Programming Languages
T.A. Standish Editor

Guarded Commands, Nondeterminacy and Formal Derivation of Programs

Edsger W. Dijkstra
Burroughs Corporation

So-called "guarded commands" are introduced as a building block for alternative and repetitive constructs that allow nondeterministic program components for which at least the activity evoked, but possibly even the final state, is not necessarily uniquely determined by the initial state. For the formal derivation of programs expressed in terms of these constructs, a calculus will be be shown.

Key Words and Phrases: programming languages, sequencing primitives, program semantics, programming language semantics, nondeterminacy, case-construction, repetition, termination, correctness proof, derivation of programs, programming methodology

CR Categories: 4.20, 4.22

Author's address: Burroughs, Plataanstraat 5, Nuenen—4565, The Netherlands.

1. Introduction

In Section 2, two statements, an alternative construct and a repetitive construct, are introduced, together with an intuitive (mechanistic) definition of their semantics. The basic building block for both of them is the so-called "guarded command," a statement list prefixed by a boolean expression: only when this boolean expression is initially true, is the statement list eligible for execution. The potential nondeterminacy allows us to map otherwise (trivially) different programs on the same program text, a circumstance that seems largely responsible for the fact that programs can now be derived in a manner more systematic than before.

In Section 3, after a prelude defining the notation, a formal definition of the semantics of the two constructs is given, together with two theorems for each of the constructs (without proof).

In Section 4, it is shown how, based upon the above, a formal calculus for the derivation of programs can be founded. We would like to stress that we do not present "an algorithm" for the derivation of programs: we have used the term "a calculus" for a formal discipline—a set of rules—such that, if applied successfully: (1) it will have derived a correct program; and (2) it will tell us that we have reached such a goal. (We use the term as in "integral calculus.")

2. Two Statements Made from Guarded Commands

If the reader accepts "other statements" as indicating, say, assignment statements and procedure calls, we can give the relevant syntax in BNF [2]. In the following we have extended BNF with the convention that the braces {...} should be read as "followed by zero or more instances of the enclosed."

⟨guarded command⟩ ::= ⟨guard⟩ → ⟨guarded list⟩
⟨guard⟩ ::= ⟨boolean expression⟩
⟨guarded list⟩ ::= ⟨statement⟩ {; ⟨statement⟩}
⟨guarded command set⟩ ::= ⟨guarded command⟩
{[] ⟨guarded command⟩}
⟨alternative construct⟩ ::= **if** ⟨guarded command set⟩ **fi**
⟨repetitive construct⟩ ::= **do** ⟨guarded command set⟩ **od**
⟨statement⟩ ::= ⟨alternative construct⟩ |
⟨repetitive construct⟩ | "other statements"

The semicolons in the guarded list have the usual meaning: when the guarded list is selected for execution its statements will be executed successively in the order from left to right; a guarded list will only be

selected for execution in a state such that its guard is true. Note that a guarded command by itself is *not* a statement: it is a component of a guarded command set from which statements can be constructed. If the guarded command set consists of more than one guarded command, they are mutually separated by the separator □; our text is then an arbitrarily ordered enumeration of an unordered set; i.e. the order in which the guarded commands of a set appear in our text is semantically irrelevant.

Our syntax gives two ways for constructing a statement out of a guarded command set. The alternative construct is written by enclosing it by the special bracket pair **if** . . . **fi**. If in the initial state none of the guards is true, the program will abort; otherwise an arbitrary guarded list with a true guard will be selected for execution.

Note. If the empty guarded command set were allowed **if fi** would be semantically equivalent to "abort". (End of note.)

An example—illustrating the nondeterminacy in a very modest fashion—would be the program that for fixed x and y assigns to m the maximum value of x and y:

if $x \geq y \rightarrow m := x$
□ $y \geq x \rightarrow m := y$
fi.

The repetitive construct is written down by enclosing a guarded command set by the special bracket pair **do** . . . **od.** Here a state in which none of the guards is true will not lead to abortion but to proper termination; the complementary rule, however, is that it will only terminate in a state in which none of the guards is true: when initially or upon completed execution of a selected guarded list one or more guards are true, a new selection for execution of a guarded list with a true guard will take place, and so on. When the repetitive construct has terminated properly, we know that all its guards are false.

Note. If the empty guarded command set were allowed **do od** would be semantically equivalent to "skip". (End of note.)

An example—showing the nondeterminacy in somewhat greater glory—is the program that assigns to the variables $q1$, $q2$, $q3$, and $q4$ a permutation of the values $Q1$, $Q2$, $Q3$, and $Q4$, such that $q1 \leq q2 \leq q3 \leq q4$. Using concurrent assignment statements for the sake of convenience, we can program

$q1, q2, q3, q4 := Q1, Q2, Q3, Q4;$
do $q1 > q2 \rightarrow q1, q2 := q2, q1$
□ $q2 > q3 \rightarrow q2, q3 := q3, q2$
□ $q3 > q4 \rightarrow q3, q4 := q4, q3$
od.

To conclude this section, we give a program where not only the computation but also the final state is not necessarily uniquely determined. The program should determine k such that for fixed value n ($n > 0$) and a fixed function $f(i)$ defined for $0 \leq i < n$, k will eventually satisfy: $0 \leq k < n$ **and** $(\forall i: 0 \leq i < n: f(k) \geq f(i))$. (Eventually k should be the place of a maximum.)

$k := 0; j := 1;$
do $j \neq n \rightarrow$ **if** $f(j) \leq f(k) \rightarrow j := j + 1$
□ $f(j) \geq f(k) \rightarrow k := j; j := j + 1$
fi
od.

Only permissible final states are possible and each permissible final state is possible.

3. Formal Definition of the Semantics

3.1 Notational Prelude

In the following sections we shall use the symbols P, Q, and R to denote (predicates defining) boolean functions defined on all points of the state space; alternatively we shall refer to them as "conditions," satisfied by all states for which the boolean function is true. Two special predicates that we denote by the reserved names T and F play a special role: T denotes the condition that, by definition, is satisfied by all states; F denotes, by definition, the condition that is satisfied by no state at all.

The way in which we use predicates (as a tool for defining sets of initial or final states) for the definition of the semantics of programming language constructs has been directly inspired by Hoare [1], the main difference being that we have tightened things up a bit: while Hoare introduces sufficient pre-conditions such that the mechanisms will not produce the wrong result (but may fail to terminate), we shall introduce necessary and sufficient—i.e. so-called "weakest"—pre-conditions such that the mechanisms are guaranteed to produce the right result.

More specifically: we shall use the notation $wp(S, R)$, where S denotes a statement list and R some condition on the state of the system, to denote the weakest pre-condition for the initial state of the system such that activation of S is guaranteed to lead to a properly terminating activity leaving the system in a final state satisfying the post-condition R. Such a *wp*—which is called "a predicate transformer" because it associates a pre-condition to any post-condition R—has, by definition, the following properties.

1. For any S, we have for all states: $wp(S,F) = F$ (the so-called Law of the Excluded Miracle).
2. For any S and any two post-conditions, such that for all states $P \Rightarrow Q$, we have for all states: $wp(S,P) \Rightarrow wp(S,Q)$.
3. For any S and any two post-conditions P and Q, we have for all states ($wp(S,P)$ **and** $wp(S,Q)$) = $wp(S,P$ **and** $Q)$.
4. For any deterministic S and any post-conditions P

and Q, we have for all states ($wp(S,P)$ **or** $wp(S,Q)$) = $wp(S,P$ **or** $Q)$.

For nondeterministic mechanisms S, the equality has to be replaced by an implication; the resulting formula follows from the second property.

Together with the rules of propositional calculus and the semantic definitions to be given below, the above four properties take over the role of the "rules of inference" as introduced by Hoare [1].

We take the position that we know the semantics of a mechanism S sufficiently well if we know its predicate transformer, i.e. can derive $wp(S,R)$ for any post-condition R.

Note. We consider the semantics of S only defined for those initial states for which has been established a priori that they satisfy $wp(S,T)$, i.e. for which proper termination is guaranteed (even in the face of possibly non-deterministic behavior); for other initial states we don't care. By suitably changing S, if necessary, we can always see to it that $wp(S,T)$ is decidable. (End of note.)

Example 1. The semantics of the empty statement, denoted by "skip" are given by the definition that for any post-condition R, we havewp ("skip", R) = R.

Example 2. The semantics of the assignment statement "$x := E$" are given by wp("$x := E$", R) = R_E^x, in which R_E^x denotes a copy of the predicate defining R in which each occurrence of the variable x is replaced by (E).

Example 3. The semantics of the semicolon ";" as concatenation operator are given by
wp("$S1$; $S2$", R) = $wp(S1, wp(S2,R))$.

3.2 The Alternative Construct

In order to define the semantics of the alternative construct we define two abbreviations.
Let IF denote

if $B_1 \rightarrow SL_1 \,[]\, \ldots \,[]\, B_n \rightarrow SL_n$ **fi**;

let BB denote

$(\exists i : 1 \leq i \leq n : B_i)$;

then, by definition

$wp(IF, R) = (BB$ **and** $(\forall i : 1 \leq i \leq n : B_i \Rightarrow wp(SL_i,R)),$.

(The first term BB requires that the alternative construct as such will not lead to abortion on account of all guards false; the second term requires that each guarded list eligible for execution will lead to an acceptable final state.) From this definition we can derive—by simple substitutions:

THEOREM 1. *From* $(\forall i : 1 \leq i \leq n : (Q$ **and** $B_i) \Rightarrow wp(SL_i,R))$ *for all states we can conclude that* $(Q$ **and** $BB) \Rightarrow wp(IF,R)$ *holds for all states.*

Let t denote some integer function, defined on the state space, and let $wdec(S,t)$ denote the weakest pre-condition such that activation of S is guaranteed to lead to a properly terminating activity leaving the system in a final state such that the value of t is decreased by at least 1 (compared to its initial value). In terms of $wdec$ we can formulate the very similar:

THEOREM 2. *From* $(\forall i : 1 \leq i \leq n : (Q$ **and** $B_i) \Rightarrow wdec(SL_i,t))$ *for all states we can conclude that* $(Q$ **and** $BB) \Rightarrow wdec(IF,t)$ *holds for all states.*

Note (which can be skipped at first reading). The relation between wp and $wdec$ is as follows. For any point X in state space we can regard $wp(S, t \leq t_0)$ as an equation with t_0 as the unknown. Let its smallest solution for t_0 be $tmin(X)$. (Here we have added the explicit dependence on the state X.) Then $tmin(X)$ can be interpreted as the lowest upper bound for the final value of t if the mechanism S is activated with X as initial state. Then, by definition, $wdec(S, t) = (tmin(X) \leq t(X) - 1) = (tmin(X) < t(X))$. (End of note.)

3.3 The Repetitive Construct

As is to be expected, the definition of the repetitive construct

do $B_1 \rightarrow SL_1 \,[]\, \ldots \,[]\, B_n \rightarrow SL_n$ **od**,

that we denote by DO, is more complicated. Let

$H_0(R) = (R$ **and non** $BB)$

and for $k > 0$,

$H_k(R) = (wp(IF, H_{k-1}(R))$ **or** $H_0(R))$

(where IF denotes the *same* guarded command set enclosed by "**if fi**"). Then, by definition

$$wp(DO, R) = (\exists k : k \geq 0 : H_k(R)).$$

(Intuitively, $H_k(R)$ can be interpreted as the weakest pre-condition guaranteeing proper termination after at most k selections of a guarded list, leaving the system in a final state satisfying R.) Via mathematical induction we can prove:

THEOREM 3. *If we have for all states* $(P$ **and** $BB) \Rightarrow (wp(IF, P)$ **and** $wdec(IF, t)$ **and** $t \geq 0)$ *we can conclude that we have for all states* $P \Rightarrow wp(DO, P$ **and non** $BB)$.

Note. The antecedent of Theorem 3 is of the form of the consequents of Theorems 1 and 2. (End of note.)

Because T is the condition by definition satisfied by all states, $wp(S,T)$ is the weakest pre-condition guaranteeing proper termination for S. This allows us to formulate an alternative theorem about the repetitive construct, viz.:

THEOREM 4. *From* $(P$ **and** $BB) \Rightarrow wp(IF, P)$ *for all states, we can conclude that we have for all states* $(P$ **and** $wp(DO, T)) \Rightarrow wp(DO, P$ **and non** $BB)$.

Note. In connection with the above theorems, P is called "the invariant relation" and t is called "the variant function." Theorems 3 and 4 are easily proved by mathematical induction, with k as the induction variable. (End of note.)

4. Formal Derivation of Programs

The formal requirement of our program performing $m := max(x,y)$—see above—is that for fixed x and y it establishes the relation

R: ($m = x$ **or** $m = y$) **and** $m \geq x$ **and** $m \geq y$.

Now the Axiom of Assignment tells us that "$m := x$" is the standard way of establishing the truth of $m = x$ for fixed x, which is a way of establishing the truth of the first term of R. Will "$m := x$" do the job? In order to investigate this, we derive and simplify:

$$wp(\text{"}m := x\text{"}, R) = (x = x \textbf{ or } x = y) \textbf{ and } x \geq x \textbf{ and } x \geq y$$
$$= x \geq y.$$

Taking this weakest pre-condition as its guard, Theorem 1 tells us that

if $x \geq y \rightarrow m := x$ **fi**

will produce the correct result if it terminates successfully. The disadvantage of this program is that $BB \neq T$; i.e. it might lead to abortion; weakening BB means looking for alternatives which might introduce new guards. The obvious alternative is the assignment "$m := y$" with the guard $wp(\text{"}m := y\text{"}, R) = y \geq x$; thus we are led to our program

if $x \geq y \rightarrow m := x$
$[]\ y \geq x \rightarrow m := y$
fi

and by this time $BB = T$, and therefore we have solved the problem. (In the meantime we have proved that the maximum of two values is always defined, viz. that R considered as equation for m has always a solution.)

As an example of the derivation of a repetitive construct we shall derive a program for the greatest common divisor of two positive numbers; i.e. for fixed, positive X and Y we have to establish the final relation $x = gcd(X,Y)$.

The formal machinery only gets in motion, once we have chosen our invariant relation and our variant function. The program then gets the structure

"establish the relation P to be kept invariant";
do "decrease t as long as possible under variance of P" **od.**

Suppose that we choose for the invariant relation

P: $gcd(X,Y) = gcd(x,y)$ **and** $x > 0$ **and** $y > 0$,

a relation that has the advantage of being easily established by $x := X; y := Y$.

The most general "something" to be done under invariance of P is of the form $x, y := E1, E2$, and we are interested in a guard B such that

$$(P \textbf{ and } B) \Rightarrow wp(\text{"}x, y := E1, E2\text{"}, P)$$
$$= (gcd(X, Y) = gcd(E1, E2) \textbf{ and } E1 > 0 \textbf{ and } E2 > 0).$$

Because the guard must be a computable boolean expression and should not contain the computation of $gcd(X, Y)$—for that was the whole problem—we must see to it that the expressions E1 and E2 are so chosen, that the first term $gcd(X, Y) = gcd(E1, E2)$ is implied by P, which is true if $gcd(x, y) = gcd(E1, E2)$. In other words we are invited to massage the value pair (x,y) in such a fashion that their gcd is not changed. Because—and this is the place at which to mobilize our mathematical knowledge about the gcd-function—$gcd(x, y) = gcd(x - y, y)$, a possible guarded list would be $x := x - y$. Deriving $wp(\text{"}x := x - y\text{"}, P) = (gcd(X, Y) = gcd(x - y, y)$ **and** $x - y > 0$ **and** $y > 0)$ and omitting all terms of the conjunction implied by P, we find the guard $x > y$ as far as the invariance of P is concerned. Besides that we must require guaranteed decrease of the variant function t. Let us investigate the consequences of the choice $t = x + y$. From

$$wp(\text{"}x := x - y\text{"}, t \leq t_0) = wp(\text{"}x := x - y\text{"}, x + y \leq t_0) = (x \leq t_0),$$

we conclude that $tmin = x$; therefore $wdec(\text{"}x := x - y\text{"}, t) = (x < x + y) = (y > 0)$.

The requirement of monotonic decrease of t imposes no further restriction of the guard because $wdec(\text{"}x := x - y\text{"}, t)$ is fully implied by P, and at our first effort we come to

$x := X; y := Y;$
do $x > y \rightarrow x := x - y$ **od.**

Alas, this single guard is insufficient: from P **and non** BB we are not allowed to conclude $x = gcd(X,Y)$. In a completely analogous manner, the alternative $y := y - x$ will require as its guard $y > x$, and our next effort is

$x := X; y := Y;$
do $x > y \rightarrow x := x - y$
$[]\ y > x \rightarrow y := y - x$
od.

Now the job is done, because with this last program **non** $BB = (x = y)$ and $(P$ **and** $x = y) \Rightarrow (x = gcd(X,Y)$, because $gcd(x,x) = x$.

Note. The choice of $t = x + 2y$ and the knowledge of the fact that the gcd is a symmetric function could have led to the program

$x := X; y := Y;$
do $x > y \rightarrow x := x - y$
$[]\ y > x \rightarrow x, y := y,x$
od.

The swap $x,y := y,x$ can never destroy P: the guard of the last guarded list is fully caused by the requirement that t is effectively decreased. (End of note.)

In both cases the final game has been to find a large enough set of such guarded lists that BB, the disjunction of their guards, was sufficiently weak: in the case

of the alternative construct the purpose is avoiding abortion, in the case of the repetitive construct the goal is getting *BB* weak enough such that *P* **and non** *BB* is strong enough to imply the desired post-condition *R*.

It is illuminating to compare our first version of Euclid's Algorithm with what we would have written down with the traditional clauses:

$x := X; y := Y;$ (version A)
while $x \neq y$ **do if** $x > y$ **then** $x := x - y$
else $y := y - x$ **fi od**

and

$x := X; y := Y;$ (version B)
while $x \neq y$ **do while** $x > y$ **do** $x := x - y$ **od**;
while $y > x$ **do** $y := y - x$ **od**
od.

In the fully symmetric version with the guarded commands the algorithm has been reduced to its bare essentials, while the traditional clauses force us to choose between versions A and B (and others), a choice that can only be justified by making assumptions about the time taken for tests and about expectation values for traversal frequencies. (But even taking the time taken for tests into account, it is not clear that we have lost: the average number of necessary tests per assignment ranges with guarded commands from 1 to 2, equals 2 for version A and ranges from 1 to 2.5 for version B. If the guards of a guarded command set are evaluated concurrently—nothing in our semantics excludes that—the new version is time-wise superior to all the others.) The virtues of the *case*-construction have been extended to repetition as well.

5. Concluding Remarks

The research, the outcome of which is reported in this article, was triggered by the observation that Euclid's Algorithm could also be regarded as synchronizing the two cyclic processes "**do** $x := x - y$ **od**" and "**do** $y := y - x$ **od**" in such a way that the relation $x > 0$ **and** $y > 0$ would be kept invariantly true. It was only after this observation that we saw that the formal techniques we had already developed for the derivation of the synchronizing conditions that ensure the harmonious cooperation of (cyclic) sequential processes, such as can be identified in the total activity of operating systems, could be transferred lock, stock, and barrel to the development of sequential programs as shown in this article. The main difference is that while for sequential programs the situation "all guards false" is a desirable goal—for it means termination of a repetitive construct—one tries to avoid it in operating systems—for there it means deadlock.

The second reason to pursue these investigations was my personal desire to get a better appreciation, which part of the programming activity can be regarded as a formal routine and which part of it seems to require "invention." While the design of an alternative construct now seems to be a reasonably straightforward activity, that of a repetitive construct requires what I regard as "the invention" of an invariant relation and a variant function. My presentation of this calculus should, however, not be interpreted as my suggestion that all programs should be developed in this way: it just gives us another handle.

The calculus does, however, explain my preference for the axiomatic definition of programming language semantics via predicate transformers above other definition techniques: the definition via predicate transformers seems to lend itself most readily to being forged into a tool for the goal-directed activity of program composition.

Finally, I would like to add a word or two about the potential nondeterminacy. Having worked mainly with hardly self-checking hardware, with which nonreproducing behavior of user programs is a very strong indication of a machine malfunctioning, I had to overcome a considerable mental resistance before I found myself willing to consider nondeterministic programs seriously. It is, however, fair to say that I could never have discovered the calculus before having taken that hurdle: the simplicity and elegance of the above would have been destroyed by requiring the derivation of deterministic programs only. Whether nondeterminacy is eventually removed mechanically—in order not to mislead the maintenance engineer—or (perhaps only partly) by the programmer himself because, at second thought, he does care—e.g. for reasons of efficiency—which alternative is chosen is something I leave entirely to the circumstances. In any case we can appreciate the nondeterministic program as a helpful stepping stone.

Acknowledgments. In the first place my acknowledgments are due to the members of the IFIP. Working Group W.G.2.3 on "Programming Methodology." Besides them, W.H.J. Feijen, D.E. Knuth, M. Rem, and C.S. Scholten have been directly helpful in one way or another. I should also thank the various audiences—in Albuquerque (courtesy NSF), in San Diego and Luxembourg (courtesy Burroughs Corporation)— that have played their role of critical sounding board beyond what one is entitled to hope.

Received July 1974; revised January 1975

References

1. Hoare, C.A.R. An axiomatic basis for computer programming. *Comm. ACM 12*, 10 (Oct. 1969), 576–583.
2. Naur, Peter (Ed.). Report on the algorithmic language ALGOL 60. *Comm. ACM 3*, (May 1960), 299–314.