursive program halts whenever the iterative program does, we need only prove the following additional theorem.

Theorem: if sometime $x = a$ at *start*
and if the computation terminates
then $F(a)$ is defined.

The proof depends on the following

Lemma: if sometime $x = a$ at *more*
and if the computation terminates
then $F(a)$ is defined.

which implies the theorem immediately.

To prove the lemma, suppose that

sometime $x = a$ at *more*,

and that the computation terminates. The proof is by induction on the ordering induced by the computation, which is well-founded since the computation is assumed to terminate. In the case that $p(a)$ holds, the lemma follows because $F(a)$ is then defined to be $f(a)$; if $p(a)$ is false, on the other hand, we pass around the loop and return to *more* with x reset to $g(a)$. Since $g(a) \prec a$ in the ordering induced by the computation, our induction hypothesis implies that $F(g(a))$ is defined. However, $F(a)$ in this case is $h(a\ F(g(a)))$, which is also defined, so the proof is complete.

Now let us see briefly how the corresponding results are obtained for the remaining two transformations.

2. Introducing an array stack

In this and the succeeding section we will omit proofs, because they tend to be repetitive.

Suppose we are given a recursive program of form

$F(x) \Leftarrow$ **if** $p(x)$
then $f(x)$
else $F(h(x\ F(g(x))))$.

For instance, the Ackermann function defined in Section II-3 can be expressed in this form. The program is more complicated than the preceding form because it contains two nested recursive calls.

If h has a left identity e, we can transform the above program into an equivalent iterative program, of form

```
        input(x)
start:  index ← 2
        stack[1] ← e
        stack[2] ← x
more:   if index = 1
        then finish: output (stack[1])
        else if p(stack[index])
             then stack[index-1] ← h(stack[index-1] f(stack[index]))
                  index ← index-1
                  goto more
             else stack[index+1] ← g(stack[index])
                  index ← index+1
                  goto more.
```

Note that this time we have not assumed that h is associative. The two theorems that express the validity of our transformation are

Theorem: if sometime $x = a$ at *start*
and if $F(a)$ is defined
then sometime $stack[1] = F(a)$ at *finish*

and

Theorem: if sometime $x = a$ at *start*
and if the computation terminates
then $F(a)$ is defined.

The first theorem conveys that the iterative program is an extension of the recursive one, while the second theorem states that the recursive program terminates whenever the iterative one does. Together, the two theorems imply the equivalence of the two programs.

The proof of the first theorem depends on

Lemma: if sometime $index = i$, $i \geq 2$, $(stack[1]\ stack[2]\ \ldots\ stack[i-2]) = s$,
$stack[i-1] = a$, and $stack[i] = b$ at *more*,
and $F(b)$ is defined,
then sometime $index = i-1$, $(stack[1]\ stack[2]\ \ldots\ stack[i-2]) = s$
and $stack[i-1] = h(a\ F(b))$ at *more*.

The proof is similar to the total correctness proof of the Ackermann program we introduced in Section II-3.

The proof of the second theorem depends directly upon

Lemma: if sometime *index* = i, $i \geq 2$ and *stack*[i] = b at *more*
and if the computation terminates
then F(b) is defined,

whose proof, in turn, depends on the previous lemma. The reader may find it rewarding to complete this proof on his own.

3. Introducing a list stack

We will indicate here how to prove the validity of a transformation that removes recursion by introducing a stack represented as a list. Suppose we have a recursive program of form

```
F(x) <= if p(x)
        then f(x)
        else h(F(g1(x)) F(g2(x))).
```

Our recursive definition of the *tips* function (section III-4) is of this form, for instance. If we know that h is associative, and that h has a left identity e, we can transform our recursive program into an equivalent iterative program, of form

```
        input(x)
start:  stack ← ()
        z ← e
more:   if p(x)
        then z ← h(z f(x))
             teststack: if stack = ()
                        then finish: output(z)
                        else (x stack) ← (head(stack) tail(stack))
                             goto more
        else (x stack) ← (g1(x) g2(x)·stack)
             goto more.
```

The two theorems that express the validity of this transformation are again

Theorem: if sometime $x = a$ at *start*
and if $F(a)$ is defined
then sometime $z = F(a)$ at *finish*
(the iterative program is an extension of the recursive program),

and

Theorem: if sometime $x = a$ at *start*
and if the computation terminates
then $F(a)$ is defined
(whenever the iterative program terminates, the recursive program terminates).

The proof of the first theorem depends on the following

Lemma: if sometime $x = a$, $z = b$ and *stack* $= s$ at *more*
and if $F(a)$ is defined
then sometime $z = h(b\ F(a))$ and *stack* $= s$ at *teststack*.

The proof of the lemma is an abstract version of the proof we conducted for the *tips* example in Section II-4.

The proof of the second theorem depends on the following

Lemma: if sometime $x = a$ and $z = b$ at *more*
and if the computation terminates
then $F(a)$ is defined,

whose proof, in turn, depends on the previous lemma.

In the next section we will discuss an entirely different application of the intermittent-assertion method.

V. Application: Correctness of Continuously Operating Programs

Conventionally, in proving the correctness of a program, we describe its expected behavior in terms of an output specification, which is intended to hold when the program terminates. Some programs, such as operating systems, airline-reservation systems and management systems, however, are never expected to terminate. Such programs will be said to be *continuously operating* (Francez and Pneuli [1975]). The correctness of continuously operating programs therefore cannot be expressed by output specifications, but rather by their intended behavior while running.

Furthermore, we conventionally describe the internal workings of a program with an invariant assertion, which is intended to hold every time control passes through the corresponding point. The description of the workings of a continuously operating program, however, often involves a relationship that some event A is inevitably followed by some other event B. Such a relationship connects two different states of the program and, generally, cannot be phrased as an invariant assertion.

In other words, the standard tools for proving the correctness of terminating programs, input-output specifications and invariant assertions, are not appropriate for continuously operating programs. The intermittent-assertion method provides a natural complement here, both as a means for specifying the internal and external behavior of these programs, and as a technique for proving the specifications correct.

We will use one very simple example, an imaginary sequential operating system, to illustrate this point:

```
more:  read(requests)
setup: if requests = ()
       then goto more
       else (job requests) ← (head(requests) tail(requests))
            execute: process(job)
            goto setup.
```

At each iteration this program reads a list, *requests*, of jobs to be processed. If *requests* is empty, the program will read a new list, and will repeat this operation indefinitely until a nonempty request list is read. The system will then process the jobs one by one; when they are all processed, the system will again attempt to read a request list.

What we wish to establish about this program is that if a job j is read into the request list, it

will eventually be processed. Although this claim is not representable as an input-output specification, it is directly expressed in the following

Theorem: if sometime $j \in$ *requests* at *setup*
then sometime *job* = j at *execute*.

Here, $j \in$ *requests* means that j belongs to the list of current requests.

To prove the theorem, assume that

sometime $j \in$ *requests* at *setup*.

Then *requests* is not empty and is of the form

$\alpha\, j\, \beta$,

where α and β are the sublists of jobs occuring before and after j, respectively, in the request list. Our proof will be by complete induction on the structure of α: we assume the theorem holds whenever *requests* is of form

$\alpha'\, j\, \beta$,

for any sublist α' of α. The proof distinguishes between two cases

Case $\alpha = ()$: Then $j = head(requests)$. Since $requests \neq ()$, we reach *execute* with $job = head(requests) = j$, satisfying the conclusion of the theorem.

Case $\alpha \neq ()$: Then $\alpha = head(\alpha) \cdot tail(\alpha)$. Because again $requests \neq ()$, we process $job = head(\alpha)$, and return to *setup* with *requests* reset to $tail(\alpha)\, j\, \beta$. Since $tail(\alpha)$ is a sublist of α, we can conclude from our inductive assumption that

sometime *job* = j at *execute*,

as we had hoped.

This program is very simple, but it may serve to suggest how the intermittent-assertion method can be applied to the more realistic examples. It is possible that the method could be extended to concurrent programs as well.

VI. Conclusions

The intermittent-assertion method not only serves as a valuable tool, but also provides a general framework encompassing a wide variety of techniques for the logical analysis of programs. Diverse methods for establishing partial correctness, termination, and equivalence fit easily within this framework. Furthermore, some proofs, naturally expressed with intermittent assertions, are not as easily conveyed by the more conventional methods.

It has yet to be determined whether this approach is suitable for automatic or interactive proof of program correctness. If the lemmas and the well-founded orderings for the induction are provided by the programmer, to construct the remainder of the proof appears to be fairly mechanical. On the other hand, to find appropriate lemmas and the corresponding orderings is as difficult a task as finding the invariant assertions and well-founded orderings for the conventional ways of establishing correctness and termination. But regardless of whether its implementation turns out to be difficult, we believe that the intermittent-assertion method embodies much of the intuitive understanding about the way programs work. Therein lies its principal strength.

Acknowledgements

We would like to thank Rod Burstall and Nachum Dershowitz for many helpful discussions related to this work.

VII. References

Burstall, R.M. [Aug. 1974], *Program proving as hand simulation with a little induction*, Information Processing, 1974, North Holland Publishing Company, Amsterdam, pp. 308-312.

Burstall, R.M. and Darlington, J. [Apr. 1975], *Some transformations for developing recursive programs*, Proceedings of International Conference on Reliable Softwage, Los Angeles, Calif., pp. 465-472.

Floyd, R. W. [1967], *Assigning meaning to programs*, Proceedings of Symposium in Applied Mathematics, V. 19 (J.T. Schwartz, ed), American Mathematical Society, pp. 19-32.

Francez, N. and Pnueli, A. [Nov. 1975], *A proof method for cyclic programs*, Memo, Computer Science Dept., Tel-Aviv University, Tel-Aviv, Israel.

Hoare, C.A.R. [Oct. 1969], *An axiomatic basis of computer programming*, CACM, Vol. 12, No. 10, pp. 576-580, 583.

Katz, S.M. and Manna, Z. [Dec. 1975], *A closer look at termination*, Acta Informatica, Vol. 5, pp. 333-352.

Knuth, D.E. [1968], The Art of Programming, Addison-Wesley Publishers, Reading, Mass.

Knuth, D.E. [Dec. 1974], *Structured programming with goto statements*, Computing Surveys, Vol. 6, No. 4, pp. 261-301.

Manna, Z. [June 1971], *Mathematical theory of partial correctness*, Journal of Computer and System Sciences, Vol. 5, No. 3, pp. 239-253.

Manna, Z. [1974], Mathematical Theory of Computation, McGraw-Hill Book Company, New York, N.Y.

Morris, J.H. and Wegbreit, B. [Feb. 1976], *Subgoal induction*, Memo, Xerox Research Center, Palo Alto, Calif.

Schwarz, J. [Dec. 1975], *Event based reasoning - A system for proving correct termination of programs*, Research Report No. 12, Dept. of Artificial Intellignece, University of Edinburgh, Edinburgh, Scotland.

Topor, R.W. [1976], *A simple proof of the Schorr-Waite garbage collegection algorithm*, **Acta Informatica** (to appear).