Programming Languages

J. J. Horning Editor

Is "Sometime" Sometimes Better than "Always"? Intermittent Assertions in Proving Program Correctness

Zohar Manna
Stanford University and
The Weizmann Institute

Richard Waldinger
SRI International

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

The intermittent-assertion method is presented with a number of examples of correctness and termination proofs. Some of these proofs are markedly simpler than their conventional counterparts. On the other hand, it is shown that a proof of correctness or termination by any of the conventional techniques can be rephrased directly as a proof using intermittent assertions. Finally, it is shown how the intermittent-assertion method can be applied to prove the validity of program transformations and the correctness of continuously operating programs.

Key Words and Phrases: intermittent assertions, correctness of programs, termination of programs, program verification, program transformation, continuously operating programs.

CR Categories: 5.24

This research was supported in part by the Advanced Research Projects Agency under contract MDA903-76-C-0206, by the National Science Foundation under grant DCR72-03737A01, by the Office of Naval Research under contracts N00014-76-C-0687 and N00014-75-C-0816, and by a grant from the United States-Israel Binational Science Foundation.

Authors' addresses: Z. Manna, Artificial Intelligence Lab, Stanford University, Stanford, CA 94305; R. Waldinger, Artificial Intelligence Center, SRI International, Menlo Park, CA 94025.

1. Introduction

The most prevalent approach to proving that a program satisfies a given property has been the *invariant-assertion method*, made known largely through the work of Floyd [4] and Hoare [7]. 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 with 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*, one whose elements are ordered in such a way that no infinite decreasing sequences of elements exist. (The nonnegative integers under the regular greater-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 designated 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 [4].

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

Burstall [2] introduced a method whereby the total correctness of a program can be shown in a single proof. The approach had been applied to specific programs earlier by Knuth [9, Section 2.3.1] and others. This technique 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 use the phrase

sometime Q at L

to denote that Q is an intermittent assertion at label L, i.e. that sometime control will pass through L with assertion Q satisfied. (Similarly, we could use the phrase "always Q at L" to indicate that Q is an invariant assertion at L.) If the entrance of a program is labeled *start* and its exit is labeled *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 program's exit and satisfy the desired output specification.

If we are only interested in 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. (It could have been written as "*sometime true at finish*" because the assertion *true* always holds.)

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. The proofs of the lemmas can often involve *complete induction over a well-founded ordering* (see [13]). 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.

The intermittent-assertion method has begun to attract a good deal of attention. Different approaches to its formalization have been attempted, using predicate calculus [16], Hoare-style axiomatization [18], modal logic [15], and the Lucid formalism [1]. Topor [17] applied the method to proving the correctness of the Schorr-Waite algorithm, a complicated garbage-collecting scheme.

In this paper, we first present and illustrate the intermittent-assertion method with a variety of examples for proving correctness and termination. Some of these proofs are markedly simpler than their conventional counterparts. On the other hand, we prove that the intermittent-assertion method is at least as powerful as the conventional invariant-assertion method and the well-founded ordering method, in addition to the more recent subgoal-assertion method [12, 14] for proving partial correctness. Finally, we show that the intermittent-assertion 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.

2. The Intermittent-Assertion Method: Examples

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.

2.1 Counting the Tips of a Tree

Let us consider a simple program as a vehicle for demonstrating the basic technique. This is an algorithm for counting the *tips* of a complete 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 ← (tree)
        count ← 0
more:   if stack = ()
        then finish: output(count)
        else if head(stack) is a tip
             then count ← count + 1
                  stack ← tail(stack)
                  goto more
```

```
    else first ← head(stack)
         stack ← left(first)·[right(first)·tail(stack)]
         goto more
```

(This program is similar to one used by Burstall in [2].) We have used the notation ( ) to denote the empty list, (x) to denote the list whose sole element is x, and $x \cdot l$ to denote the list formed by adding the element x at the beginning of the list l. (Note that (x) is the same as $x \cdot ( )$.) If the list l is not empty, then *head*(l) is its first element and *tail*(l) is the list of its remaining elements. The indentation of the program indicates that, if *head*(*stack*) is a tip, all three instructions following **then** are to be executed; otherwise all three instructions following **else** are to be executed.

This program initially inserts the given *tree* as the single element of the *stack*. At each iteration, the first element is removed from the *stack*. If it is a tip, the element is counted; otherwise, its left and right subtrees are inserted as the first and second elements of the *stack*. The process terminates when the *stack* is empty; *count* is then the number of tips in the given tree.

Using intermittent assertions, we can express the total correctness of this program by the following theorem:

THEOREM. *If sometime tree = t at start,*
then sometime count = tips(t) at finish.

This theorem states the termination of the program in addition to its partial correctness, because it implies that control must eventually reach the program's exit and satisfy the output specification.

In order to apply the intermittent-assertion method, we supply a lemma to describe the behavior of the program's loop. In this case, the correctness of the program depends on the following property: If we enter the loop with some element t at the head of the *stack*, then eventually the tips of t will be counted and t will be removed from the *stack*. (Note that we may need to return to *more* many times before the *tips* of t are counted.) This property is expressed more precisely by the following lemma:

LEMMA. *If sometime count = c*
and stack = t·s at more
then sometime count = c + tips(t)
stack = s at more.

The hypothesis *count* = c in the antecedent allows us to refer to the original value of *count* in the consequent, even though the value may have changed subsequently.

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

sometime *tree* = t at *start*.

Then, following the computation specified by the program, we set *stack* to (t), *count* to 0, and reach *more*, so that

sometime *count* = 0 and *stack* =(t) = $t \cdot ( )$ at *more*.

The lemma then tells us, taking c to be 0 and s to be ( ), that

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

Because we are at *more* with *stack* = ( ), the computation proceeds to *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 t. In other words, we suppose the antecedent of the lemma, that

sometime *count* = c then *stack* = $t \cdot s$ at *more*,

and we assume inductively that the lemma holds whenever *count* = c' and *stack* = $t' \cdot s'$, where t' is any subtree of t. We then show the consequent of the lemma, that

sometime *count* = c + *tips*(t) and *stack* = s at *more*.

The proof distinguishes between two cases, depending on whether or not t is a tip.

Case t is a tip. Then *tips*(t) = 1 by the recursive definition of *tips*. Since *stack* = $t \cdot s$, it is clearly not empty, but its head t is a tip. The program therefore increases *count* by 1 and removes t from the *stack*. Thus

sometime *count* = c + 1 = c + *tips*(t)
and *stack* = s at *more*,

establishing the conclusion of the lemma in this case.

Case t is not a tip. Then *tips*(t) = *tips*(*left*(t)) + *tips*(*right*(t)), by the recursive definition of *tips*. Since t is not a tip, we pass around the **else** branch of the loop this time: we remove t from the *stack*, break it down into its left and right subtrees, replace these on the *stack* as its first and second elements, and return to *more*. Thus

sometime *count* = c
and *stack* = *left*(t)·[*right*(t)·s] at *more*.

We can then apply the induction hypothesis (taking c' to be c, t' to be *left*(t), and s' to be *right*(t)·s) since *left*(t) is a subtree of t. The induction hypothesis tells us that

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

Since *right*(t) is also a subtree of t, we can apply the induction hypothesis again (taking c' to be c + *tips*(*left*(t)), t' to be *right*(t), and s' to be s), yielding

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

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

sometime *count* = c + *tips*(t) and *stack* = s at *more*.

This is the desired conclusion of the lemma.

Note that once the lemma was formulated and the basis for the induction decided, the proofs proceeded in a fairly mechanical manner. On the other hand, choosing the lemma and the basis for induction required some ingenuity.

The proof of the lemma called upon the full power of the intermittent-assertion method. Although the recursive program that defines the *tips* function can count the tips of a subtree with a single top-level recursive call, the iterative program may require many traversals of the loop before the tips of a subtree are counted. The intermittent-assertion method allows us to relate the point at which we are about to count the tips of a subtree t with the point at which we have completed the counting and to consider the many executions of the body of the loop between these points as a single unit, which corresponds naturally to a single recursive call of $tips(t)$.

The conventional invariant-assertion method, on the other hand, requires that we identify a condition that allows us to relate the situation before and after each single execution of the body of the loop. There may be no natural connection between these two points; consequently our invariant-assertion must be exceptionally complete. In this case, such an assertion is

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

where $\sum_{s \in stack} tips(s)$ is the sum of the *tips* of all the elements of the *stack* (cf. [11]). Once we know this assertion, the invariant-assertion proof is also straightforward. However, to formulate the above assertion, we are required to relate all the elements of the stack, while, to understand the program or to produce the intermittent-assertion proof, we only needed to consider the first element of the stack.

The intermittent-assertion proof established termination at the same time as correctness; to prove termination by the conventional well-founded ordering approach, we can show that the value of the pair

$$(tips(tree) - count \quad tips(head(stack)))$$

always decreases in the lexicographic ordering each time we return to *more*. In other words, either the first component $tips(tree)-count$ is reduced, or the first component remains fixed and the second component $tips(head(stack))$ is reduced. Both components remain nonnegative at all times.

Although finding the above pair requires a bit of ingenuity, this termination proof is relatively straightforward. In the next section, we will see a program for which the simplest known conventional termination proof is significantly more complicated than the intermittent-assertion proof of total correctness.

2.2 Ackermann Function

The Ackermann function, denoted by $A(x\ y)$, is defined recursively for nonnegative integers x and y as

```
A(x y) ⇐ if x = 0
          then y + 1
          else if y = 0
               then A(x−1 1)
               else A(x−1 A(x y−1))
```

For example, $A(1\ 1) = A(0\ A(1\ 0)) = A(0\ A(0\ 1)) = A(0\ 2) = 3$.

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

$$A(4\ 4) = 2^{2^{2^{2^{2^{2^{2}}}}}} - 3.$$

An iterative program to compute the same function is

```
        input (x_0 y_0)
start:  stack[1] ← x_0
        stack[2] ← y_0
        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. If at some stage the recursive program is computing

$$A(s_0\ A(s_1\ \ldots\ A(s_{i-1}\ s_i)\ldots)),$$

then at the corresponding stage of the iterative computation

$$stack = (s_0\ s_1\ \ldots\ s_{i-1}\ s_i) \text{ and } index = i.$$

Using intermittent assertions, we can express the program's total correctness by:

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 employ the following lemma:

LEMMA. *If sometime $index = i$, $i \geq 2$,*
$stack[1{:}i-2] = s$,
$stack[i-1] = a$, and
$stack[i] = b$ at more,
then sometime $index = i-1$,
$stack[1{:}i-2] = s$ and
$stack[i-1] = A(a\ b)$ at more.

Here s represents a tuple of stack elements. The abbreviation $stack[1{:}i-2] = s$ will be used to denote that s equals the tuple of elements $(stack[1]\ stack[2]\ \ldots\ stack[i-2])$; this expression 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, is 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 shall 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:$i-2$] = 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 nonnegative integers; in other words, we assume the lemma holds whenever *stack*[*index*-1] = a' and *stack*[*index*] = b', where a' and b' are any nonnegative integers such that $a' < a$ or such that $a' = a$ and $b' < b$, and show that it then holds when *stack*[*index*-1] = a and *stack*[*index*] = b, i.e.

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

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 we have

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

the inductive hypothesis can be applied (taking a' to be $a - 1$ and b' to be 1) to yield that

sometime *index* = $i-1$,
  stack[1:$i-2$] = 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($a-1$ A(a $b-1$)) by the recursive definition. Since *index* $\neq 1$, *stack*[*index*-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 $b-1$) $<$ (a b),

our induction hypothesis applies (taking a' to be a and b' to be $b-1$), yielding

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

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.

Because *index* = i and

(*stack*[$i-1$] *stack*[i]) = ($a-1$ A(a $b-1$)) $<$ (a b),

we can apply the induction hypothesis once more (taking a' to be $a - 1$ and b' to be A(a $b - 1$)) to obtain that

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

which is the desired conclusion in this case. □

This completes the intermittent-assertion proof of the total correctness of the Ackermann program; we believe it reflects our understanding of the way the program works. The invariant-assertion proof of the partial correctness is quite natural; at each iteration it can be shown that

A(*stack*[1] A(*stack*[2] ...
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).

On the other hand, the known proofs of the termination of this iterative program using the conventional well-founded ordering method are extremely complicated, and we challenge the intrepid reader to construct such a proof.

2.3 Greatest Common Divisor of Two Numbers

In the previous two examples, we have applied the intermittent-assertion method to programs involving only one loop. 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 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 y > x
                       then y ← y − x
                            goto reducey
              goto more
```

This program is motivated by the following properties of the *gcd*:

$gcd(x\ y) = y$ if $x = y$,
$gcd(x\ y) = gcd(x-y\ y)$ if $x > y$, and
$gcd(x\ y) = gcd(x\ y-x)$ if $y > x$.

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

THEOREM.-*If sometime $x = a, y = b$ and $a, b > 0$ at start, then sometime $y = gcd(a\ b)$ at finish.*

To prove this theorem, we need a lemma that describes the internal behavior of the program.

LEMMA. *If sometime $x = a, y = b$, and $a, b > 0$ at more or sometime $x = a, y = b$, and $a, b > 0$ at reducex or sometime $x = a, y = b$, and $a, b > 0$ at reducey, then sometime $y = gcd(a\ b)$ at finish.*

To show that the lemma implies the theorem, we assume that

sometime $x = a$, $y = b$, and $a, b > 0$ at *start*.

Then control passes to *more*, so that

sometime $x = a$, $y = b$, and $a, b > 0$ at *more*.

But then the lemma implies that

sometime $y = gcd(a\ b)$ at *finish*,

which is the desired conclusion of the theorem.

It remains to prove the lemma. We suppose

sometime $x = a$, $y = b$, and $a, b > 0$ at *more*
or sometime $x = a$, $y = b$, and $a, b > 0$ at *reducex*
or sometime $x = a$, $y = b$, and $a, b > 0$ at *reducey*.

The proof proceeds by induction on $a + b$; we assume inductively that the lemma holds whenever $x = a'$ and $y = b'$, where $a' + b' < a + b$, and show that

sometime $y = gcd(a\ b)$ at *finish*.

We must distinguish between three cases.

Case $a = b$. Regardless of whether control is at *more, reducex,* or *reducey,* control passes to *finish* with $y = b$, so that

sometime $y = b$ at *finish*.

But in this case $b = gcd(a\ b)$, by a given property of the *gcd* function, so we have

sometime $y = gcd(a\ b)$ at *finish*,

which is the desired conclusion of the lemma.

Case $a > b$. Regardless of whether control is at *more*, *reducex*, or *reducey*, control reaches *reducex* and passes around the top inner loop, resetting x to $a - b$, so that

sometime $x = a-b$ and $y = b$ at *reducex*.

For simplicity, let us denote $a-b$ and b by a' and b', respectively. Note that

$a', b' > 0$
$a + b > a' + b'$, and
$gcd(a'\ b') = gcd(a-b\ b) = gcd(a\ b)$.

This last condition follows by a given property of the *gcd*.

Because a', $b' > 0$ and $a + b > a' + b'$, the induction hypothesis implies that

sometime $y = gcd(a'\ b')$ at *finish*;

i.e., by the third condition above,

sometime $y = gcd(a\ b)$ at *finish*.

Case $b > a$. This case is disposed of in a manner symmetric to the previous case.

This concludes the proof of the lemma. The total correctness of the program is thus established. □

It is not difficult to prove the partial correctness of the above program by 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 $gcd(x\ y) = gcd(x_0\ y_0)$

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 nonnegative integers ordered by the regular lexicographic ordering. 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 [8]. Although this method is effective, it is not the most natural in establishing the termination of the *gcd* program.

3. 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 have seen examples of intermittent-assertion proofs of correctness and termination that are simpler than any known conventional counterparts. In this section we show that the reverse is never the case; in fact, we can directly rephrase any partial-correctness proof using the invariant-assertion method as an intermittent-assertion proof. The same result applies to another standard partial-correctness proof technique, the "subgoal-assertion method." Furthermore, we will show that any termination proof using the well-founded ordering method can also be expressed using intermittent assertions instead. Therefore we can always use the intermittent-assertion method in place of the established techniques.

To characterize the conventional techniques precisely, we find it convenient to introduce some new notations, which are described more fully in [13]. Let x be a complete list of the variables of a given program, and let x_0 denote their initial values. Suppose that we have designated a special set of labels $L_0, L_1, \ldots, L_h$, where L_0 and L_h are the program's entrance (*start*) and exit (*finish*), respectively. It is assumed that each of the program's loops passes through at least one of the designated labels. A path between two designated labels is said to be *basic* if it does not pass through any designated label (except at its endpoints). For each basic path α from label L_i to L_j, we let $t_\alpha(x)$ denote the condition that must hold for control to pass from L_i along path α to L_j, and we let $g_\alpha(x)$ be the transformation of the values of x effected in traversing the path α. Thus, if $x = a$ at L_i and condition $t_\alpha(a)$ holds, then control will pass along path α, reaching L_j with $x = g_\alpha(a)$.

We now define the ordering that will enable us to mimic conventional partial-correctness proofs by the intermittent-assertion method. Suppose that the program is intended to apply to inputs satisfying the input specification $P(x_0)$. Then the *ordering* $>$ *induced by the computation* is defined as follows:

$(a\ i) > (b\ j)$

if control passes through L_i with $x = a$ and then eventually passes through L_j with $x = b$ for some computation that initially satisfies the input specification $P(x_0)$ and that ultimately terminates. This ordering is well-founded because any infinite decreasing sequence in the ordering would correspond to an infinite computation of the program, but we have only defined the ordering for finite (terminating) computations.

Now let us see how the concepts we have introduced allow us to rephrase an invariant-assertion proof of the partial correctness of a program as an intermittent-assertion proof.

3.1 Invariant-Assertion Method

Suppose that we have used the invariant-assertion technique to prove that a program is partially correct with respect to some input specification $P(x_0)$ and output specification $R(x_0\ x)$. Then we have a set of invariant assertions $Q_0(x_0\ x), Q_1(x_0\ x), \ldots, Q_h(x_0\ x)$ corresponding to the designated labels $L_0, L_1, \ldots, L_h$, for which we have proved that, for every x_0 and x,

(1) $P(x_0) \Rightarrow Q_0(x_0\ x_0)$
(the input specification implies the initial invariant assertion) and
(2) $Q_h(x_0\ x) \Rightarrow R(x_0\ x)$
(the final invariant assertion implies the output specification),

and, for each basic path α from L_i to L_j, we have proved the *verification condition*

(3_α) $Q_i(x_0\ x)$ and $t_\alpha(x) \Rightarrow Q_j(x_0\ g_\alpha(x))$
(the invariant assertion before the path implies the invariant assertion after).

Conditions (1) and (3_α) establish that each $Q_i(x_0\ x)$ is indeed an *invariant assertion* at L_i; it has the property that, each time we pass through L_i, $Q_i(x_0\ x)$ will be true for the current value of x. Condition (2) then implies that, if the program terminates, the desired output specification will be satisfied. Together, these conditions establish the partial correctness of our program.

From the given proof of the partial correctness of the program, we can extract an intermittent-assertion proof of the same result. The theorem that expresses the partial correctness in the intermittent-assertion notation is as follows:

THEOREM. *If sometime $x = x_0$ and $P(x_0)$ at start*
and the computation terminates,
then sometime $R(x_0\ x)$ at finish.

This theorem expresses the partial correctness of the program because it includes the explicit assumption that the particular computation being considered terminates. Given the assertions $Q_i(x_0\ x)$ from the invariant-assertion proof, we can construct the following lemma, which will enable us to prove the partial-correctness theorem:

LEMMA. *For every i, $0 \le i \le h$,*
if sometime $x = a$, $P(x_0)$ and $Q_i(x_0\ a)$ at L_i
and the computation terminates,
then sometime $R(x_0\ x)$ at finish.

To prove that the lemma implies the theorem, assume

sometime $x = x_0$ and $P(x_0)$ at *start*
and the computation terminates.

Our invariant-assertion proof includes a proof of (1), that $P(x_0) \Rightarrow Q_0(x_0\ x_0)$. That proof can be incorporated here, to yield

sometime $x = x_0$, $P(x_0)$ and $Q_0(x_0\ x_0)$ at L_0
and the computation terminates

(because L_0 is identical to *start*). Taking $i = 0$ in the lemma, we may deduce

sometime $R(x_0\ x)$ at *finish*,

which is the desired conclusion of the theorem.

To prove the lemma, we suppose

sometime $x = a$, $P(x_0)$ and $Q_i(x_0\ a)$ at L_i
and the computation terminates,

for some i between 0 and h. The proof is by induction on the ordering $>$ induced by the computation. Thus we assume inductively that the lemma holds whenever $x = a'$ at $L_{i'}$, where $(a\ i) > (a'\ i')$. The proof distinguishes between two cases.

If $i = h$, we have supposed that

sometime $x = a$ and $Q_h(x_0\ a)$ at L_h.

Incorporating the proof of (2) and recalling that L_h is *finish*, we have

sometime $R(x_0\ x)$ at *finish*,

which is the desired conclusion of the lemma.

On the other hand, if $0 \le i < h$, control must follow some basic path α to a designated label L_j. For this path, $t_\alpha(a)$ must be true, and $x = g_\alpha(a)$ when control reaches L_j. Because $Q_i(x_0\ a)$ and $t_\alpha(a)$ are true, we can reproduce the proof of (3_α) to deduce that $Q_j(x_0\ g_\alpha(a))$ is true. Thus

sometime $x = g_\alpha(a)$ and $Q_j(x_0\ g_\alpha(a))$ at L_j.

Because x_0 has been assumed to satisfy the input specification $P(x_0)$ and because the computation has been assumed to terminate, we have that

$(a\ i) > (g_\alpha(a)\ j)$,

by the definition of the ordering induced by the computation, and therefore that

sometime $R(x_0\ x)$ at *finish*,

by our induction hypothesis. This completes the proof of the lemma.

We have thus constructed an intermittent-assertion proof of the partial correctness of the program, assuming that we were given an invariant-assertion proof. In the next section we indicate how the same procedure can be applied to subgoal-assertion proofs.

3.2. Subgoal-Assertion 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-assertion method*, relates these variables to their ultimate values when the program halts. We 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.

Suppose now that we have used the subgoal-assertion method to prove that a program is partially correct with respect to some input specification $P(x_0)$ and output specification $R(x_0\ x)$. Then we have a set of subgoal assertions $Q_0^*(x\ x_h)$, $Q_1^*(x\ x_h)$, ... , $Q_h^*(x\ x_h)$, corresponding to the designated labels $L_0, L_1, \ldots, L_h$, with the intuitive meaning that $Q_i^*(x\ x_h)$ must hold for the current value of x as control passes through L_i and the ultimate value x_h of x when the computation halts. For these assertions we have proved that for every x_0, x, and x_h:

(1*) $Q_h^*(x_h\ x_h)$,
(the final subgoal assertion always holds for the final value of x) and

(2*) $P(x_0)$ and $Q_0^*(x_0\ x_h) \Rightarrow R(x_0\ x_h)$
(the input specification and the initial subgoal assertion imply the output specification),

and, for each basic path α from L_i to L_j, we have proved the verification condition

(3_α^*) $Q_j^*(g_\alpha(x)\ x_h)$ and $t_\alpha(x) \Rightarrow Q_i^*(x\ x_h)$,
(the subgoal assertion after the path implies the subgoal assertion before).

The subgoal-assertion method works backward through the computation, whereas the invariant-assertion method works forward. Condition (1*) implies that the final subgoal assertion always holds. Conditions (3_α^*) say that, if the appropriate subgoal assertion holds when control reaches the end of a path, then the corresponding subgoal assertion holds when control is at the beginning of the path. If the program does terminate, conditions (1*) and (3_α^*) imply that each $Q_i^*(x\ x_h)$ is indeed a *subgoal assertion* at L_i; it has the property that, each time we pass through L_i, $Q_i^*(x\ x_h)$, will be true for the current value of the program's variables x and its ultimate value x_h. Condition (2*) then implies that, if the program terminates, the desired output specification will be satisfied. Together, these conditions imply the partial correctness of the given program.

To contrast the invariant-assertion and the subgoal-assertion methods, let us consider a simple program to compute the *gcd*:

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

Here $rem(y\ x)$ is the remainder of dividing y by x. The notation $(x\ y) \leftarrow (rem(y\ x)\ x)$ means that the values of x and y are simultaneously assigned to be $rem(y\ x)$ and x, respectively.

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

$R(x_0\ y_0\ y)$: $y = gcd(x_0\ y_0)$,

we can employ the invariant assertions

$Q_{start}(x_0\ y_0\ x\ y)$: $x_0 > 0$ and $y_0 > 0$
(i.e., $P(x_0\ y_0)$)
$Q_{more}(x_0\ y_0\ x\ y)$: $x \geq 0$ and $y > 0$ and
$gcd(x\ y) = gcd(x_0\ y_0)$
$Q_{finish}(x_0\ y_0\ x\ y)$: $y = gcd(x_0\ y_0)$
(i.e., $R(x_0\ y_0\ y)$).

On the other hand, to prove the same result by the subgoal-assertion method, we can use the subgoal assertions

$Q^*_{start}(x\ y\ y_h)$: $x \geq 0$ and $y > 0 \Rightarrow y_h = gcd(x\ y)$,
$Q^*_{more}(x\ y\ y_h)$: $x \geq 0$ and $y > 0 \Rightarrow y_h = gcd(x\ y)$,
$Q^*_{finish}(x\ y\ y_h)$: $y = y_h$.

The reader may observe that the invariant assertions relate the program variables x and y with their initial values x_0 and y_0 and the subgoal assertions relate the program variables with the ultimate value y_h of y.

Let us return to the general case. From a given subgoal-assertion proof of the partial correctness of a program, we can mechanically paraphrase the argument as an intermittent-assertion proof, just as we did for the invariant-assertion method. The theorem that expresses the partial correctness of the program is again:

THEOREM. *If sometime $x = x_0$ and $P(x_0)$ at start and the computation terminates, then sometime $R(x_0\ x)$ at finish.*

The lemma that we shall use in proving the theorem, however, is different from the lemma in the invariant-assertion case:

LEMMA. *For every i, $0 \leq i \leq h$, if sometime $x = a$ and $P(x_0)$ at L_i and the computation terminates, then sometime $Q_i^*(a\ x)$ at finish.*

To construct a proof that the lemma implies the theorem, we take $i = 0$ in the Lemma, and employ in the new proof the justification for Condition (2*) from the given subgoal-assertion proof.

The proof of the lemma is constructed in a way analogous to the earlier invariant-assertion case. Induction is again based on the ordering $>$ induced by the computation. When $i = h$ we use the proof of Condition (1*), and if $0 \leq i < h$ we use the inductive hypothesis and the proof of (3^*_α). □

The results of this section could actually have been proved in another way: The subgoal-assertion method is known (see [14]) to be equivalent in power to the invariant-assertion method, in the sense that any proof by either method can be transformed directly into a proof by the other method. We also know that any invariant-assertion proof can be transformed into an intermittent-assertion proof of the same result, by the process given in the preceeding section. Therefore, any subgoal-assertion proof can be transformed into an intermittent-assertion proof, by first transforming it into an invariant-assertion proof, and then into an intermittent-assertion proof.

We have remarked that the invariant-assertion method relates the current values of the program variables to their initial values, whereas the subgoal-assertion 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.

3.3. 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 ordering approach.

Suppose that we have used the well-founded ordering approach to prove the termination of a given program with respect to some input specification $P(x_0)$. Then we have found a well-founded ordering $>$ over a set W, and, for some set of designated labels $L_0, L_1, \ldots, L_h$, we have found a set of invariant assertions $Q_0(x_0\ x), Q_1(x_0\ x), \ldots, Q_h(x_0\ x)$ and a set of expressions $E_0(x_0\ x), E_1(x_0\ x), \ldots, E_h(x_0\ x)$ for which we have proved the following conditions for every x_0 and x:

(1) $P(x_0) \Rightarrow Q_0(x_0\ x_0)$
(the input specification implies the initial invariant assertion),

(2_α) $Q_i(x_0\ x)$ and $t_\alpha(x) \Rightarrow Q_j(x_0\ g_\alpha(x))$ for every basic path α from L_i to L_j
(the invariant assertion before the path implies the invariant assertion after),

(3_i) $Q_i(x_0\ x) \Rightarrow E_i(x_0\ x) \in W$ for each label L_i
(the value of the expression belongs to W when control passes through L_i), and

(4_α) $Q_i(x_0\ x)$ and $t_\alpha(x) \Rightarrow E_i(x_0\ x) > E_j(x_0\ g_\alpha(x))$ for every basic path α from L_i to L_j
(as control passes from L_i to L_j, the value of the corresponding expression is reduced).

The above conditions establish the termination of the program. Conditions (1) and (2_α) ensure that each $Q_i(x_0\ x)$ is indeed an invariant assertion at L_i: Whenever control passes through L_i, assertion $Q_i(x_0\ x)$ is true for the current value of x. Condition (3) then tells us that each time control passes through L_i the value of the expression $E_i(x_0\ x)$ belongs to W.

Now suppose that Conditions (1)-(4) are satisfied but the program does not terminate for some input x_0 satisfying the input specification $P(x_0)$. Control then passes through an infinite sequence of designated labels; the values of the corresponding expressions $E_i(x_0\ x)$ constitute an infinite sequence of elements of W. Condition (4) then implies that this is a decreasing sequence under the well-founded ordering, thereby contradicting the definition of a well-founded set. Conditions (1)-(4) therefore suffice to establish the termination of the given program.

It is our task to transform a proof by the above method into an intermittent-assertion proof of the termination of the program. The following theorem expresses the desired property:

THEOREM. *If sometime* $x = x_0$ *and* $P(x_0)$ *at start then sometime at finish.*

Recall that "*sometime at finish*" expresses the termination of the program in the intermittent-assertion notation. We can prove this theorem by establishing the following lemma:

LEMMA. *For every* i, $0 \leq i \leq h$,
if sometime $x = a$ *and* $Q_i(x_0\ a)$ *at* L_i,
then sometime at finish.

To construct a proof that the lemma implies the theorem, we take i to be 0 in the lemma and incorporate the given proof of Condition (1) into the intermittent-assertion proof of the theorem.

To prove the lemma we use induction over the same well-founded ordering $>$ that we employed in the given termination proof. Suppose that

sometime $x = a$ and $Q_i(x_0\ a)$ at L_i

for some designated label L_i. We assume inductively that the lemma holds whenever $x = a'$ and $Q_{i'}(x_0\ a')$ at $L_{i'}$, where $E_i(x_0\ a) > E_{i'}(x_0\ a')$. If $i = h$, termination has already occurred. Otherwise control must follow some path α from L_i to L_j, i.e., $t_\alpha(a)$ is true. Thus

sometime $x = g_\alpha(a)$ at L_j.

Because both $Q_i(x_0\ a)$ and $t_\alpha(a)$ hold, the proof of Condition (2) enables us to deduce $Q_j(x_0\ g_\alpha(a))$. The proof of Condition (3) can be incorporated to yield

$E_i(x_0\ a) \in W$ and $E_j(x_0\ g_\alpha(a)) \in W$,

because both $Q_i(x_0\ a)$ and $Q_j(x_0\ g_\alpha(a))$ are true. By Condition (4) then, we have

$E_i(x_0\ a) > E_j(x_0\ g_\alpha(a))$.

We can now use the induction hypothesis, with $i' = j$ and $a' = g_\alpha(a)$, yielding the desired conclusion

sometime at *finish*. □

In this section we have shown how proofs by the conventional methods for establishing partial correctness and termination of programs may be translated into intermittent-assertion proofs of the same results. The translation process is purely mechanical and does not increase the complexity of the proof. For this reason we can conclude that in employing the intermittent-assertion method we have not lost any of the power of the existing methods.

Is it possible that a similar translation could be performed in the other direction? For example, couldn't we devise a procedure for translating any partial-correctness proof by the intermittent-assertion method into a conventional invariant-assertion proof of comparable complexity? We believe not. We have seen no invariant-assertion proof for the *tips* program that does not require consideration of the sum of the tips of *all* the elements in the stack. We have seen no termination proof of the iterative Ackermann program by the conventional method that employs such a simple well-founded ordering as the intermittent-assertion proof. Without formulating a precise notion of the "complexity" of a proof, we cannot argue rigorously that the intermittent-assertion method is strictly more powerful than the conventional methods, but we maintain that this is so.

4. Application: Validity of Transformations That Eliminate Recursion

In discussing the *tips* program (Section 2.1), 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. [3, 6, 10]) 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 possible expense of clarity) while still maintaining its correctness.

If we were applying this method 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 2 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 are *valid*, i.e. that they 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 illustrate how intermittent assertions can be employed to establish the validity of such transformations. We present the intermittent-assertion proof of the validity of a transformation that removes a recursion by introducing a stack. This transformation could have been used to produce our iterative *tips* program from its recursive definition.

Suppose we have a recursive program of form

```
F (x) ⇐ if p(x)
        then f(x)
        else h(F(g_1(x)) F(g_2(x))).
```

(For simplicity, let us assume that p, f, g_1, g_2, and h are defined for all arguments.) 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 the form

```
        input(x)
start:  stack ← (x)
        z ← e
more:   if stack = ()
        then finish: output(z)
        else if p(head(stack))
             then z ← h(z f(head(stack)))
                  stack ← tail(stack)
                  goto more
             else first ← head(stack)
                  stack ← g_1(first)·[g_2(first)·tail(stack)]
                  goto more
```

The validity of this transformation is expressed by the following two theorems:

THEOREM 1. *If sometime* $x = a$ *at start and* F(a) *is defined, then sometime* $z =$ F(a) *at finish.*

THEOREM 2. *If sometime* $x = a$ *at start and the iterative computation terminates, then* F(a) *is defined.*

Theorem 1 contains the condition that F(a) is defined (that the recursive computation of F with input a will terminate). This condition is necessary for otherwise the iterative program will not terminate, and therefore control will never reach *finish* at all. If we succeed in proving Theorem 1, we shall have established that the iterative program terminates whenever the original recursive program does and returns the same output; in other words, the iterative program computes an *extension* of the function computed by the recursive program rather than the exact same function. Theorem 2 shows that the recursive program halts whenever the iterative program does. Together, Theorems 1 and 2 imply that the recursive and iterative programs are equivalent. The proof of Theorem 1 is analogous to the proof of the total correctness of the *tips* program; it requires the following lemma:

LEMMA 1. *If sometime* $z = c$ *and stack* $= a \cdot s$ *at more and* F(a) *is defined, then sometime* $z = h(c$ F$(a))$ *and stack* $= s$ *at more.*

To show that the lemma implies Theorem 1, assume

sometime $x = a$ at *start*

and that F(a) is defined. Then immediately control passes to *more*; so

sometime $z = e$ and *stack* $= (a) = a \cdot ()$ at *more*.

By the lemma (taking c to be e and s to be ()), we have

sometime $z = h(e$ F$(a))$ and *stack* $= ()$ at *more*.

But $h(e$ F$(a)) =$ F(a) by Property (2), that e is a left identity of h. Because *stack* is (), control passes to *finish*, and we deduce

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

which is the desired conclusion of the theorem.

To prove the lemma, suppose

sometime $z = c$ and *stack* $= a \cdot s$ at *more*,

where F(a) is defined. The proof employs complete induction on a over the *ordering* $>$ *induced by the recursive computation*. This is the ordering such that

$d > d'$,

where F(d') is called recursively during the computation of F(d) and where the computation of F(d) terminates. In particular, if F(d) is defined, $d > g_1(d)$ and $d > g_2(d)$. This ordering $>$ is well-founded because an infinite decreasing sequence in the ordering would correspond to an infinite, nonterminating computation of the recursive program, but the ordering has only been defined for finite (terminating) computations.

We shall assume inductively that the lemma holds whenever $z = c'$ and *stack* $= a' \cdot s'$, where $a > a'$ in the ordering $>$ induced by the recursive computation, and show that it holds when $z = c$ and *stack* $= a \cdot s$ as well. We distinguish between two cases, depending on the truth of $p(a)$.

Case $p(a)$ *is true*. Then F(a) $= f(a)$, by the recursive definition of F. Because a is at the head of the *stack*, the *stack* is not empty and p(*head*(*stack*)) is true; therefore we follow the **then** branch of the program; so

sometime $z = h(c\ f(a))$ and *stack* $= s$ at *more*.

But $f(a) =$ F(a); so we have

sometime $z = h(c$ F$(a))$ and *stack* $= s$ at *more*,

which is the desired conclusion.

Case $p(a)$ *is false*. Here F(a) $= h($F$(g_1(a))$ F$(g_2(a)))$, by the recursive definition of F. Note that F(a) is defined; therefore F($g_1(a)$) and F($g_2(a)$) are also defined. Because *stack* is not empty and $p(head(stack))$ is false, control follows the **else** branch of the loop body; so

sometime $z = c$ and $stack = g_1(a) \cdot [g_2(a) \cdot s]$ at *more*.

Recall that $a > g_1(a)$ because we have assumed that $F(a)$ is defined; therefore we can apply the induction hypothesis (taking c' to be c, a' to be $g_1(a)$, and s' to be $g_2(a) \cdot s$) to obtain

sometime $z = h(c\ F(g_1(a)))$
and $stack = g_2(a) \cdot s$ at *more*.

Because $a > g_2(a)$, we can apply the induction hypothesis a second time (taking c' to be $h(c\ F(g_1(a)))$, a' to be $g_2(a)$, and $s' = s$). We derive

sometime $z = h(h(c\ F(g_1(a)))\ F(g_2(a)))$
and $stack = s$ at *more*.

By the associativity of h (Property (1)) and the recursive definition of F, we have

$h(h(c\ F(g_1(a)))\ F(g_2(a))) =$
$h(c\ h(F(g_1(a))\ F(g_2(a)))) = h(c\ F(a))$.

Therefore we can conclude that

sometime $z = h(c\ F(a))$ and $stack = s$ at *more*,

completing the proof of the lemma. □

So far we have only established Theorem 1, that the function computed by the iterative program is an extension of the function computed by the recursive program. We still need to prove Theorem 2, that, if the iterative program terminates, then the recursive program also terminates. This proof depends on another lemma.

LEMMA 2. *If sometime* $z = c$
and stack $= a \cdot s$ *at more*
and the iterative computation terminates,
then $F(a)$ *is defined.*

Lemma 2 implies Theorem 2 directly because the *stack* is initialized to $(a) = a \cdot ()$.

The proof of the lemma employs induction over the ordering $>$ induced by the iterative computation. In this ordering, $(c_1\ s_1) > (c_2\ s_2)$, where c_1 and c_2 are successive values of the variable z at *more* and s_1 and s_2 are successive values of the *stack* at *more* during a terminating computation of the iterative program.

To prove the lemma, suppose that

sometime $z = c$ and $stack = a \cdot s$ at *more*

and that the iterative computation terminates. We assume inductively that the lemma holds whenever $z = c'$ and $stack = a' \cdot s'$, where $(c\ a \cdot s) > (c'\ a' \cdot s')$ in the ordering induced by the computation, and show that $F(a)$ is then defined. We distinguish between two cases.

Case $p(a)$ is true. Here $F(a) = f(a)$ by the recursive program, and therefore $F(a)$ is defined.

Case $p(a)$ is false. Here $F(a) = h(F(g_1(a))\ F(g_2(a)))$, by the recursive program. Since *stack* is not empty and $p(head(stack))$ is false, the iterative computation follows the **else** branch; so

sometime $z = c$ and $stack = g_1(a) \cdot [g_2(a) \cdot s]$ at *more*.

Because the computation was assumed to terminate, we have that

$(c\ a \cdot s) > (c\ g_1(a) \cdot [g_2(a) \cdot s])$,

and therefore, by our induction hypothesis, that $F(g_1(a))$ is defined.

By Lemma 1, we have that

sometime $z = h(c\ F(g_1(a)))$
and $stack = g_2(a) \cdot s$ at *more*.

Again, by the induction hypothesis, we have that $F(g_2(a))$ is defined. Because both $F(g_1(a))$ and $F(g_2(a))$ are defined and $F(a) = h(F(g_1(a))\ F(g_2(a)))$, we can conclude that $F(a)$ is defined. □

We have just shown the validity of the transformation that was actually used to produce the iterative *tips* program in Section 2.1. As in that section, we could have used the conventional invariant-assertion technique in the proof of Theorem 1. However, although we could employ the standard $\sum$ notation to denote repeated applications of the $+$ operation in the *tips* invariant assertion, we would have had to invent a new notation to denote repeated application of the function h in the invariant assertion for the iterative program here.

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

5. 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 information systems, however, are never expected to terminate. Such programs will be said to be *continuously operating* (see, for example, [5]). 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 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 occurring 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 more realistic examples.

Note that, when we make a statement of form

if sometime P at L_1,
then sometime Q at L_2,

we do not necessarily imply that condition Q is satisfied at L_2 *after* condition P is satisfied at L_1; in fact, condition Q could hold *before* condition P. Thus, in the above example, we should be perfectly content if some especially fast operating system were able to process the job before it was submitted. In fact, the proof techniques that we have used in this paper will only allow us to prove an implication of the above form if Q holds at L_2 after P holds at L_1. Additional techniques would be necessary if we wanted to prove such an implication if Q actually holds before P.

Throughout this paper, in proving an implication of the above form, we have tacitly assumed that conditions P and Q are satisfied at different stages of the *same* computation. It is possible to relax this assumption and relate different computations by extending our notation appropriately. We believe one could then apply the intermittent-assertion method to prove properties of nondeterministic and concurrent programs as well.

6. 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 which phases of the intermittent-assertion proof process will be amenable to implementation in verification systems. If the lemmas and the well-founded orderings for the induction are provided by the programmer, constructing the remainder of the proof appears to be fairly mechanical. On the other hand, to find the appropriate lemmas and the corresponding orderings may require some ingenuity. We believe that the intermittent-assertion method will have practical impact because it allows us to incorporate our intuitive understanding about the way a program works directly into a proof of its correctness.

Acknowledgments. We would like to thank Rod Burstall and Nachum Dershowitz for many helpful discussions related to this work. We would also like to thank Ed Ashcroft, Edsger Dijkstra, Jim King and Wolfgang Polak for their careful critical reading of the manuscript.

Received June 1976; revised May 1977

References
1. Ashcroft, E.A., and Wadge, W.W. Intermittent-assertion proofs in Lucid. Information Processing 77, North-Holland Pub. Co., Amsterdam, 1977, pp. 723-726.
2. Burstall, R.M. Program proving as hand simulation with a little induction. Information Processing 74, North-Holland Pub. Co., Amsterdam, 1974, pp. 308-312.
3. Burstall, R.M., and Darlington, J. A transformation system for developing recursive programs. *J. ACM, 24*, 1 (Jan. 1977), 44-67.
4. Floyd, R.W. Assigning meaning to programs. Proc. Symp. in Applied Math. Vol. 19, J.T. Schwartz, Ed., Amer. Math. Soc., Providence, R.I., 1967, pp. 19-32.
5. Francez, N., and Pnueli, A. A proof method for cyclic programs. To appear in *Acta Informatica*.
6. Gerhart, S.L. Correctness-preserving program transformations. Second Symp. on Principles of Programming Languages, Palo Alto, Calif., Jan. 1975, pp. 54-65.
7. Hoare, C.A.R. An axiomatic basis of computer programming. *Comm. ACM 12*, 10 (Oct. 1969), 576-580, 583.
8. Katz, S.M., and Manna, Z. A closer look at termination. *Acta Informatica 5* (Dec. 1975), 333-352.
9. Knuth, D.E. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. Addison-Wesley Reading, Mass., 1968.
10. Knuth, D.E. Structured programming with goto statements. *Computing Surveys 6*, 4 (Dec. 1974), 261-301.
11. London, R.L. A view of program verification. Proc. Conf. on Reliable Software, Los Angeles, Calif., April 1975, 534-545.
12. Manna, Z. Mathematical theory of partial correctness. *J. Comptr. Syst. Sci. 5*, 3 (June 1971), 239-253.
13. Manna, Z. *Mathematical Theory of Computation*. McGraw-Hill, New York, 1974.
14. Morris, J.H., and Wegbreit, B. Subgoal induction. *Comm. ACM 20*, 4 (April 1977), 209-222.
15. Pratt, V.R. Semantical considerations on Floyd-Hoare logic. Proc. 17th Symp. on Foundations of Comptr. Sci., Houston, Tex., Oct. 1976, pp. 109-121.
16. Schwarz, J. Event-based reasoning—a system for proving correct termination of programs. Proc. Third Int. Colloquium on Automata, Languages and Programming, Edinburgh, Scotland, July 1976, pp. 131-146.
17. Topor, R.W. A simple proof of the Schorr-Waite garbage collection algorithm. To appear in *Acta Informatica*.
18. Wang, A. An axiomatic basis for proving total correctness of goto-programs. *BIT 16* (1976), 88-102.

Graphics and Image Processing — James D. Foley, Editor

Some New Methods of Detecting Step Edges in Digital Pictures

Bruce J. Schachter and Azriel Rosenfeld
University of Maryland

This note describes two operators that respond to step edges, but not to ramps. The first is similar to the digital Laplacian, but uses the max, rather than the sum, of the x and y second differences. The second uses the difference between the mean and median gray levels in a neighborhood. The outputs obtained from these operators applied to a set of test pictures are compared with each other and with the standard digital Laplacian and gradient. A third operator, which uses the distance between the center and centroid of a neighborhood as an edge value, is also briefly considered; it turns out to be equivalent to one of the standard digital approximations to the gradient.

Key Words and Phrases: image processing, pattern recognition, edge detection
CR Category: 3.63

The support of the Directorate of Mathematical and Information Sciences, U.S. Air Force Office of Scientific Research, under Contract F44620-72C-0062, is gratefully acknowledged, as is the help of Shelly Rowe in preparing this paper. Authors' address: Computer Science Center, University of Maryland, College Park, MD 20712.
Copyright © 1978, ACM
© 1978 ACM 0001-0782/78/0200-0172 $00.75