Generalizing Specifications for Uniformly Implemented Loops

DOUGLAS D. DUNLOP and VICTOR R. BASILI
University of Maryland

The problem of generalizing functional specifications for **while** loops is considered. This problem occurs frequently when trying to verify that an initialized loop satisfies some functional specification, i.e., produces outputs which are some function of the program inputs.

The notion of a valid generalization of a loop specification is defined. A particularly simple valid generalization, a base generalization, is discussed. A property of many commonly occurring **while** loops, that of being uniformly implemented, is defined. A technique is presented which exploits this property in order to systematically achieve a valid generalization of the loop specification. Two classes of uniformly implemented loops that are particularly susceptible to this form of analysis are defined and discussed. The use of the proposed technique is illustrated with a number of applications. Finally, an implication of the concept of uniform loop implementation for the validation of the obtained generalization is explained.

Categories and Subject Descriptors: D.2.4 [**Sofware Engineering**]: Program Verification—*assertion checkers, correctness proofs, validation;* F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*invariants*

General Terms: Languages, Theory, Verification

Additional Key Words and Phrases: Generalizing specifications, uniformly implemented loops, iteration condition

1. INTRODUCTION

Consider the problem of proving/disproving a **while** loop correct with respect to some functional specification, f, i.e., f requires the output variable (or variables) to be some function of the inputs to the loop. If the loop precondition is weak enough so that the domain of f contains the intermediate states which appear after each loop iteration (i.e., if the loop precondition is a loop invariant), the loop is said to be *closed* for the domain of f. An important result in program verification is that if the loop is closed for the domain of its specification, there are two easily constructed verification conditions based solely on the specification, loop predicate, and loop body which are necessary and sufficient conditions for the correctness of the loop (assuming termination) with respect to its specification [9, 10]. If the loop is not closed for the domain of the specification

This work was supported in part by the Air Force Office of Scientific Research Contract AFOSR-F49620-80-C-001 to the University of Maryland.

Authors' current addresses: D.D. Dunlop, Intermetrics, Inc., 4733 Bethesda Ave., Suite 415, Bethesda, MD 20814; V.R. Basili, Department of Computer Science, College Park, MD 20742.

function, a *generalized* specification (i.e., one that implies the original specification) which satisfies the closure requirement must be discoverd before these verification conditions can be constructed. This problem is analogous to that of discovering an adequate loop invariant for an inductive assertion proof [7] of the program. These two alternative approaches for verifying looping programs are compared and contrasted in [1, 5].

We remark that the restricted specification often occurs in the process of analyzing an initialized **while** loop, that is, one that consists of a **while** loop preceded by some initialization code. This initialization typically takes the form of assignments of constant values to some of the variables manipulated by the loop. Examples include setting a counter to zero, a search flag to *FALSE*, a queue variable to some particular configuration, etc. It is clear that the initialized loop is correct with respect to some specification if and only if the **while** loop by itself is correct with respect to a slightly modified specification. This specification has the same postcondition as the original specification and a precondition that is the original precondition together with the condition that the initialized variables have their initialized values. Since the initialized variables will typically assume other values as the loop iterates, the loop most likely will not be closed for the domain of this specification, and a generalization of it will be necessary in order to verify the correctness of the program.

Example 1. The following program multiplies natural numbers using repeated addition:

```
{$v \geq 0, k \geq 0$}
$z := 0$;
while $v > 0$ do
  $z := z + k$;
  $v := v - 1$
  od
{$z = v0 * k$}.
```

The term $v0$ appearing in the postcondition refers to the initial value of v. The program is correct if and only if

```
{$z = 0, v \geq 0, k \geq 0$}
while $v > 0$ do
  $z := z + k$;
  $v := v - 1$
  od
{$z = v0 * k$}
```

is correct. Since this loop precondition requires z to have the value 0, and z assumes other values as the loop executes, the loop is not closed for this precondition. Thus, before this program can be verified using the aforementioned technique, this specification must be generalized to something like

```
{$v \geq 0, k \geq 0$}
while $v > 0$ do
  $z := z + k$;
  $v := v - 1$
  od
{$z = z0 + v0 * k$}
```

where $z0$ refers to the initial value of the variable z.

The approach to this problem suggested in this paper is one of observing how particular changes in the value of some input variable (e.g., z in the example) affect the result produced by the loop body of the loop under consideration. Clearly, in general, a change in the value of an input variable may cause an arbitrary (and seemingly unrelated) change in the loop-body result. In many commonly occurring cases, however, the result produced by the loop body is "uniform" across the entire spectrum of possible values for the input variable. The primary purpose of this paper is to present a formal characterization of this type of loop-body behavior. Our experience suggests that loops satisfying the property are susceptible to a routine form of analysis, and hence the formalization is offered as one characterization of a "well-structured" (in the semantic sense) loop. The property is exploited specifically here in order to obtain a generalized specification for the loop being analyzed.

The generalizations considered here have the property that the loop is correct with respect to the generalization if and only if the loop is correct with respect to the original specification. Thus if the loop is closed for the domain of the generalization, the program can be proved/disproved by verifying it relative to the generalization.

It is natural to expect that the ease with which a generalized specification may be obtained for a loop would depend largely on the nature of the loop. Results in [13], for example, show that the problem of generalizing the loop specification for any program in a particular class of programs is NP-complete. On the other hand, work presented here and elsewhere [2, 4, 11] indicates that there do exist categories of loops for which generalized specifications can be obtained in a direct, routine manner. We feel that the notion of "uniform" loop-body behavior discussed in this paper is valuable not only as a tool by which such generalizations may be obtained, but also as an attempt at a characterization of loops which are susceptible to routine analysis, and hence in this sense, easy to verify and comprehend.

The following section defines the necessary notation and terminology and then introduces the idea of a generalized loop specification. Section 3 defines a uniformly implemented loop and states several implications of this definition for the problem of generalizing a specification for such a loop. These results are applied in several example programs in Section 4. In Section 5, a simplified procedure is suggested for proving/disproving a uniformly implemented loop correct with respect to the obtained generalization. Finally, several guidelines for recognizing uniformly implemented loops are presented in Section 6.

2. PRELIMINARIES

We consider a verification problem of the form

```
{$\langle z, x\rangle \in D(f)$}
while $B(\langle z, x\rangle)$ do
  $z, x := h'(z, x), h''(z, x)$
  od
{$\langle z, x\rangle = f(\langle z0, x0\rangle)$}.
```

In this problem, f is a data-state-to-data-state function. The data state consists of two variables, z and x. The notation $D(f)$ means the set of states in the domain of f (i.e., the set of states for which f is defined). The terms $z0$ and $x0$ refer to the initial values of z and x, respectively. The effect of the loop body is partitioned into two functions h' and h'' which describe the new values of z and x, respectively.

The loop is referred to as P. The data-state-to-data-state function computed by the loop (which, presumably, is not explicitly known) is denoted by $[P]$. Thus $D([P])$ is the set of states for which P terminates. As a shorthand notation, we use Y for the state $\langle z, x\rangle$ and H for the data-state-to-data-state function computed by the loop body, i.e.,

$$H(Y) = H(\langle z, x\rangle) = \langle h'(z, x), h''(z, x)\rangle.$$

Note that the important assumption being made is that the complete semantics of the loop body in question are understood, that is, that the definition of H (and, in turn, h' and h'') is known over the entire loop-body input domain. Depending on the nature of an actual loop appearing in a program, deriving H from the loop-body text may be an arbitrarily complex problem (e.g., the loop body may itself be a nested **while–do** loop). We do not address this problem here; we assume the loop being analyzed has been transformed by some technique into the above program schema.

Suppose the loop is not closed for $D(f)$ in that this set contains only a restricted collection of values (maybe only one) of z and that other intermediate values of z occur as the loop iterates. The variable z is called the *key variable*. Our goal here is to discover some more general specification f' which includes each of these intermediate values of the key variable in its domain. This generalization process (in one form or another) is necessary for a proof of correctness of the program under consideration.

Definition. P is *correct* with respect to (wrt) a function f if and only if (iff) for all Y in $D(f)$, $[P](Y)$ is defined and $[P](Y) = f(Y)$.

Definition. A superset f' of f is a *valid generalization* of f iff P is correct wrt f, then P is correct wrt f'.

Note that the collection of supersets of f is partially ordered by "is a valid generalization of." The following definition supplies the notation we use to describe generalizations of the specification function f.

Definition. If S is a set of ordered pairs of data states, f' is the *extension of f defined by S* iff f' is a function and f' is the union of f and S.

Definition. If g is the extension of f defined by

$$\{(Y, Y) \mid {}^{\sim}B(Y)\} \tag{1}$$

then g is the *base generalization* of f.

Thus, if g is the base generalization of f, then

$$g(Y1) = Y2 \leftrightarrow (f(Y1) = Y2 \text{ or } ({}^{\sim}B(Y1) \;\&\; Y1 = Y2)).$$

Throughout this paper, we continue to use the function symbol g for the base generalization of f. We remark that g exists provided the union of f and (1) is a function, i.e., provided

$$Y \in D(f) \ \& \ \tilde{}B(Y) \rightarrow f(Y) = Y$$

holds. If this condition is not satisfied, P must not be correct wrt f. Hence, if P is correct wrt f, the base generalization of f exists.

THEOREM 1. *If g is the base generalization of f, then g is a valid generalization of f.*

PROOF. Suppose P is correct wrt f. We must show that P is correct wrt g. Let $Y \in D(g)$. If $Y \in D(f)$, the loop handles the input correctly by hypothesis. If Y is not in $D(f)$, we must have $\tilde{}B(Y)$ and $g(Y) = Y$. The program and g map Y to itself, and thus are in agreement. Consequently P is correct wrt g, and g is a valid generalization of f. □

The theorem utilizes the fact that the loop must compute the identity function over inputs where the loop predicate is false. Combining this information with the program specification f results in a valid generalization of f.

Definition. A valid generalization f' of f is *adequate* if the loop is closed for $D(f')$.

The important characteristic of an adequate valid generalization f' is that it can be used to prove/disprove the correctness of P wrt the original specification f. Since the loop is closed for $D(f')$, P can be proved/disproved correct wrt f' using standard techniques [3, 8–10, 12, 13]. Specifically, P is correct wrt f' iff each of the following conditions hold:

$$\text{for all} \quad Y \in D(f') \quad \text{the loop terminates} \tag{2}$$

$$Y \in D(f') \ \& \ \tilde{}B(Y) \rightarrow f'(Y) = Y \tag{3}$$

$$Y \in D(f') \ \& \ B(Y) \rightarrow f'(Y) = f'(H(Y)). \tag{4}$$

If P is correct wrt f', then P is necessarily correct wrt any subset of f', including f. If P is not correct wrt f', then by the definition of a valid generalization, P must not be correct wrt f.

Example 2. The following program tests whether a particular key appears in an ordered binary tree:

```
{success = FALSE}
while tree ≠ NULL and ~success do
  if     name(tree) = key then success := TRUE
  elseif name(tree) < key then tree := right(tree)
  else                         tree := left(tree) fi
  od
{success = IN(tree0, key)}
```

The function $IN(tree0, key)$ appearing in the postcondition is a predicate which means "the ordered binary tree *tree*0 contains a node with name field *key*." The Boolean variable *success* is chosen as the key variable since it is constrained to the value *FALSE* in the input specification. Thus *success* plays the role of z and the pair of variables $\langle tree, key \rangle$ correspond to x in the program schema discussed above. The specification function f is

$$f(\langle FALSE, tree, key \rangle) = \langle IN(tree, key),\ tree',\ key' \rangle$$

where $tree'$ and key' are the final values of the variables *tree* and *key* computed by the loop, respectively. That is, since the final values of these variables are not of interest in this example, we specify these final values so as to be automatically correct. Using Theorem 1, a valid generalization of this specification is

```
$g(\langle success, tree, key \rangle) =$ if $\tilde{}success$ then
                          $\langle IN(tree, key), tree', key' \rangle$
                        elseif $tree = NULL$ or $success$ then
                          $\langle success, tree, key \rangle$
```

which is equivalent to the following.

$$g(\langle success,\ tree,\ key \rangle) = \langle success \textbf{ or } IN(tree,\ key),\ tree',\ key' \rangle.$$

In this example, the domain of the base generalization g of f includes each value of the key variable (i.e., *FALSE* and *TRUE*) and is thus adequate. Consequently, this generalization can be used to prove/disprove the correctness of the program.

In most cases, however, the heuristic suggested in Theorem 1 is insufficient to generate an adequate generalization. Indeed, the base generalization is an adequate generalization only in the case when the sole reason for the closure condition not holding is the existence of potential final values of the key variable (e.g., *TRUE* in the example) which are absent from $D(f)$. In order to obtain a generalization that includes general values of the key variable, an important characteristic of the loop body which seems to be present in many commonly occurring loops will be exploited.

3. UNIFORMLY IMPLEMENTED LOOPS

Definition. Let P be a loop of the form described above. Let C be a set, let Z be the set of values the key variable z may assume, and let X be the set of values the remaining variable or variables x may assume. Let

$$\$' : C \times Z \rightarrow Z$$

be an infix binary operator. The loop P is *uniformly implemented* wrt $\$'$ iff each of the following conditions hold

$$B(\langle z, x \rangle) \rightarrow h'(c\$'z, x) = c\$'h'(z, x) \tag{5}$$

$$B(\langle z, x \rangle) \rightarrow h''(c\$'z, x) = h''(z, x) \tag{6}$$

$$B(\langle z, x \rangle) \rightarrow B(\langle c\$'z, x \rangle) \tag{7}$$

for all $c \in C$, $z \in Z$, and $x \in X$.

Conditions (5) and (6) of this definition state that a modification to the key variable by the operation $\$'$ causes a *slight* but *orderly* change in the result produced by the loop body. The change is slight because the only difference in the result produced by the loop body occurs in the key variable. The difference is orderly because it corresponds precisely to the same $\$'$ operation that served to modify the input value of the key variable. Condition (7) specifies that such a modification does not cause the loop predicate B to change from true to false.

As a shorthand notation, we define the infix operator $\$$ as

$$c \$ Y = c \$ \langle z, x \rangle = \langle c \$' z, x \rangle.$$

In this notation, (5)–(7) are equivalent to

$$B(Y) \rightarrow c \$ H(Y) = H(c \$ Y) \tag{8}$$

and

$$B(Y) \rightarrow B(c \$ Y). \tag{9}$$

Example 3. Consider again the program from Example 1 which multiplies natural numbers using repeated addition:

```
{$z = 0, v \geq 0, k \geq 0$}
while $v > 0$ do
  $z := z + k$;
  $v := v - 1$
  od
{$z = v0 * k$}.
```

Let z be the key variable. The pair $\langle v, k \rangle$ corresponds to the variable x occurring in the above schema. The loop is uniformly implemented wrt $+$, where C and Z are both the set of natural numbers. Note that adding some constant to the input value of z has the effect of adding the same constant to the value of z output by the loop body. Now consider the following alternative implementation of multiplication:

```
{$z = 0, v \geq 0, k \geq 0$}
while $v > 0$ do
    if $k = 0$ and $z = 0$     then $z := v - 1$
  elseif $k = 0$ and $z \neq 0$ then $z := z - 1$
  elseif $z < k$             then $z := z + k$
  elseif $z = k$             then $z := z * 2 * v$
  else                            $z := z - k$ fi;
  $v := v - 1$
  od
{$z = v0 * k$}.
```

Again, let z be the key variable. This loop is not uniformly implemented wrt $+$. Intuitively, this is due to the high degree of dependence of the loop-body behavior on the value of the key variable. The result of this dependence is that adding some constant to the value of z causes an unorderly change in the value of z output by the loop body.

The reader may wonder whether the second multiplication program above might be uniformly implemented with respect to some operation other than $+$. We remark that *any* loop is uniformly implemented wrt $\$' : C \times Z \rightarrow Z$ defined by

$$c \$' z = z$$

for all $c \in C$ and $z \in Z$. For the purpose of this paper, we rule out such trivial operations, i.e., we require that for some $z \in Z$, there exists some $c \in C$ such that

$$c\$'z \neq z.$$

With this assumption, there does not exist an operation with respect to which the second of the above loops is uniformly implemented (or more briefly, the loop is not uniformly implemented). To see this, suppose the loop were uniformly implemented wrt $\$' : C \times Z \rightarrow Z$. Let $c0$ and $z0$ be some fixed elements of C and Z, respectively, which satisfy

$$c0\$'z0 \neq z0.$$

Since (5) must hold for all $c \in C$, $z \in Z$, and $x \in X$, we choose $z = z0$, $c = c0$, and $k = c0\$'z0$. Applying (5) gives

$$v > 0 \rightarrow h'(c0\$'z0, \langle v, c0\$'z0\rangle) = c0\$'h'(z0, \langle v, c0\$'z0\rangle)$$

for all v. We consider three exhaustive cases based on the values of $c0$ and $z0$. First, suppose $c0\$'z0 = 0$. Then we must have (since $z0 \neq 0$)

$$v > 0 \rightarrow v - 1 = c0\$'(z0 - 1).$$

Since this must hold for all v, and since the value of $c0\$'(z0 - 1)$ is fixed by the original selection of $c0$ and $z0$, this is a contradiction. Next, suppose $c0\$'z0 \neq 0$ and $z0 < c0\$'z0$. Then

$$v > 0 \rightarrow (c0\$'z0) * 2 * v = c0\$'(z0 + (c0\$'z0))$$

for all v. Again, since the expression to the right of the equality sign is fixed and $(c0\$'z0) * 2 * v$ varies with different values of v ($c0\$'z0$ is nonzero), this is a contradiction. The third case, where $c0\$'z0 \neq 0$ and $z0 > c0\$'z0$, leads to a similar contradiction, and thus (5) does not hold. We conclude that the second multiplication program above is not uniformly implemented. That is, there does not exist a nontrivial modification that can be applied to the variable z which always results in a slight and orderly change in the result produced by the loop body.

The results presented here are based on the following lemma concerning uniformly implemented loops. The lemma describes the output of a uniformly implemented loop P for some modified input $c\$Y$ (i.e., $[P](c\$Y)$) in terms of the output of the loop for the input Y (i.e., $[P](Y)$) and the output of the loop for the input $c\$[P](Y)$ (i.e., $[P](c\$[P](Y))$).

LEMMA 1. *Let P be uniformly implemented wrt $\$'$. Then*

$$Y \in D([P]) \rightarrow [P](c\$Y) = [P](c\$[P](Y)). \tag{10}$$

PROOF. We use induction on the number of iterations of P on Y. For the base case of 0 iterations, $[P](Y) = Y$, and the lemma holds. Suppose it holds for all input data states Y requiring $n - 1$ iterations where $n > 0$. Let $Y1$ require n iterations. Since $n > 0$, $B(Y1)$ holds. By (9), $B(c\$Y1)$. Note that $H(Y1)$ requires $n - 1$ iterations on P; thus by the inductive hypothesis,

$$[P](c\$H(Y1)) = [P](c\$[P](H(Y1))).$$

Due to the uniform implementation this is

$$[P](H(c\$ Y1)) = [P](c\$[P](H(Y1))).$$

Using the loop property $B(Y) \rightarrow [P](Y) = [P](H(Y))$ on both sides, we get

$$[P](c\$ Y1) = [P](c\$[P](Y1)).$$

Thus the inductive step holds and the lemma is proved. □

The general idea behind our use of the lemma is as follows. Suppose the value $[P](Y)$ is known for some particular Y; that is, suppose we know what the loop produces for the input Y. In addition, suppose that, given the result $[P](Y)$, the quantity $[P](c\$[P](Y))$ is also known. With this information, we can then use Lemma 1 to "solve" for the (possibly unknown) value $[P](c\$ Y)$. This additional information concerning the input/output behavior of the loop can be used as an aid in constructing a valid generalization of the specification f.

How can we find the value $[P](Y)$ and then the value $[P](c\$[P](Y))$ for some Y? The key lies in assuming the loop P is correct wrt f. If P is not correct wrt f, *any* generalization of f obtained by the technique will be a valid generalization by definition. Under this assumption, $[P](Y)$ is known for $Y \in D(f)$, that is, $Y \in D(f) \rightarrow [P](Y) = f(Y)$, and hence Lemma 1 implies

$$Y \in D(f) \rightarrow [P](c\$ Y) = [P](c\$f(Y)). \tag{11}$$

Consider now the base generalization g of f defined in Section 2. Recall that g is simply f augmented with the identity function over the domain where the loop predicate B is false. Assuming as before that P is correct wrt f, P is then correct wrt g by Theorem 1; hence $Y \in D(g) \rightarrow [P](Y) = g(Y)$. Thus (11) implies

$$Y \in D(f) \text{ and } c\$f(Y) \in D(g) \rightarrow [P](c\$ Y) = g(c\$f(Y)). \tag{12}$$

Thus we can "solve" for the behavior of the loop on the input $c\$ Y$, assuming $Y \in D(f)$, $c\$f(Y) \in D(g)$, and P is correct wrt f. This suggests that if f' is the extension of f defined by

$$\{(c\$ Y, g(c\$f(Y))) \mid Y \in D(f) \text{ and } c\$f(Y) \in D(g)\} \tag{13}$$

then f' is a valid generalization of f. Before giving a formal proof of this result, however, we first consider the question of the existence of such an extension of f. Specifically, it could be that for some c and Y satisfying $Y \in D(f)$ and $c\$f(Y) \in D(g)$, $c\$\ Y \in D(f)$ and $f(c\$ Y) \neq g(c\$f(Y))$. This would imply that the extension of f defined by (13) does not exist. The following theorem states that this implies P is not correct wrt f.

THEOREM 2. *Let P be uniformly implemented wrt $\$'$. Let g be the base generalization of f. If P is correct wrt f, then there exists a function f' which is the extension of f defined by* (13), *that is,*

$$\{(c\$ Y, g(c\$f(Y))) \mid Y \in D(f) \text{ and } c\$f(Y) \in D(g)\}.$$

PROOF. Let f' be the function computed by the loop, i.e., $[P]$. Since P is correct wrt f, P is correct wrt g, and f' is a superset of both f and g. By the lemma,

$$f'(c\$Y) = f'(c\$f'(Y))$$

for all $Y \in D(f)$. Since $f'(Y) = f(Y)$ for $Y \in D(f)$ and $f'(Y) = g(Y)$ for $Y \in D(g)$,

$$Y \in D(f) \text{ and } c\$f(Y) \in D(g) \rightarrow f'(c\$Y) = g(c\$f(Y))$$

holds. Thus (13) is a subset of f'. Since f is a subset of f', the union of f and (13) is a subset of f'. Hence this union is a function and is thus the extension of f defined by (13). □

The following theorem is the central result presented here. The theorem formalizes the use of Lemma 1 in the manner suggested above, that is, that the extension of f described in the previous theorem is a valid generalization of the original specification.

THEOREM 3. *Let P be uniformly implemented wrt $\$'$. Let g be the base generalization of f. If f' is the extension of f defined by* (13), *that is,*

$$\{(c\$Y, g(c\$f(Y))) \mid Y \in D(f) \text{ and } c\$f(Y) \in D(g)\}$$

then f' is a valid generalization of f.

PROOF. Suppose P is correct wrt f. We must show P is correct wrt f'. Let $Y \in D(f')$. If $Y \in D(f)$, the loop handles the input correctly by hypothesis. If Y is not in $D(f)$, we must have $Y = c\$Y1$ where $Y1 \in D(f)$ and $c\$f(Y1) \in D(g)$. By Lemma 1, $[P](Y) = [P](c\$[P](Y1))$. Since P is correct wrt f, this is $[P](Y) = [P](c\$f(Y1))$. By Theorem 1, P is correct wrt g. Using this, the equality can be written as $[P](Y) = g(c\$f(Y1))$. By the definition of f', this implies $[P](Y) = f'(Y)$. Thus P and f' are in agreement on the input Y and consequently are in agreement on any input in $D(f')$. Hence P is correct wrt f' and thus f' is a valid generalization of f.

The significance of Theorem 3 is that it provides a guideline for generalizing the specification of a uniformly implemented loop. If the loop is closed for the domain of the resulting specification, the generalization can then be used to prove/disprove the program correct wrt the original specification.

4. APPLICATIONS

In this section we illustrate the use of Theorem 3 with a number of example programs which fall into either of two subclasses of uniformly implemented loops. The subclasses correspond to the two possible circumstances which can occur when $c\$f(Y)$ of set definition (13) belongs to the set $D(g)$: the first because $\tilde{}B(c\$f(Y))$, and the second because $c\$f(Y) \in D(f)$. In each of these situations, the set definition (13) takes on a particularly simple form.

Definition. A uniformly implemented loop satisfying

$$\tilde{}B(Y) \rightarrow \tilde{}B(c\$Y)$$

is a *Type A* loop.

Observe that this condition along with (9) indicates that a Type A uniformly implemented loop satisfies

$$B(Y) \leftrightarrow B(c\$ Y)$$

i.e., the value of the loop predicate B is independent of a change to the data state by the operator \$.

The intuition behind a Type A uniformly implemented loop is as follows. Whenever an execution of a Type A loop terminates (i.e., $\tilde{}B(Y)$ holds) and the resulting data state is modified by the operator \$, the result is a new data state which, when viewed as a loop input, corresponds to zero iterations of the loop (i.e., the predicate B is still false despite the modification). This property is reflected in the following corollary.

COROLLARY 1. *Let P be a Type A loop. If f' is the extension of f defined by*

$$\{(c\$ Y, c\$ f(Y)) \mid Y \in D(f)\} \tag{14}$$

then f' is a valid generalization of f.

PROOF. The proof consists of showing that (13) and (14) are equivalent for a Type A loop which is correct wrt f. By Theorem 3, the corollary then holds. Let P be a Type A loop which is correct wrt f. A consequence of the correctness property is that $\tilde{}B(f(Y))$ for all $Y \in D(f)$. Since P is a Type A loop, this implies $\tilde{}B(c\$ f(Y))$. Thus $c\$ f(Y) \in D(g)$ and $g(c\$ f(Y)) = c\$ f(Y)$. Consequently, (13) and (14) are equivalent.

Of course, once a generalization f' has been obtained via Corollary 1, there is no reason that result cannot be fed back into the corollary to obtain a (possibly) further generalization f'' (using f' for f, f'' for f'). This notion suggests the following general case of Corollary 1. □

COROLLARY 2. *Let P be a Type A loop. If f' is the extension of f defined by*

$$\{(c1\$ (c2\$ \ldots (cn\$ Y) \ldots), c1\$ (c2\$ \ldots (cn\$ f(Y)) \ldots)) \mid Y \in D(f) \text{ and } n \geq 0\}$$

then f' is a valid generalization of f.

Example 4. Consider the following program to compute exponentiation:

```
{w = 1, e > 0, d ≥ 0}
while d > 0 do
  if odd(d) then w := w*e fi;
  e := e*e; d := d/2
  od
{w = e0 ^ d0}
```

The infix operator, denoted by a caret appearing in the postcondition, represents integer exponentiation. In this example, w plays the role of the key variable z, and the pair $\langle e, d\rangle$ corresponds to the variable x. We now consider with respect to what operations the loop might be uniformly implemented. For any operation \$', (7) holds (because w does not appear in the loop predicate) as does (6) (because the values produced in e and d are independent of w). Furthermore, (5) must hold for inputs which bypass the updating of w. Thus the uniformity conditions reduce to

$$d > 0 \text{ and } odd(d) \rightarrow (c \$' w) * e = c \$' (w * e).$$

Due to its associativity, it is clear the loop is uniformly implemented with respect to $*$, where the sets C and Z are the set of integers. Since they key variable does not appear in the loop predicate, it is necessarily a Type A loop. The specification function here is

$$f(\langle 1, e, d\rangle) = \langle e \hat{\ } d, e', d'\rangle$$

where $e > 0$, $d \geq 0$, and e' and d' are the final values computed by the loop for the variables e and d. Applying Corollary 1, the function f' defined by

$$f'(\langle c * 1, e, d\rangle) = \langle c * (e \hat{\ } d), e', d'\rangle$$

where $e > 0$ and $d \geq 0$ is a valid generalization of f. Since this holds for all c, the definition of f' can be rewritten as

$$f'(\langle w, e, d\rangle) = \langle w * (e \hat{\ } d), e', d'\rangle$$

where w is an arbitrary integer, $e > 0$ and $d \geq 0$. The generalization f' is adequate and can thus be used to test the correctness of the program wrt the original specification. Applying (2), (3), and (4), these necessary and sufficient verification conditions are

(i) the loop terminates for all $e > 0$, $d \geq 0$,
(ii) $d = 0 \rightarrow w = w * (e \hat{\ } d)$,
(iii) $w * (e \hat{\ } d)$ is a loop constant (i.e., $e0 \hat{\ } d0 = w * (e \hat{\ } d)$ is a loop invariant),

respectively. In Section 5, we discuss a simplification of the last of these verification conditions which applies for uniformly implemented loops.

Example 5 [11]. The following program constructs the preorder traversal of a binary tree with root node r. The program uses a stack variable *st* and records the traversal in a sequence variable *seq*.

```
{seq = NULL, st = (r) * stack st contains only the root node r*/}
while st ≠ EMPTY do
  p ≤ st;   /* pop the top off the stack */
  seq := seq || name(p);   /* concatenate name of p onto seq */
  if right(p) ≠ NIL then st ≤ right(p) fi;   /* push onto st */
  if left(p) ≠ NIL then st ≤ left(p) fi
  od
{seq = PREORDER(r)}
```

The function *PREORDER*(r) appearing in the postcondition is the sequence consisting of the preorder traversal of the binary tree with root node r. Let *seq* be the key variable. Reasoning similar to that employed in Example 4 indicates here that the loop is uniformly implemented wrt $\|$, where the sets C and Z are the set of all sequences. It is a Type A loop. The specification function is

$$f(\langle NULL, (r)\rangle) = \langle PREORDER(r), st'\rangle.$$

Again, the prime notation is used to represent the final values of variables that are of no interest. Applying Corollary 1 we obtain

$$f'(\langle seq, (r)\rangle) = \langle seq \parallel PREORDER(r), st'\rangle$$

as a valid generalization of f. In this case, f' is not adequate since it does not specify a behavior of the loop for arbitrary values of the stack st. We will return to this example after considering another subclass of uniformly implemented loops.

Definition. A uniformly implemented loop satisfying

$$\tilde{}B(Y) \rightarrow c\,\$\,Y \in D(f)$$

is a *Type B* loop.

The intuition behind a Type B uniformly implemented loop is as follows. Whenever an execution of a Type B loop terminates (i.e., $\tilde{}B(Y)$ holds) and the resulting data state is modified by the operator \$, the result is a new data state which is a "valid" starting point for a new execution of the loop (i.e., this new state is in $D(f)$). This property is reflected in the following corollary.

COROLLARY 3. *Let P be a Type B loop. If f' is the extension of f defined by*

$$\{(c\,\$\,Y, f(c\,\$\,f(Y))) \mid Y \in D(f)\} \tag{15}$$

then f' is a valid generalization of f.

PROOF. The proof consists of showing that (13) and (15) are equivalent for a Type B loop which is correct wrt f. By Theorem 3, the corollary then holds. Let P be a Type B loop which is correct wrt f. A consequence of the correctness property is that $\tilde{}B(f(Y))$ for all $Y \in D(f)$. Since P is a Type B loop, this implies $c\,\$\,f(Y) \in D(f)$. Thus $c\,\$\,f(Y) \in D(g)$ and $g(c\,\$\,f(Y)) = f(c\,\$\,f(Y))$. Consequently, (13) and (15) are equivalent.

As before, a general case of this corollary can be stated which corresponds to an arbitrary number of its applications.

COROLLARY 4. *Let P be a Type B loop. If f' is the extension of f defined by*

$$\{(c1\,\$\,(c2\,\$\,(\ldots\$\,(cn\,Y)\ldots)), f(c1\,\$\,f(c2\,\$\,f(\ldots\$\,f(cn\,\$\,f(Y))\ldots)))) \mid Y \in D(f) \text{ and } n \geq 0\},$$

then f' is a valid generalization of f.

Example 5 (continued). We now consider the problem of further generalizing the derived specification in the previous example. The variable for which the loop is not closed, st, will now be the key variable. Consider an operation $c\,\$'\,st$ that has the effect of adding an element c to the stack st. Before being more precise about this operation, we consider how the loop body works and how its output depends on the value of the key variable st.

We observe that the loop-body behavior relies heavily on the characteristics of the node on the top of the stack. Consequently, a modification $c\,\$'\,st$ to st which pushed a new node c onto the top of st would not cause a slight and orderly change in the result produced by the loop body, and the uniformity conditions

(5)–(7) would not hold. However, because the loop-body behavior seems to be independent of what lies underneath the top of the stack, we suspect the loop is uniformly implemented wrt *ADDUNDER*, where C is the set of binary tree nodes, Z is the set of stacks of binary tree nodes, and c *ADDUNDER* *st* is the stack that results from adding c to the bottom of *st*. Conditions (5)–(7) for this operation indicate that, indeed, this is the case.

Let f be the generalization f' from the previous example. In keeping with the convention described above, since *st* is now the key variable, we reverse the order in which the two variables appear in the data state, that is, we will write $\langle st, seq\rangle$ instead of $\langle seq, st\rangle$.

The program is a Type B uniformly implemented loop since

$$st = EMPTY \rightarrow \langle c\ ADDUNDER\ st, seq\rangle \in D(f)$$

where c is a node of a binary tree, and specifically

$$st = EMPTY \rightarrow f(\langle c\ ADDUNDER\ st, seq\rangle) = \langle st', seq \,\|\, PREORDER(c)\rangle. \quad (16)$$

Applying Corollary 4, if $(r, cn, \ldots, c1)$ is an arbitrary stack (with r on top, $c1$ on the bottom, $n \geq 0$), then

$$\begin{aligned} & f'(\langle (r, cn, \ldots, c1), seq\rangle) \\ &= f'(c1 \,\$\, (c2 \,\$\, (\ldots \$\, (cn \,\$\, \langle (r), seq\rangle) \ldots))) \\ &= f(c1 \,\$\, f(c2 \,\$\, f(\ldots \$\, f(cn \,\$\, f(\langle (r), seq\rangle)) \ldots))) \\ &= f(c1 \,\$\, f(c2 \,\$\, f(\ldots \$\, f(cn \,\$\, \langle st', seq \,\|\, PREORDER(r)\rangle) \ldots))). \end{aligned}$$

Recall that st' refers to the final value of *st* computed by the loop. The loop predicate indicates this will always be the value *EMPTY*. Hence (16) can be applied to this expression from inside out, giving

$$\begin{aligned} & f(c1 \,\$\, f(c2 \,\$\, f(\ldots \$\, \langle st', seq \,\|\, PREORDER(r) \,\|\, PREORDER(cn)\rangle \ldots))) \\ &= \cdots \\ &= \langle st', seq \,\|\, PREORDER(r) \,\|\, PREORDER(cn) \,\|\, \ldots \,\|\, PREORDER(c1)\rangle. \end{aligned}$$

Note that f' now defines a loop behavior for all sequences *seq* and nonempty stacks *st*. The base generalization of f' supplements f' with a behavior for the empty stack *st* and is thus an adequate generalization.

Example 6 [6]. The following program computes Ackermann's function using a sequence variable s of natural numbers. The notation $s(1)$ is the rightmost element of s and $s(2)$ is the second rightmost, etc. The sequence $s(..3)$ is s with $s(2)$ and $s(1)$ removed.

```
{$s = \langle m, n\rangle, m \geq 0, n \geq 0$}
while $size(s) \neq 1$ do
  if     $s(2) = 0$ then $s := s(..3) \| \langle s(1) + 1\rangle$
  elseif $s(1) = 0$ then $s := s(..3) \| \langle s(2) - 1, 1\rangle$
  else                   $s := s(..3) \| \langle s(2) - 1, s(2), s(1) - 1\rangle$ fi
  od
{$s = \langle A(m, n)\rangle$}
```

The function $A(m, n)$ appearing in the postcondition is Ackermann's function. The specification function is

$$f(\langle s(2), s(1)\rangle) = \langle A(s(2), s(1))\rangle.$$

Let s be the key variable. Because the loop-body behavior is independent of the leftmost portion of s, the loop is uniformly implemented wrt $|$, where C is the set of natural numbers, Z is the set of nonempty sequences of natural numbers, and $c \mid s = \langle c\rangle \parallel s$. The program is also a Type B loop. By Corollary 4 (where $n > 1$),

$$\begin{aligned}
&f'(\langle s(n), s(n-1), \ldots, s(1)\rangle)\\
&= f'(s(n) \,\$\, (s(n-1) \,\$\, (\ldots \$\, (s(3) \,\$\, \langle s(2), s(1)\rangle) \ldots)))\\
&= f(s(n) \,\$\, f(s(n-1) \,\$\, f(\ldots \$\, f(s(3) \,\$\, f(\langle s(2), s(1)\rangle)) \ldots)))\\
&= f(s(n) \,\$\, f(s(n-1) \,\$\, f(\ldots \$\, f(s(3) \,\$\, \langle A(s(2), s(1))\rangle) \ldots)))\\
&= f(s(n) \,\$\, f(s(n-1) \,\$\, f(\ldots \$\, f(\langle s(3), A(s(2), s(1))\rangle) \ldots)))\\
&= f(s(n) \,\$\, f(s(n-1) \,\$\, f(\ldots \$\, \langle A(s(3), A(s(2), s(1)))\rangle \ldots)))\\
&= \cdots\\
&= \langle A(s(n), A(s(n-1), \ldots, A(s(3), A(s(2), s(1))) \ldots))\rangle
\end{aligned}$$

is a valid generalization of f. As in the previous example, the base generalization of this function is adequate.

5. SIMPLIFYING THE ITERATION CONDITION

The view of **while** loop verification presented here is one of a two step process, the first step being the discovery of an adequate valid generalization f' of the loop specification f, the second being the proof of three basic conditions (i.e., (2)–(4)) based on this generalization. We have seen that the uniform nature of a loop implementation may be used in the first step as an aid in discovering an appropriate generalization. In this section, we exploit the same loop characteristic to substantially simplify one of the conditions which must be proven in the second step of this process.

The verification condition of interest is (4) above, that is,

$$Y \in D(f') \text{ and } B(Y) \rightarrow f'(Y) = f'(H(Y))$$

and is labeled the *iteration condition* in [10]. This condition assures that as the loop executes, the intermediate values of Y remain in the same level set of f', that is, the value of f' is constant across the loop iterations. Previously we argued that if P is uniformly implemented wrt $\$'$, a change in the key variable by $\$'$ causes a slight but orderly change in the result produced by H. Roughly speaking then, the behavior of H is largely independent of the key variable. If f' is chosen so as to be equally independent of the key variable, and the above condition holds for $Y = \langle z, x\rangle$, where x is arbitrary but the key variable z has a specific simple value, we might expect the condition to hold for all Y. Such an expectation would be based on the belief that the truth or falsity of this condition would also be largely independent of the key variable.

We formally characterize this circumstance in the following definition.

Definition. Let P be a loop of the form described above. A generalization f' of f is *represented by* f iff

$$Y \in D(f) \text{ and } B(Y) \rightarrow f(Y) = f'(H(Y)) \tag{17}$$

implies

$$Y \in D(f') \text{ and } B(Y) \rightarrow f'(Y) = f'(H(Y)). \tag{18}$$

Thus if f' is represented by f, condition (17) can be used in place of the iteration condition (18) in proving the loop is correct wrt f' (and hence wrt f). The significance of this situation is that the iteration condition can be tested with the key variable constrained by initialization (as prescribed in $D(f)$). In practice, the result is one of having to prove a substantially simpler verification condition.

The following theorems state that the use of Corollaries 2 and 4 lead to generalizations which are represented by the original specification.

THEOREM 4. *Let P be a Type A loop. Suppose f' is the valid generalization of f defined in Corollary 2 and suppose f' is adequate. Then f' is represented by f.*

PROOF. Suppose (17) holds and select some arbitrary Y' from $D(f')$ satisfying $B(Y')$. Thus there exists $c1, \ldots, cn \in C$, $n \geq 0$, and $Y \in D(f)$ such that

$$Y' = c1\,\$\,(c2\,\$\,(\ldots\$\,(cn\,\$\,Y)\ldots)).$$

By the definition of a Type A loop, we must have $B(Y)$. Applying the definition of f' yields

$$f'(Y') = c1\,\$\,(c2\,\$\,(\ldots\$\,(cn\,\$\,f(Y))\ldots))$$

which is equal to

$$c1\,\$\,(c2\,\$\,(\ldots\$\,(cn\,\$\,f'(H(Y))\ldots))$$

by (17) since $B(Y)$ holds. Since $H(Y) \in D(f')$ (since f' is adequate), there exists $d1, \ldots, dm \in C$, $m \geq 0$, and $Y1 \in D(f)$ such that

$$H(Y) = d1\,\$\,(d2\,\$\,(\ldots\$\,(dm\,\$\,Y1)\ldots)).$$

Furthermore,

$$f'(H(Y)) = d1\,\$\,(d2\,\$\,(\ldots\$\,(dm\,\$\,f(Y1))\ldots)).$$

Hence, continuing from above,

$$f'(Y') = c1\,\$\,(\ldots\$\,(cn\,\$\,(d1\,\$\,(\ldots\$\,(dm\,\$\,f(Y1))\ldots)))\ldots)$$

which is equal to

$$f'(c1\,\$\,(\ldots\$\,(cn\,\$\,(d1\,\$\,(\ldots\$\,(dm\,\$\,Y1)\ldots)))\ldots))$$

from the definition of f'. Thus

$$f'(Y') = f'(c1\,\$\,(\ldots\$\,(cn\,\$\,H(Y))\ldots))$$

which is equal to

$$f'(H(c1\,\$\,(\ldots\$\,(cn\,\$\,Y)\ldots)))$$

from the uniformity condition (8). Hence

$$f'(Y') = f'(H(Y'))$$

and the theorem is proved. □

THEOREM 5. *Let P be a Type B loop. Suppose f' is the valid generalization of f defined in Corollary 4 and suppose f' is adequate. Then f' is represented by f.*

PROOF. Suppose (17) holds and select some arbitrary Y' from $D(f')$ satisfying $B(Y')$. Thus there exists $c1, \ldots, cn \in C$, $n \geq 0$, and $Y \in D(f)$ such that

$$Y' = c1\,\$\,(c2\,\$\,(\ldots\$\,(cn\,\$\,Y)\ldots)).$$

We make the assumption that $B(Y)$. Otherwise, by the definition of a Type B loop, the term $cn\,\$\,Y$ can be replaced by another $Y \in D(f)$. Since $B(Y')$, this process can be continued until Y' is written in the form above, with $Y \in D(f)$ and $B(Y)$. Applying the definition of f' yields

$$f'(Y') = f(c1\,\$\,f(c2\,\$\,f(\ldots\$\,f(cn\,\$\,f(Y))\ldots)))$$

which is equal to

$$f(c1\,\$\,f(c2\,\$\,f(\ldots\$\,f(cn\,\$\,f'(H(Y)))\ldots)))$$

by (17) since $B(Y)$ holds. Since $H(Y) \in D(f')$ (since f' is adequate), there exists $d1, \ldots, dm \in C$, $m \geq 0$, and $Y1 \in D(f)$ such that

$$H(Y) = d1\,\$\,(d2\,\$\,(\ldots\$\,(dm\,\$\,Y1)\ldots)).$$

Furthermore,

$$f'(H(Y)) = f(d1\,\$\,f(d2\,\$\,f(\ldots\$\,f(dm\,\$\,f(Y1))\ldots))).$$

Hence, continuing from above,

$$f'(Y') = f(c1\,\$\,f(\ldots\$\,f(cn\,\$\,f(d1\,\$\,f(\ldots\$\,f(dm\,\$\,f(Y1))\ldots)))\ldots))$$

which is equal to

$$f'(c1\,\$\,(\ldots\$\,(cn\,\$\,(d1\,\$\,(\ldots\$\,(dm\,\$\,Y1)\ldots)))\ldots))$$

from the definition of f'. Thus

$$f'(Y') = f'(c1\,\$\,(\ldots\$\,(cn\,\$\,H(Y))\ldots))$$

which is equal to

$$f'(H(c1\,\$\,(\ldots\$\,(cn\,\$\,Y)\ldots)))$$

from the uniformity condition (8). Hence

$$f'(Y') = f'(H(Y'))$$

and theorem is proved. □

EXAMPLE 7. Consider the exponentiation program of Example 4. The generalization obtained from Corollary 2 is

$$f'(\langle w, e, d\rangle) = \langle w*(e\hat{}d), e', d'\rangle$$

where $e > 0$, $d \geq 0$. Since f' is represented by f, the iteration condition corresponding to (17) is

$$\begin{aligned}&c > 0 \text{ and } d > 0 \text{ and } \mathit{odd}(d) \rightarrow e\hat{}d = e*((e*e)\hat{}(d/2))\\&c > 0 \text{ and } d > 0 \text{ and } \tilde{}\mathit{odd}(d) \rightarrow e\hat{}d = 1*((e*e)\hat{}(d/2))\end{aligned}$$

can be used in place of that corresponding to (18)

$$\begin{aligned}&c > 0 \text{ and } d > 0 \text{ and } \mathit{odd}(d) \rightarrow w*(e\hat{}d) = (w*e)*((e*e)\hat{}(d/2))\\&c > 0 \text{ and } d > 0 \text{ and } \tilde{}\mathit{odd}(d) \rightarrow w*(e\hat{}d) = w*((e*e)\hat{}(d/2)).\end{aligned}$$

The benefits of this simplification are more striking for more complex types of key variables. To illustrate, consider the program to compute Ackermann's function in Example 6. Applying Corollary 4 to the base generalization g of f yields the generalization defined by

$$\begin{aligned}n > 1 \rightarrow {}& f'(\langle s(n), s(n-1), \ldots, s(1)\rangle)\\ = {}& \langle A(s(n), A(s(n-1), \ldots, A(s(3), A(s(2), s(1)))\ldots))\rangle\end{aligned}$$

and

$$f(\langle s(1)\rangle) = \langle s(1)\rangle.$$

Since f' is represented by g, the iteration condition

$$\begin{aligned}s(2) = 0 &\rightarrow \langle A(s(2), s(1))\rangle = \langle s(1) + 1\rangle\\ s(2) \neq 0 \text{ and } s(1) = 0 &\rightarrow \langle A(s(2), s(1))\rangle = \langle A(s(2) - 1, 1)\rangle\\ s(2) \neq 0 \text{ and } s(1) \neq 0 &\rightarrow \langle A(s(2), s(1))\rangle = \langle A(s(2) - 1, A(s(2), s(1) - 1))\rangle\end{aligned}$$

can be used in place of

$$\begin{aligned}&n > 1 \text{ and } s(2) = 0 \rightarrow\\&\quad\langle A(s(n), A(s(n-1), \ldots, A(s(3), A(s(2), s(1)))\ldots))\rangle\\&\quad=\langle A(s(n), A(s(n-1), \ldots, A(s(3), s(1) + 1)\ldots))\rangle\\&\quad n > 1 \text{ and } s(2) \neq 0 \text{ and } s(1) = 0\\&\rightarrow\langle A(s(n), A(s(n-1), \ldots, A(s(3), A(s(2), s(1)))\ldots))\rangle\\&\quad=\langle A(s(n), A(s(n-1), \ldots, A(s(3), A(s(2) - 1, 1))\ldots))\rangle\\&\quad n > 1 \text{ and } s(2) \neq 0 \text{ and } s(1) \neq 0\\&\rightarrow\langle A(s(n), A(s(n-1), \ldots, A(s(3), A(s(2), s(1)))\ldots))\rangle\\&\quad=\langle A(s(n), A(s(n-1), \ldots, A(s(3), A(s(2) - 1, A(s(2), s(1) - 1)))\ldots))\rangle.\end{aligned}$$

6. RECOGNIZING UNIFORMLY IMPLEMENTED LOOPS

Although the problem of recognizing uniformly implemented loops is in general an unsolvable problem, the following guidelines seem useful in a large number of situations.

Recognizing uniformly implemented loops can be viewed as a search for an operation with respect to which the loop is uniformly implemented. In practice, condition (5) is the most demanding constraint on this operation. An effective strategy, therefore, is to use (5) as a guideline to suggest candidate operations. Conditions (6) and (7) must be proved to show that the loop is uniformly implemented with respect to some particular candidate.

Often the modification to the key variable z in the loop body is performed by a statement of the form

$$z := z \# e(x)$$

for some dyadic operation # and some function e. In this case, condition (5) suggests the loop may be uniformly implemented wrt # or some directly related operation. For example, if # is associative, condition (5) holds for #. If # satisfies

$$(a \# b) \# c = (a \# c) \# b$$

(e.g., subtraction), and an inverse $\#'$ of # exists satisfying

$$a \# b = c \leftrightarrow b \#' c = a$$

(e.g., addition if # is subtraction), condition (5) holds for $\#'$.

Another case commonly occurs when the future values of the key variable z are independent of x, that is,

$$h'(z, x1) = h'(z, x2)$$

for all z, $x1$, and $x2$. This situation arises most frequently when z is some data structure which varies dynamically as the loop iterates. Typically, there exists some particular aspect or portion of the data structure (e.g., the top of a stack, the end of a sequence, the leaf nodes in a tree) which guides its modification. A useful heuristic that can be employed in this circumstance is to consider only operations which maintain (i.e., keep invariant) this particular aspect of the data structure. Selecting such an operation $\$'$ guarantees that the "change" experienced by the data structure in the loop body will be independent of any modification $\$'$ and thus ensures that condition (5) will hold.

In any case, recognizing uniformly implemented loops and determining the operation with respect to which they are uniformly implemented is often facilitated when the intended effect of the loop body (as regards the key variable) is documented in the program source text. Such documentation abstracts what the loop body does from the method employed to achieve this result and thus makes analysis of the loop as a whole easier.

To illustrate, consider the following program to compute the maximum value in a subarray $a[i \,..\, n]$ of natural numbers:

```
{$m = 0$}
while $i \le n$ do
  if $m < a[i]$ then $m := a[i]$ fi;
  $i := i + 1$
  od
{ $m = MAXIMUM(a, i0, n)$}.
```

If the effect on m in the loop body were documented as

$$m := MAX(m, a[i])$$

its updating would be of the form $m := m \# a[i]$ and the heuristic discussed above could be employed to help determine that the loop is uniformly implemented wrt $\# = MAX$.

7. RELATED WORK

The first work on generalizing functional specifications for loops appears in [4]. These results are refined in [10] and are studied in considerable detail in [11]. The major contribution of this research seems to be the identification of two loop classes or schemas which are "naturally provable." The first class is called the accumulating loop schema and can be viewed as a (commonly occurring) special case of the Type A loops discussed here. Specifically, a program in the accumulating loop schema with associative binary operation $\$'$ in the sense of [4] is necessarily uniformly implemented wrt $\$'$ and meets the criterion for a Type A loop presented here.

The second of these classes is called the structured data schema. A loop in this class is uniformly implemented with respect to an operator which adds an element to the data structure being processed in such a way that it is not the "next" element to be removed from the structure (e.g., recall the use of *ADDUNDER* in the tree traversal example). A loop in this class necessarily meets the criterion for a Type *B* loop presented here. The program to compute Ackermann's function does not fit in the structured data schema. We remark that the analysis presented here relies on the loop body computing a function, i.e., it relies on the loop body being deterministic. Consequently, the above comments do not apply to the nondeterministic structured data loops analyzed in [11].

In [11] Misra states that the important common feature between these program classes is that "... they act upon data in a 'uniform' manner; changes in the input data lead to certain predictable changes in the result obtained." The work we have described can be viewed as an attempt to characterize this commonality and to generalize the work in [11] based on this characterization.

More recently, Basu in [2] considers the problem of generalizing loop specifications and uses the idea of a loop being "uniform over a linear data domain." One difference between Basu's work and that presented here is that Basu considers only programs in the accumulating loop schema (in the sense of [4] without the closure requirement). More importantly, Basu's idea of uniform behavior is based on the behavior of the loop as a whole and seems to be largely independent of the loop body. Our approach relies solely on the characteristics of the loop body.

Misra points out in [10, 11] that the iteration condition for his structured data schema can be simplified in a manner similar to that presented here; our results show that the same simplification can be applied to his accumulating loop schema. Again, an appropriate view of our research is one of generalizing this earlier work by investigating the theory which underlies these phenomena.

8. SUMMARY AND CONCLUSIONS

It is felt that a critical aspect of reading, understanding, and verifying program loops is generalizing the behavior of a loop over a restricted set of inputs to that over a more general set of inputs. The view of the generalization process presented here is one of ascertaining how changes in values of particular input variables affect the subsequent computation of the loop. This process is facilitated if these changes correspond to particularly simple modifications in the result produced by the loop body.

Of course, the simplest possible modifications in the result produced by the loop body would be no modification at all, i.e., when the output of the loop body (and hence the loop) is completely independent of changes in these input variables. This situation, however, rarely occurs in practice since it implies that the input values of these variables serve no purpose in view of the intended effect of the loop. It is felt that the definition of a uniformly implemented loop presented here is the "next best" alternative, and yet a large number of commonly occurring loops seem to possess this property. The definition states that in terms of the execution of the loop body, prescribed changes in the input value of the key variable affect only the final value of the key variable; all other final values are independent of the change. Just as importantly, the modification caused in the final value of the key variable is necessarily the same as the change in its corresponding input value. This property is analogous to that possessed by a function of one variable with unit slope in analytic geometry: increasing the input argument by some constant causes the function value to be increased by exactly the same quantity. Taken together, these factors account for the pleasing symmetry between \$ and H in condition (8).

Viewed as a verification technique for uniformly implemented loops, the procedure described here can be thought of as transforming the problem of discovering the general loop specification into the problem of discovering the operation with respect to which the loop is uniformly implemented. Clearly, this is of no benefit if the latter is no easier to solve than the former. In many cases, however, it seems that simple syntactic checks are sufficient for identifying this operation. For example, in the tree traversal program, the fact that the loop body does not test the stack for emptiness [4] is a sufficient condition for the loop being uniformly implemented with respect to *ADDUNDER*.

It is felt that the notion of uniformly implemented loops may have an application in the program development process. Specifically, when designing an initialized loop to compute some function, the programmer should attempt to construct the loop in such a way that it is uniformly implemented with respect to some easily stated operation. Our work indicates that these loops are susceptible to a rather routine form of analysis. Furthermore, implementing a loop in a uniform fashion requires maintaining a certain amount of independence between program variables (or perhaps portions of program variables in the case of structures) and a simple dependence between the input/output values computed by the loop body. Such programs are desirable since the ease with which a loop can be understood depends largely on the complexity of the interactions and interconnections among program variables. We remark that the question of whether a given program is "well structured" has been viewed largely as a *syntactic* issue (e.g., use of a restricted set of control structures); we offer the definition of a uniformly implemented loop as an attempt at a characterization of a *semantically* well-structured program.

REFERENCES

1. Basili, V.R., and Noonan, R.E. A comparison of the axiomatic and functional models of structured programming. *IEEE Trans. Softw. Eng. SE-6* (Sept. 1980), 454–465.
2. Basu, S. A note on synthesis of inductive assertions. *IEEE Trans. Softw. Eng. SE-6* (Jan. 1980), 32–39.
3. Basu, S.K., and Misra, J. Proving loop programs. *IEEE Trans. Softw. Eng. SE-1* (March 1975), 76–86.
4. Basu, S.K., and Misra, J. Some classes of naturally provable programs. In *Proceedings of the 2nd International Conference on Software Engineering* (San Francisco, Oct. 13–15). IEEE, New York, 1976, pp. 400–406.
5. Dunlop, D., and Basili, V.R. A comparative analysis of functional correctness. *Comput. Surv. 14*, 2 (June 1982), 229–244.
6. Gries, D. Is sometime ever better than alway? *ACM Trans. Program. Lang. Syst. 1* (Oct. 1979), 258–265.
7. Hoare, C.A.R. An axiomatic basis for computer programming. *Commun. ACM 12* (Oct. 1969), 576–583.
8. Mills, H.D. Mathematical foundations for structured programming. Rept. FSC 72-6012, IBM Federal Systems Division, Bethesda, MD, 1972.
9. Mills, H.D. The new math of computer programming. *Commun. ACM 18* (Jan. 1975), 43–48.
10. Misra, J. Some aspects of the verification of loop computations. *IEEE Trans. Softw. Eng. SE-4* (Nov. 1978), 478–486.
11. Misra, J. Systematic verification of simple loops. Tech. Rep. TR-97, University of Texas, Austin, Tex., March 1979.
12. Morris, J.H., and Wegbreit, B. Subgoal induction. *Commun. ACM 20* (April 1977), 209–222.
13. Webreit, B. Complexity of synthesizing inductive assertions. *J. ACM 24* (July 1977), 504–512.

Received January 1982; revised October 1983 and July 1984; accepted July 1984