A Generalized Iterative Construct and Its Semantics

ED ANSON
Northeastern University

A new programming language construct, called DOupon, subsumes Dijkstra's selective (IF) and iterative (DO) constructs. DOupon has a predicate transformer approximately equivalent in complexity to that for DO. In addition, it simplifies a wide variety of algorithms, in form as well as in discovery and proof. Several theorems are demonstrated that are useful for correctness proofs and for optimization and that are not applicable to DO or IF. The general usefulness of DOupon derives from a separation of the concerns of invariance, through iteration, from those of termination.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Program Verification—*correctness proofs*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics*; D.3.3 [**Programming Languages**]: Language Constructs—*control structures*

General Terms: Languages, Theory, Verification

Additional Key Words and Phrases: Iterative constructs, predicate transformer

1. INTRODUCTION

Representing the semantics of programs (and program parts) as predicate transformers is useful, not only for proving properties of programs, but for aiding their synthesis as well [2]. When coupled with a powerful set of statements composed of guarded commands, the formalism can be extremely useful.

Guarded statement forms, described by Dijkstra [2], include a purely selective form (IF) and a purely iterative form (DO). A new form (DOupon) subsumes the iterative and selective forms. It provides more expressive power and simplifies both the synthesis and the proofs of algorithms. Furthermore, it reduces by one the number of distinct control structures required for programming.

Although a syntactic form is introduced for DOupon, it is not to be construed as proposed syntax for a practical programming language. Indeed, the form used is probably not particularly useful for programming. (It is nevertheless useful for discussing semantics.) The intent of this paper is to introduce a new semantic form, which can be used to simplify future languages, and to give new insight into the logic of programming.

The revision of this paper was supported by California Computer Products, Inc.
Author's current address: CalComp Display Products Division, C.S. 908, PTP2-2D01, Hudson, NH 03051-0908.

Almost simultaneously with the original submission of this paper, Parnas [7] proposed a construct that is semantically very similar to DOupon. However, it is different in a crucial way. Ironically, Parnas considered a form semantically equivalent with DOupon, but rejected it for supposed efficiency reasons. Consequently, this paper now includes a discussion of Parnas's construct and a detailed comparison of the semantic differences between the two forms. It is shown that DOupon is much simpler semantically and can lead to more efficient implementations than Parnas's construct.

Section 2 informally introduces DOupon, and Section 3 gives a simple example of its usefulness. Section 4 formally defines the predicate transformer for DOupon. Section 5 demonstrates two theorems, which indicate unusual flexibility of the construct in the design of algorithms, and Section 6 illustrates this flexibility with an example. Section 7 demonstrates some theorems related to optimizing transformations, unique to DOupon, and Section 8 applies some of those theorems to solve a programming problem.

2. SYNTAX AND INFORMAL SEMANTICS OF A NEW ITERATIVE CONSTRUCT

The statement forms defined by Dijkstra [2] include the IF and DO, briefly described as follows:

$$\textbf{IF } Q_1 \rightarrow M_1 \,\square\, Q_2 \rightarrow M_2 \,\square \ldots \square\, Q_n \rightarrow M_n \textbf{ FI} \qquad \text{(IF)}$$

and

$$\textbf{DO } P_1 \rightarrow L_1 \,\square\, P_2 \rightarrow L_2 \,\square \ldots \square\, P_m \rightarrow L_m \textbf{ OD} \qquad \text{(DO)}$$

In these statements, the P_i and Q_i are predicates defined on the state space of the program and are known as guards. The L_i and M_i are statements of any kind.

Informally, IF executes by first determining which of the guards are true. If one or more guards are true, one of the true guards (Q_i) is selected arbitrarily and the corresponding statement (M_i) is executed. If none of the guards is true, the statement fails. In the case where more than one guard is true, the statement is nondeterministic. Note that the guard selection is not necessarily random. The nondeterminism simply implies that the programmer cannot know which true guard will be chosen. For example, an implementation which chooses the first true guard found would be as valid as a totally random choice.

The DO statement executes in a manner similar to the IF, in that any one of the true guards may cause its corresponding statement to be executed. However, if this occurs, the entire statement is then executed again. When no guard is true, the statement terminates normally. Again, if more than one of the guards can be true at any given time, the statement is nondeterministic.

The new statement form, DOupon, combines the features of IF and DO:

$$\begin{aligned}&\textbf{DO} \quad P_1 \rightarrow L_1 \,\square\, P_2 \rightarrow L_2 \,\square \ldots \square\, P_m \rightarrow L_m\\ &\textbf{UPON } Q_1 \rightarrow M_1 \,\square\, Q_2 \rightarrow M_2 \,\square \ldots \square\, Q_n \rightarrow M_n \textbf{ OD}\end{aligned} \qquad \text{(DOupon)}$$

As a notational convenience throughout this paper, we consistently use P and Q, and L and M, to refer to the iterative and selective sets of guards and statements, respectively. We also use $\textbf{IF}_P$ to represent an IF statement composed

of the P guards, whether taken from a DO or DOupon. We use IF_Q to represent an IF statement composed of the Q guards from a DOupon.

Informally, the DOupon statement executes as follows: If any of the guards Q_i are true, execution is as in IF_Q. That is, the statement terminates after executing one of the guarded statements from the UPON clause. Otherwise, if one or more of the guards P_i are true, a corresponding statement L_i is executed and the entire statement repeats. Accordingly, the guards P_i are called iterative guards, and the guards Q_i are called the terminating guards. If no guard is true, the statement fails.

Unlike the DO statement, DOupon can terminate even if some iterative guards are true. The termination condition is stated positively, rather than negatively as in DO. Furthermore, when termination occurs, an action is immediately selected that is appropriate to the particular reason for termination. These properties, rather than the combination of forms in itself, result in the special usefulness of DOupon.

Parnas [7] proposes a construct similar to DOupon. Although he uses different syntax and calls the structure IT-TI, it is given a similar syntax to DOupon here, to avoid issues irrelevant to semantics. It will be called DOterm here, by analogy to DOupon. It thus looks like this:

DO $P_1 \rightarrow L_1 \,\square\, P_2 \rightarrow L_2 \,\square \ldots \square\, P_m \rightarrow L_m$
TERM $Q_1 \rightarrow M_1 \,\square\, Q_2 \rightarrow M_2 \,\square \ldots \square\, Q_n \rightarrow M_n$ OD (DOterm)

A "term" clause is used here, instead of "upon," to signify a different interpretation of the termination guards. Parnas allows that execution may terminate if one of the Q guards is true, but if one of the P guards is also true, execution may either continue or terminate, nondeterministically. Unlike DOupon, an iterative guard may be chosen regardless of the truth of the termination guards.

Parnas argues that such a form would be more efficient than DOupon, since it is not necessary to evaluate all the termination guards of DOterm in order to continue iteration. In a highly parallel processor, this might sometimes be an advantage. However, it also increases the nondeterminism of the iteration, possibly causing it to continue long after it could have terminated in a desirable state. It seems prudent (at least in some cases) to terminate as soon as a termination guard indicates that termination is valid.

It is particularly worth noticing that, in any sequential implementation, the guards must be evaluated in some order. In such a case, the supposed efficiency of DOterm is not realized, but its limitations are. Of course, DOterm can be implemented with termination guards evaluated first. But, in such an implementation, its implementation is indistinguishable from DOupon, while the semantic limitations remain. As is shown in Section 4, the semantics of DOupon would generally be preferred.

3. A SIMPLE EXAMPLE

A common occurrence in programming is that, when an iteration terminates, one of two or more conditions is known to hold, and a different action is required for each. A simple example is a sequential search of a small array, with different actions taken for each of the two (or more) keys, or for failure.

The following program fragment searches the first n elements of an array named "*list*" for the first element containing either of two specified keys. If it finds either key, a procedure is executed, which depends on which key is found. If neither key is in the array, a message to that effect is printed. To simplify the algorithm, we assume the array is initialized so that $list[n + 1]$ contains some value that is not equal to either key. Although this assumption is not necessary, it helps us to avoid distracting details.

```
i := 1;
DO i ≤ n                  → i := i + 1
UPON list[i] = Key1 → TypeOneAction(Key1, i)
  □ list[i] = Key2  → TypeTwoAction(Key2, i)
  □ i > n           → print("Search Failed")
OD
```

The reader may have noticed that, since $i > n$ causes termination, the iteration guard ($i \leq n$) can be weakened to **true**, without affecting the correctness of the program. The guard is given in its present form, as kind of documentation of the fact that only $n + 1$ elements are guaranteed to be in the array, and thus it is meaningless (incorrect) to advance i past that value. In Section 7, we discuss in more detail the conditions under which guards may be weakened or strengthened. In particular, the corollary to Theorem 4 applies to the present example.

At this point, it is worth observing that, with conventional iterative statements, this program is somewhat more complicated. An equivalent solution, using DO and IF, follows:

```
i := 1;
DO (i ≤ n) and (list[i] ≠ Key1) and (list[i] ≠ Key2) → i := i + 1 OD;
IF  list[i] = Key1 → TypeOneAction(Key1, i)
  □ list[i] = Key2 → TypeTwoAction(Key2, i)
  □ i > n          → print("Search Failed")
FI
```

This program requires an extra evaluation of the predicates upon termination. Many programmers find this objectionable for esthetic reasons as well as for reasons of efficiency. It also tends to complicate proofs of correctness. It is worth noting that none of the guards can be weakened in this version. A similar structure would be required by DOterm, in which IF is replaced by TERM. The simpler structure, used for DOupon, would not generally terminate correctly if DOterm semantics were applied, since iteration could continue after a key is found.

This example is one of a large class of problems, which require either reevaluation of predicates or the setting of flags, in order to solve the problem with conventional statement forms. This fact has led others to propose alternative loop forms, with exits at arbitrary points [1, 3, 8]. However, the semantics of such multiexit loops are difficult to define accurately, since they suffer from a similarity to the **goto**. DOupon solves the problem cleanly, while only slightly restricting the ability to exit from a loop.

The difficulty of defining the semantics of **goto** is fully explained in [6] and arises from the fact that execution does not pass to the next statement of the program. Hoare-type logics, as well as weakest precondition semantics, implicitly

assume that control passes from each statement directly to the next. **goto** violates this assumption, leading to a breakdown of the reasoning systems. Multiexit loops (and exception mechanisms) have a similar anomaly at the exit points. Although the Hoare-type semantics of a single multiexit loop may be defined adequately in some cases, nested structures cause problems.

As shown in the next section, the semantics of DOupon are well defined and are as simple as those for DO and IF. Proofs of correctness may even be simpler in many cases.

4. FORMAL SEMANTICS OF THE ITERATIVE CONSTRUCT

The semantics of DOupon are described by its weakest precondition function, in a manner similar to that used by Dijkstra [2] to define IF and DO.

For any statement S, and predicate R, the function wp(S, R) yields the weakest precondition predicate guaranteeing that S will terminate normally, in a state satisfying R. DOupon is a statement of the form

DO $P_1 \rightarrow L_1 \,\square\, P_2 \rightarrow L_2 \,\square \ldots \square\, P_m \rightarrow L_m$
UPON $Q_1 \rightarrow M_1 \,\square\, Q_2 \rightarrow M_2 \,\square \ldots \square\, Q_n \rightarrow M_n$ OD

As a notational convenience, we define $PP \equiv (\exists j{:}1 \leq j \leq m{:}\ P_j)$, and $QQ \equiv (\exists j{:}1 \leq j \leq n{:}\ Q_j)$, to refer to combinations of the iterative and terminating guards, respectively.

As defined in [2, p. 34], Dijkstra's IF statement is of the form

IF $Q_1 \rightarrow M_1 \,\square\, Q_2 \rightarrow M_2 \,\square \ldots \square\, Q_n \rightarrow M_n$ FI

and is described by

$wp(\text{IF}, R) = QQ$ and $(\forall j{:}1 \leq j \leq n{:}\ Q_j \Rightarrow wp(M_j, R))$.

The DOupon statement is described recursively, using a function H_k(DOupon, R), which is the weakest precondition guaranteeing that DOupon terminates in k or fewer iterations, yielding a state which satisfies the predicate R. Since DOupon terminates immediately if and only if one or more of the Q guards is true, we have

$H_0(\text{DOupon}, R) = wp(\text{IF}_Q, R)$.

Since each successive iteration reduces by one the limit implied by H_k, it follows that H_k(DOupon, R) is true if and only if execution of any one of the P guarded statements yields a state satisfying H_{k-1}(DOupon, R), or if the statement terminates immediately. Therefore, for any $k > 0$,

$H_k(\text{DOupon}, R) = wp(\text{IF}_P, H_{k-1}(\text{DOupon}, R))$ or $H_0(\text{DOupon}, R)$

Given this definition, we define

$wp(\text{DOupon}, R) = (\exists k{:}k \geq 0{:}\ H_k(\text{DOupon}, R))$

This definition is essentially different from that of DO [2, p. 35], only in the definition of H_0. For DO, H_0(DO, R) = (R and non PP). DOupon is thus slightly more complex than DO because it also includes the equivalent of IF. However, in those cases where the DO would be followed by an IF, DOupon is actually simpler

in the aggregate. Where the subsequent IF is not needed, DOupon reduces to

DO $P_1 \rightarrow L_1 \square P_2 \rightarrow L_2 \square \ldots \square P_m \rightarrow L_m$
UPON non PP → **skip**
OD

which is semantically identical with DO. In an actual programming language, it would probably be useful to introduce syntactic sugar for the phrase non PP. In this context, the value is trivial to compute, once all the P_i have been computed.

Note, also, that a DOupon statement with an empty iterative part is equivalent with IF. Since DO and IF are degenerate forms of DOupon, the semantics of DOupon need never be more complex than for DO and IF. Indeed, they are often simpler.

On the other hand, the semantics of DOterm are considerably more complex. Parnas [7] describes the semantics of his construct, using set theory. Here, the equivalent weakest precondition semantics are given, to facilitate comparison of DOterm with DOupon.

Since the iterative part of DOterm may continue whenever any P guards are true, termination is assured only when all are false. This introduces additional complexity into the base of the recursive definition, which thus becomes

H_0(DOterm, R) = non PP and wp(IF$_Q$, R).

For $k > 1$, the H_k expression describes iteration for the P guards and termination for the Q guards. We thus obtain

H_k(DOterm, R) = H_0(DOterm, R) or ((PP or QQ) and (PP $\Rightarrow$ wp(IF$_P$, H_{k-1}(DOterm, R)) and (QQ $\Rightarrow$ wp(IF$_Q$, R))

However, since $H_0 \Rightarrow$ QQ, this may be simplified to

H_k(DOterm, R) = (PP or QQ) and (PP $\Rightarrow$ wp(IF$_P$, H_{k-1}(DOterm, R))) and (QQ $\Rightarrow$ wp(IF$_Q$, R)).

The most obvious consequence of this is that weakest precondition expressions involving DOterm are intrinsically more difficult to evaluate than are expressions involving DOupon. Less obvious is the fact that the weakest precondition for DOterm is much stronger than for DOupon, which seems to imply a less powerful construct. Indeed, Parnas [7] alludes to the fact that his construct makes termination more difficult to obtain.

5. THE BASIC THEOREMS FOR DOupon

The weakest precondition, defined in Section 4, is not directly useful for programming. That is, a programmer does not generally determine the weakest precondition for each program part. Instead, he or she uses a bag of tricks, accumulated over years of experience, to determine the most useful way to implement a desired program. Each trick from the bag is, in effect, a theorem abut the behavior of some program schema. In order to develop such theorems in a disciplined manner, it is necessary to have methods that are readily applicable.

This section demonstrates several theorems that apply to useful programming techniques. Additional theorems are given in Section 7. All theorems are given with informal proofs only. Indeed, what is given amounts to a sketch of each

proof, intended to suggest the plan for a more detailed proof. Such detailed proofs are left as exercises for the ambitious reader.

Dijkstra's basic theorem for the DO construct [2, p. 38] provides a useful way to demonstrate desired properties of DO or to discover a correct iterative algorithm. Given a predicate V (an invariant condition), and the knowledge that (V and PP) $\Rightarrow$ wp(IF$_P$, V), we can conclude that (V and wp(DO, true)) $\Rightarrow$ wp(DO, V and non PP). Putting it another way: Given that (V and PP) $\Rightarrow$ wp(IF$_P$, V) and also that, for some predicate R, (V and non PP) $\Rightarrow$ R, we may conclude (V and wp(DO, **true**)) $\Rightarrow$ wp(DO, R). That is, given the invariant condition, only termination need be demonstrated to prove total correctness.

A similar result can be shown for DOupon.

THEOREM 1. *Given* (V *and non* QQ) $\Rightarrow$ wp(IF$_P$, V), *and* (V *and* QQ) $\Rightarrow$ wp(IF$_Q$, R), *we may conclude* (V *and* wp(DOupon, **true**)) $\Rightarrow$ wp(DOupon, R).

The proof is similar to Dijkstra's proof for DO. The first antecedent guarantees that V is invariant within the iterative part of DOupon. The invariance of V, together with the second antecedent, guarantees that termination leads to R, since termination occurs if and only if QQ holds. Therefore, the conjunction of V with a guarantee of termination guarantees that DOupon will terminate in a state satisfying R.

Another result follows for DOupon, which has no useful analog for DO or DOterm.

THEOREM 2. *Given* V $\Rightarrow$ wp(IF$_P$, V), *and* (V *and* QQ) $\Rightarrow$ wp(IF$_Q$, R), *we may conclude* (V *and* wp(DOupon, **true**)) $\Rightarrow$ wp(DOupon, R).

This follows directly from Theorem 1 because the first antecedent is stronger.

This result differs from Theorem 1 only because the first antecedent does not include non QQ. This implies a stronger invariant condition than Theorem 1 because Theorem 2 allows the invariance to fail when QQ holds.

Although Theorem 2 follows directly from Theorem 1, it is stated separately to emphasize an important property of DOupon. Notice that the equivalent theorem for DO (omitting the second antecedent, which cannot apply) would not be useful. DO requires the weaker form of invariance, as stated by Dijkstra. That is, since (for DO) wp(IF$_P$, V) $\Rightarrow$ PP, the antecedent V $\Rightarrow$ wp(IF$_P$, V) yields V $\Rightarrow$ PP, and thus wp(DO, **true**) $\Rightarrow$ non V. Thus, the conclusion becomes the vacuous statement (V and non V) $\Rightarrow$ wp(DO, R). Put more simply, the antecedent of Theorem 2, applied to DO, would guarantee that DO does not terminate.

DOterm satisfies Theorem 1 as well, but termination is more difficult to establish. Furthermore, Theorem 2 is vacuous for DOterm, since the stronger antecedent rules out any guarantee of termination. The reasoning, in this case, follows that for DO.

For some classes of problems, the stronger invariant allowed by Theorem 2 is useful, as is illustrated by the example in the next section.

6. ANOTHER EXAMPLE

Let us suppose that we are designing a memory management system. Let us futher suppose that the nature of the application is such that if, at any time, an

allocation request exceeds the amount of available memory, the application fails. This is realistic, for instance, in a transaction-oriented system, where (at the very least) a transaction must fail if it cannot obtain sufficient memory.

A reasonable way to evaluate a design for our memory manager would be to simulate its behavior under a random sequence of requests supposed to represent a realistic load. To do this, we create a data structure that represents the memory map together with statistics about its history. Then we create a time-ordered even queue. Allocation and release events are simulated until a specified simulated time elapses, or an allocation fails. The results are reported.

The following is a sketch of a program for the simulation, which assumes the following: Allocation requests arrive at random intervals and allocate random amounts of memory. Once allocated, a block of memory is released to the pool in a random amount of time. We will not concern ourselves here with distributions of the random functions, or even with the details of the allocation strategy or of desirable statistics.

```
set memory map to a starting configuration;
schedule first request;
DO    next event a request → simulate allocation;
                             schedule release;
                             schedule next allocation
  □   next event a release → simulate release
UPON  next event a request > MemAvail → print failure notice
  □   next event has time > t → print statistics
OD
```

In this program skeleton, the iterative part is essentially a simulator of the memory management system. The termination guards simply indicate conditions under which the simulation is to be terminated. We are assuming here that the simulator is correct. For example, it must maintain a variable *MemAvail* that reflects the remaining available simulated memory. Simulating an event is assumed to remove it from the event queue, and scheduling an event adds an event to the time-ordered queue, with *time* greater than the current simulated time.

The invariant condition, maintained by the iteration, is simply a correspondence between the data and the state of the simulated system. In particular, the memory map represents the result of all simulated events, and the first event on the queue is the next that would occur in the real system. This condition is trivially established prior to entering DOupon, and the invariance follows, if the simulator is correct. Given this condition, either of the two termination conditions leads to the correct results, as can be easily seen. By Theorem 2, the program is partially correct. Termination (and thus total correctness) follows from the observation that simulated time is monotonically increasing [2, p. 41].

What is most significant is that the use of either DO or DOterm for the simulation would require incorporation of the *MemAvail* test and/or the time test into each of the iterative guards. This is not only cumbersome, but would be unnatural, since those conditions have nothing to do with the logic of the simulation itself. Furthermore, the tests would still have to be made again, upon termination, in order to produce the desired output. On the other hand, the DOupon version would allow the same iterative part to be used with different

termination conditions, which need not necessarily involve time or available memory. Such advantages, due to DOupon, would tend to increase with the overall complexity of a simulation.

The DOupon thus shows its usefulness, by separating the concerns of invariance from those of termination. The example given is typical of a wide class of converging iterative algorithms. For example, a Taylor series, a Newton iteration, or a binary search of the real line, are potentially unending sequences. Nevertheless, each converges toward a solution. It is semantically convenient to specify the (arbitrary) termination criterion separately from the iteration guards. DO would require the effective inclusion of the termination guards in each of the iterative guards, in order to assure termination. DOterm would also require inclusion of termination criteria in the iterative guards in order to prevent further iteration after convergence is achieved. In such cases, the supposed comparative efficiency of DOterm is not realized, because the termination criteria must still be computed on each iteration.

7. SOME OPTIMIZATION RESULTS

A useful strategy for discovering a good algorithm is as follows: First, devise a simple algorithm, known to be correct. Then apply safe transformations which improve its efficiency. Throughout these steps, a nondeterministic algorithm is at least notationally useful, since it does not unnecessarily constrain choices. The semantics of the iterative construct are consistent with many possible sequential implementations, one of which may be chosen based on efficiency considerations. This process is illustrated by the example in Section 8.

DOupon offers unique optimization transformations, due to the fact that the iteration and termination conditions may be manipulated more or less independently. In particular, we show in this section that strengthening or weakening the guards is safe, under a wide range of conditions, and can lead to a better algorithm.

THEOREM 3. *If the correctness of a* DOupon *statement has been established by means of Theorem* 1 *or Theorem* 2, *then strengthening one or more of the iterative guards preserves total correctness, provided that* (V *and non* QQ) $\Rightarrow$ PP *still holds.*

Observe that the only antecedent of either theorem, affected by strengthening an iterative guard, is the first, namely, (V and non QQ) $\Rightarrow$ wp(IF$_P$, V). In particular, the value of wp(IF$_P$, V) may be affected. However, wp(IF$_P$, V) = (PP and ($\forall j$:1 $\leq j \leq$ m: $P_j \Rightarrow$ wp(L_j, V))). Strengthening any P_j cannot negate the truth of the second part, so only PP need be established again, as stated by the Theorem.

Notice that, since wp(IF$_P$, V) $\Rightarrow$ PP, verifying the proviso of Theorem 3 generally amounts to less work than reestablishing the antecedent of Theorem 1.

Notice also that this result applies to total correctness, as well as to partial correctness. That is, if Theorem 1 or 2 has established partial correctness, and other means have established termination, strengthening the iterative guards preserves termination under the conditions of Theorem 3. This is because such

strengthening of iterative guards can only reduce the nondeterminism of the execution. That is, any sequence of execution possible with the strengthened guards was also possible with the weaker guards. If such sequences were correct before, they still are. The proviso of Theorem 3 is sufficient to assure that at least one guard is true at the beginning of each iteration.

Strengthening the iterative guards can lead to quicker convergence of an iteration, by effectively choosing (from the correct alternatives) a statement that makes more progress than the others. The fact that this can be done without affecting the termination condition is convenient. The DO statement does not have this property, since strengthening its guards can weaken its termination condition, allowing the DO to terminate in an incorrect state.

COROLLARY. *If the antecedents of Theorem* 1 *are satisfied, but termination of the* DOupon *cannot be established, strengthening one or more iterative guards may help establish total correctness.*

With the given conditions, strengthening of the guards preserves partial correctness. In many cases, the first form of an algorithm may fail to converge because one or more of the guards are too weak. That is, some steps may be possible that make no progress toward the desired state, or that actually diverge. It will often be sufficient simply to strengthen one or more guards to prevent this. In the extreme case, a guard may become false, and the corresponding statement may thus be removed.

THEOREM 4. *If the partial correctness of a* DOupon *statement has been established by means of Theorem* 1 *or Theorem* 2, *then weakening one or more of the iterative guards preserves partial correctness, provided that* (V *and non* QQ) $\Rightarrow$ wp(IF$_P$, V) *still holds.*

The proof of this theorem is similar to that for Theorem 3. Weakening an iterative guard only affects the first antecedent of Theorem 1 or 2, and so only that antecedent must be reestablished. However, since the guard is being weakened, the entire invariance must be demonstrated anew. Furthermore, weakening the guards also requires that termination be established anew, due to the converse of the Corollary to Theorem 3.

It would at first appear that this theorem has very limited usefulness. Nevertheless, it is useful, as illustrated in Section 8. In particular, speedup can occur due to simplification of the guards, and in many cases it is not difficult to show that the invariance and termination arguments used for the original algorithm apply as well to the new one. If only one guard is changed at a time, the following corollary may prove useful.

COROLLARY. *If the partial correctness of a* DOupon *statement has been established by means of Theorem* 1 *or Theorem* 2, *then weakening any iterative guard* P_i *to* P_i' (*i.e.*, $P_i \Rightarrow P_i'$) *preserves partial correctness, provided that* (V *and non* QQ *and* P_i') $\Rightarrow$ wp(L_i, V) *still holds. A special case of this corollary is particularly useful. When a guard* P_i *is weakened to* P_i', *such that* (P_i' *and* (V *and non* QQ)) = P_i, *total correctness is preserved.*

Proof of this corollary is left to the reader.

THEOREM 5. *If the correctness of a* DOupon *statement has been established by means of Theorem* 1 *or Theorem* 2, *then weakening one or more of the termination guards preserves total correctness, provided that* (V *and* QQ) $\Rightarrow$ wp(IF$_{Q}$, R) *still holds.*

Weakening the termination guards affects both antecedents of Theorem 1, and only the second antecedent of Theorem 2. However, weakening QQ strengthens the left side of the implication in the first antecedent and cannot make that antecedent false. The second antecedent, in both cases, is simply the proviso of Theorem 5.

COROLLARY. *If the correctness of a* DOupon *statement has been established by means of Theorem* 1 *or Theorem* 2, *then weakening any termination guard* Q_i *preserves total correctness, provided that* (V *and* Q_i) $\Rightarrow$ wp(M_i, R) *still holds.*

This follows directly from Theorem 5. Since weakening a termination guard also weakens QQ, it is necessary to show that the proviso of Theorem 5 still holds in the additional states allowed by the weakened QQ. However, only the weakened guard can be selected in those additional cases, and thus it is sufficient to show that the corresponding statement gives the desired result. This is the proviso of the corollary.

Again, these results apply to total, as well as partial, correctness. Weakening QQ can cause earlier termination but cannot prevent termination already demonstrated. If it is discovered that the antecedents of these results apply, earlier termination is often obtained by making the transformation. This is because a weaker termination condition is easier to converge upon.

THEOREM 6. *If the partial correctness of a* DOupon *statement has been established by Theorem* 1 *or Theorem* 2, *then strengthening one or more termination guards* Q_i *preserves partial correctness.*

Since the strengthened termination guards imply the original ones, the antecedents of Theorem 1 (or Theorem 2) still hold.

Since it may be more difficult for the iterative part of the loop to satisfy the new (stronger) termination criterion QQ, new proof of termination is required. As with Theorem 4, this theorem may appear to have limited usefulness. However, it sometimes happens that the original termination argument applies as well to the strengthened requirements, and that the stronger criterion is more efficiently computed. Such a case is illustrated by the example in the next section.

Generally, we have shown in this section that the guards of DOupon may be strengthened or weakened under a wide range of conditions, without affecting the correctness of a program. Similar properties do not usefully apply to DO or DOterm. Strengthening an iterative guard of a DO statement simultaneously weakens the termination criteria, and vice versa. DOterm suffers from a similar problem, though in a different way. Weakening an iterative guard may prevent termination of DOterm, whereas it cannot do so to DOupon. However, apart from termination problems, DOterm behaves similarly to DOupon, with respect to the transformations discussed in this section.

8. A THIRD EXAMPLE

A problem, which often arises in computer graphics, may be stated as follows. Given a line in the plane, from point A = (X_0, Y_0) to B = (X_1, Y_1), draw that part of the line (if any) which passes through an upright rectangular window bounded by $X_{min} \leq x \leq X_{max}$ and $Y_{min} \leq y \leq Y_{max}$. That is, if the line passes through the window, draw the part within the window; otherwise draw nothing.

The simplest form of the algorithm that solves this problem is

```
DO    some part of line outside window → remove at least some of the part outside
UPON  line entirely inside window → draw the line
  □   line entirely outside window → skip
OD
```

The invariant condition, which is maintained by the iterative part of this algorithm, is that the line includes every part of the original line that passes through the window, and includes nothing that was not part of the original line. Clearly, this condition is maintained, since only those parts of the line outside the window are removed. Certainly, if the invariant holds, and the remaining line is entirely inside the window, what gets drawn is exactly that part of the original line that was inside the window. Likewise, if no part of the remaining line is in the window, none of the original was, so we draw nothing.

Termination of this algorithm can be demonstrated by observing that the total length of the line remaining outside the window is monotonic decreasing and that the algorithm terminates when that amount reaches zero. Details of the proof are left as an exercise for the reader. For a hint, see [2, p. 41].

Note that the algorithm, in this form, would not be even partially correct with DOterm semantics applied. This is because the iterative part could remove all of the line, making all guards false. The DOupon version prevents this from occurring, by assuring termination in such a state. A correct algorithm, using either DO or DOterm, would require more complex guards.

We now proceed to elaborate on this algorithm, and improve it, until a practical algorithm is found.

Since the window is convex, a line passing through an edge (passing from the inside to the outside) does not reenter the window. It would thus suffice for the iterative guard to determine whether the line crosses any boundaries of the window. Where it does, the part outside the window can be eliminated. The following algorithm implements this strategy and substitutes an equivalent set of guarded statements for the strengthened iterative guard. Although the combined effect of its iterative guards is somewhat stronger than in the original algorithm, Theorem 3 assures us that the resulting algorithm is still correct:

```
DO    line crosses left boundary   → remove part to the left
  □   line crosses right boundary  → remove part to the right
  □   line crosses top boundary    → remove part above
  □   line crosses bottom boundary → remove part below
UPON  line entirely inside window  → draw the line
  □   line entirely outside window → skip
OD
```

The algorithm is clearly still correct, but testing for intersections with the rectangle's boundary is not computationally efficient enough for a useful algorithm. Theorem 4 permits us to weaken the iterative guards to a form which is more quickly computed. To do this, we extend each of the four window boundaries to infinite lines, each dividing the plane into two parts. It is easy to determine which half plane any point lies in, and if the two ends of the line lie in opposite half planes, the line crosses the extended boundary. Furthermore, if a line crosses none of these lines, it certainly does not cross the window's boundary. This leads to the following algorithm

```
DO   line crosses Xmin                  → remove part to left
 □   line crosses Xmax                  → remove part to right
 □   line crosses Ymin                  → remove part below
 □   line crosses Ymax                  → remove part above
UPON line entirely inside window        → draw the line
 □   line entirely outside window       → skip
OD
```

The corollary to Theorem 4 shows that this transformation is safe, since only the part of the line outside the window is ever removed. Although termination must be reestablished, it is easily demonstrated in the same manner as for the original algorithm. We now have a somewhat more efficient algorithm, which is still known to be correct. However, the second termination condition is still inefficient to compute.

To further improve the efficiency of the algorithm, we use the observation that, if the line is entirely on the outside of any particular extended boundary (e.g., left of X_{min}), it is also entirely outside the window. We also observe that the line is entirely inside the window if and only if it is on the inside of all extended boundaries (e.g., right of X_{min}). We thus strengthen the second termination guard, and reword the first, obtaining

```
DO   line crosses Xmin                          → remove part to left
 □   line crosses Xmax                          → remove part to right
 □   line crosses Ymin                          → remove part below
 □   line crosses Ymax                          → remove part above
UPON line entirely inside all extended boundaries → draw line
 □   line entirely outside some extended boundary → skip
OD
```

According to Theorem 6, this transformation preserves partial correctness. Termination can again be established by the same means as before. Even though a few more iterations may now be required in order to terminate, each iteration costs less, and the overall performance is improved. As a final optimizing step, we observe that, if the line is entirely inside (or entirely outside) any boundary, it does not cross it. Furthermore, the termination guards cover all cases where no boundary is crossed. It therefore follows from Theorem 3 that the iteration guards can be strengthened (and thus simplified) by making use of the assumption

that a boundary is crossed, yielding

```
DO  X0 < Xmin  → move A to Xmin
  □ X1 < Xmin  → move B to Xmin
  □ X0 > Xmax  → move A to Xmax
  □ X1 > Xmax  → move B to Xmax
  □ Y0 < Ymin  → move A to Ymin
  □ Y1 < Ymin  → move B to Ymin
  □ Y0 > Ymax  → move A to Ymax
  □ Y1 > Ymax  → move B to Ymax
UPON line entirely inside all extended boundaries   → draw line
  □  line entirely outside some extended boundary → skip
OD
```

As a notational convenience, the phrase "move A to X_{min}" is used to indicate that the coordinates of point A are changed to those where the line crosses the X_{min} boundary. Under the indicated condition, this has the effect of removing the part of the line outside the boundary.

Implementation of this form of the algorithm generally makes use of two small sets (or bit arrays), which record which of the four extended boundaries each of the end points is outside of. The iteration guards then reduce to checking set membership. The first termination guard is simply a test of whether the union of the two sets is empty. The second termination guard tests whether the intersection of the two sets contains at least one element. In this form, the algorithm is equivalent to the Cohen-Sutherland line clipping algorithm [5, p. 66], except for the degree of nondeterminism, and the absence of flags. Implementation of this algorithm, using DO or DOterm, would require stronger iterative guards in order to assure termination, since in this form several iterative guards remain true at completion.

9. CONCLUSIONS

The generalized iterative statement, DOupon, has several advantages over DO. It simplifies some commonly used algorithms by avoiding flags and repeated tests. It simplifies discovery of new algorithms by decoupling the requirements of termination from those of invariance under iteration. It also provides several useful optimizing transformations, which cannot be used with DO and which are safe only because the iteration guards do not affect the termination condition.

DOupon subsumes both DO and IF, in that either of the latter can be expressed as a degenerate form of DOupon. This is analogous to the manner in which IF subsumes **if-then-else** and **case**. Although this unification of the forms is theoretically useful, a practical language would probably still include the simpler forms as syntactic sugar.

Still further unification of language constructs can be obtained by slightly redefining DOupon. In case none of the guards is true, DOupon fails. However, if we specify instead that it simply waits until a guard becomes true, and if we add suitable interprocess communication primitives, the construct becomes similar to those introduced by Hoare for communicating sequential processes [4]. Hoare's distributed termination convention would be replaced by explicit termination conditions. Although such a change would significantly alter the semantics of

DOupon, those given in this paper would still be implied (as a degenerate case) when only one process accesses the relevant variables. In particular, the statement would wait forever (a form of failure) if none of the guards is true. Formal definition of this extension remains a topic for further research.

ACKNOWLEDGMENTS

Comments by Kenneth Baclawski, Richard Bouchard, and Michael Weiss, who reviewed earlier drafts of this paper, contributed substantially to the clarity of presentation. Comments by the referees were also constructive and resulted in substantial improvement to the discussion. The author's current employer, CalComp, provided substantial material support to the revision of this paper.

REFERENCES

1. BOCHMANN, G. V. Multiple exits from a loop without the GOTO. *Commun. ACM 16*, 7 (July 1973), 443–444.
2. DIJKSTRA, E. W. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, N.J., 1976.
3. EVANS, R. V. Multiple exists from a loop using neither GOTO nor labels. *Commun. ACM 17*, 11 (Nov. 1974), 650.
4. Hoare, C. A. R. Communicating sequential processes. *Commun. ACM 21*, 6 (Aug. 1978), 666–677.
5. NEWMAN, W. M., AND SPROULL, R. F. *Principles of Interactive Computer Graphics*. McGraw-Hill, New York, 1979, pp. 65–67.
6. O'DONNELL, M. J. A critique of the foundations of Hoare-style programming logics. *Commun. ACM 25*, 12 (Dec. 1982), 927–935.
7. PARNAS, D. L. A generalized control structure and its formal definition. *Commun. ACM 26*, 8 (Aug. 1983), 572–581.
8. WIRTH, N. Modula: A language for modular programming. *Softw. Pract. Exper. 7* (1977), 3–35.

Received August 1983; revised August 1986; accepted October 1986