Guarded commands, non-determinacy and a calculus for the derivation of programs.

by Edsger W.Dijkstra *)

*) Author's address: BURROUGHS

Plataanstraat 5

NUENEN - 4565

The Netherlands.

Abstract. So-called "guarded commands" are introduced as a building block for alternative and repetitive constructs that allow non-deterministic 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 shown.

Guarded commands, non-determinacy and a calculus for the derivation of programs.

1. Introduction.

In section 2, two statements, an alternative construct and a repetitive construct will be 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 non-determinacy allows us to map otherwise (trivially) different programs on the same program text, a circumstance that seems largely responsible for the fact that

now programs can be derived in a more systematic manner than before.

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

In section 4, it will be shown, how 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

2)   it will tell us that we have reached such a goal.

(In choosing the term "calculus" we have been inspired by the "integral calculus" and the "propositional calculus" where we have a very similar situation.)

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".

$$< \text{guarded command} > ::= < \text{guard} > \rightarrow < \text{guarded list} >$$
$$< \text{guard} > ::= < \text{boolean expression} >$$
$$< \text{guarded list} > ::= < \text{statement} > \{ ; < \text{statement} > \}$$
$$< \text{guarded command set} > ::= < \text{guarded command} > \{ [] < \text{guarded command} > \}$$
$$< \text{alternative construct} > ::= \underline{if} < \text{guarded command set} > \underline{fi}$$
$$< \text{repetitive construct} > ::= \underline{do} < \text{guarded command set} > \underline{od}$$
$$< \text{statement} > ::= < \text{alternative construct} > \mid < \text{repetitive construct} > \mid$$
$$\text{"other statements"} \quad .$$

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 <u>not</u> a statement: it is 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: "<u>if</u> ... <u>fi</u>". 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.

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

An example --illustrating the non-determinacy 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 :

$$\underline{if}\ x \geq y \rightarrow m:= x$$
$$[]\ y \geq x \rightarrow m:= y$$
$$\underline{fi}\qquad .$$

The repetitive construct is written down by enclosing a guarded
command set by the special bracket pair "<u>do</u> ... <u>od</u>" . 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.

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

An example --showing the non-determinacy 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

$$q1, q2, q3, q4 := Q1, Q2, Q3, Q4;$$
$$\underline{do}\ q1 > q2 \rightarrow q1, q2 := q2, q1$$
$$[]\ q2 > q3 \rightarrow q2, q3 := q3, q2$$
$$[]\ q3 > q4 \rightarrow q3, q4 := q4, q3$$
$$\underline{od} \qquad .$$

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:

$$0 \leq k < n \ \underline{and}\ (\forall i:\ 0 \leq i < n:\ f(k) \geq f(i)) \qquad .$$

(Eventually k should be the place of a maximum.)

$$k := 0;\ j := 1;$$
$$\underline{do}\ j \neq n \rightarrow \underline{if}\ f(j) \leq f(k) \rightarrow j := j + 1$$
$$[]\ f(j) \geq f(k) \rightarrow k := j;\ j := j + 1$$
$$\underline{fi}$$
$$\underline{od} \quad .$$

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

$$wp(S, P) \Rightarrow wp(S, Q) \quad .$$

3)   For any S and any two post-conditions P and Q we have for all states

$$(wp(S, P) \text{ and } wp(S, Q)) = wp(S, P \text{ and } Q) \quad .$$

4)   For any S and any two post-conditions P and Q we have for all states

$$(wp(S, P) \text{ or } wp(S, Q)) \Rightarrow wp(S, P \text{ or } Q) \quad .$$

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. This position is taken in full acknowledgement of the fact that in
the case of non-deterministic mechanisms, the knowledge of the predicate
transformer does not give a complete description: for those initial states
that do not necessarily lead to a properly terminating activity, the
knowledge of the predicate transformer does not give us any information
about the final states in which the system might find itself after proper
termination. (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 have

$$wp("skip", R) = R \quad .$$

Example 2. The semantics of the assignment statement "x:= E" are
given by
$$wp("x:= E", R) = R_E^x \quad ,$$

in which R_E^x denotes a copy of the predicate defining R in which each
occurence of the variable "x" is replaced by "(E)".

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

3.2. The alternative construct.

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

Let "IF" denote
$$\underline{if}\ B_1 \rightarrow SL_1\ [\!]\ \dots\ [\!]\ B_n \rightarrow SL_n\ \underline{fi} \quad ;$$

let "BB" denote
$$(\exists i:\ 1 \leq i \leq n:\ B_i) \quad ;$$

then, by definition

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

(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

$$(\forall i: 1 \leq i \leq n: (Q \text{ and } B_i) \Rightarrow wp(SL_i, R)) \text{ for all states}$$

we can conclude that

$$(Q \text{ and } BB) \Rightarrow wp(IF, R) \quad \text{holds for all states} .$$

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

$$(\forall i: 1 \leq i \leq n: (Q \text{ and } B_i) \Rightarrow wdec(SL_i, t)) \text{ for all states}$$

we can conlude that

$$(Q \text{ and } BB) \Rightarrow wdec(IF, t) \quad \text{holds for all states.}$$

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,

$$wdec(S, t) = (tmin(X) \leq t(X) - 1) = (tmin(X) < t(X)) \quad .$$

(End of note.)

3.3. The repetitive construct.

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

$$\underline{do}\ B_1 \rightarrow SL_1\ [\!]\ \ldots\ [\!]\ B_n \rightarrow SL_n\ \underline{od}$$

that we denote by "DO" , is more complicated.

Let
$$H_0(R) = (R\ \underline{and}\ \underline{non}\ BB)$$
and for $k > 0$:
$$H_k(R) = (wp(IF,\ H_{k-1}(R))\ \underline{or}\ H_0(R))$$

(where "IF" denotes the <u>same</u> guarded command set enclosed by "<u>if</u>   <u>fi</u>") then, by definition

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

(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

<u>Theorem 3</u>. From

$$(P\ \underline{and}\ BB) \Rightarrow (wp(IF,\ P)\ \underline{and}\ wdec(IF,\ t))\quad \text{for all states}$$
and
$$(BB\ \underline{and}\ P) \Rightarrow (t \geq 0)\quad \text{for all states}$$
we can conclude that we have for all states

$$P \Rightarrow wp(DO,\ P\ \underline{and}\ \underline{non}\ BB)\qquad .$$

Note that the antecedent of Theorem 3 is of the form of the consequents of Theorems 1 and 2.

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.

<u>Theorem 4</u>. From

$$(P\ \underline{and}\ BB) \Rightarrow wp(IF,\ P)\quad \text{for all states,}$$
we can conclude that we have for all states

$$(P\ \underline{and}\ wp(DO,\ T)) \Rightarrow wp(DO,\ P\ \underline{and}\ \underline{non}\ BB)\qquad .$$

In connection with the above theorems "P" is called "the invariant relation"

and "t" is called "the variant function".

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: $(m = x$ <u>or</u> $m = y)$ <u>and</u> $m \geq x$ <u>and</u> $m \geq y$.

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

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

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

<u>if</u> $x \geq y \rightarrow$ m:= x <u>fi</u>

will produce the correct result if it terminates succesfully. The disad-
vantage 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

$$wp(\text{"m:= y"}, R) = y \geq x   ;$$

thus we are led to our program

<u>if</u> $x \geq y \rightarrow$ m:= x
⫾ $y \geq x \rightarrow$ m:= y
<u>fi</u>

and by this time BB = T and therefore we have solved the problem. (In
the mean time 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 deriviation 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

$$x = gcd(X, Y) \qquad .$$

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";

<u>do</u> "decrease t as long as possible under invariance

of P" <u>od</u> .

Suppose that we choose for the invariant relation

P: $\quad gcd(X, Y) = gcd(x, y)$ <u>and</u> $x > 0$ <u>and</u> $y > 0$

a relation that has the advantage of being easily established by

$$x := X; \quad y := Y \qquad .$$

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

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

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

$$gdc(x, y) = gcd(E1, E2) \qquad .$$

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 where to mobilize our mathematical knowledge about the gcd-function--

$$gcd(x, y) = gcd(x - y, y)$$

a possible guarded list would be

$$x := x - y \qquad .$$

Deriving

$$\text{wp}("x := x - y", R) = \left(\text{gcd}(X, Y) = \text{gcd}(x - y, y) \ \underline{\text{and}} \ x - y > 0 \ \underline{\text{and}} \ y > 0\right)$$

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

$$t = x + y \qquad .$$

From

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

we conclude that
$$\text{tmin} = x \qquad ;$$

therefore
$$\text{wdec}("x := x - y", \ t) = \left(x < x + y\right) = \left(y > 0\right) \qquad .$$

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

$$x := X; \ y := Y;$$
$$\underline{\text{do}} \ x > y \rightarrow x := x - y \ \underline{\text{od}} \qquad .$$

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

$$x := X; \ y := Y;$$
$$\underline{\text{do}} \ x > y \rightarrow x := x - y$$
$$\ \| \ y > x \rightarrow y := y - x$$
$$\underline{\text{od}} \qquad .$$

Now the job is done, because with this last program $\underline{\text{non}}$ BB $= \left(x = y\right)$ and $\left(P \ \underline{\text{and}} \ x = y\right) \Rightarrow \left(x = \text{gcd}(X, Y) \right.$ because $\text{gdc}(x, x) = x$.

$\underline{\text{Note.}}$ 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

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

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.

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 suffi-
ciently 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 .

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 "$\underline{do}$ x := x - y $\underline{od}$" and "$\underline{do}$ y := y - x $\underline{od}$'
in such a way that the relation x > 0 $\underline{and}$ 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 co-operation 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 --among other things in order to be
able to evaluate how realistic some claims towards "automatic programming"
were-- which part of the programming activity can be regarded as formal
routine and which part of it seems to require "invention". While the
design of an alternative construct now seems to be a reasonably straight-

forward activity, that of a repetitive construct requires what I regard as "the invention" of an invariant relation and a variant function. For me, the main value of the calculus shown in section 4 is that it has strenght- ened my skepticism about some of the claims or goals of "automatic pro- gramming"; me presenting this calculus should not be interpreted as me suggesting that all programs should be developed that 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 say a word or two about the role of the po- tential non-determinacy. I quote in this connection C.A.R.Hoare: "A system which permits user programs to become non-deterministic presents dreadful problems to the maintenance engineer: it is not a "facility" to be lightly granted." (This is particularly true in the absence of self-checking hard- ware.) I myself had to overcome a considerable mental resistance before I found myself willing to consider non-deterministic programs seriously. It is, however, fair to say that I could not have discovered the calculus shown before having taken that hurdle and I leave it to the environment whether the non-determinacy is eventually resolved by human intervention or mechanically, in a reproducible manner or not. (It is only in an envi- ronment in which all programs should be deterministic, where non-reproducible behaviour is interpreted as machine malfunctioning: I can easily think of an environment in which non-reproducible user program behaviour is quite naturally and almost always correctly taken as an indication that the user in question has written a non-deterministic program!)

Acknowledgements.

In the first place my acknowledgements are due to the members of the IFIP Working Group W.G.2.3 on "Programming Methodology": it was, as a matter of fact, during the Blanchland meeting of this Working Group in October 1973 that the guarded commands were both born and shown to the

public. In connection with this effort its members R.M.Burstall, C.A.R. Hoare, J.J.Horning, J.C.Reynolds, D.T.Ross, G.Seegmüller, N.Wirth and M. Woodger deserve my special thanks. 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.

[1]   Hoare, C.A.R., An Axiomatic Basis for Computer Programming, Comm.ACM 12 (Oct. 1969) 576 - 583.

[2]   Naur, Peter, (Ed.) Report on the Algorithmic Language ALGOL 60, Comm.ACM 3 (May 1960) 299 - 314

26th June 1974   prof.dr.Edsger W.Dijkstra
NUENEN   Burroughs Research Fellow