Theoretical Computer Science 13 (1981) 175–192
© North-Holland Publishing Company

ON THE TOTAL CORRECTNESS OF NONDETERMINISTIC PROGRAMS*

David HAREL**
IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, U.S.A.

Communicated by A. Meyer
Received June 1978
Revised April 1979

Abstract. For an arbitrary programming language with nondeterminism to be implementable, the existence of *computation trees* modelling the possible changes in state in the course of a computation is postulated. A general definition of what constitutes an *execution method* is then presented. Falling naturally out of these ideas is the correspondence between execution methods and total correctness, in that different properties are required of a program to be correct when different methods are adopted. We describe a variety of plausible methods of execution falling under the general definition and single out four particular ones. The arguments made are then illustrated by analysing the properties required by Dijkstra of guarded commands in view of these four methods. We conclude that a general approach such as that suggested here seems to be needed for dealing with programming languages and execution methods other than the particular ones treated by Dijkstra.

1. Introduction

The main objective of this paper is the presentation of a convincing argument to the extent that, contrary to the case with deterministic programs, the concept of a nondeterministic program being totally correct depends heavily upon the way in which such a program is executed.

In general, the possibilities associated with the behavior of a nondeterministic program in some state can be collected in a *computation tree*, the nodes of which are labelled with states. The root is labelled with the start state and the leaves, if any, either with final states, i.e., states in which the program can terminate, or with some indication of failure. It is, of course, possible that the tree will also contain infinite paths. A branching point in the tree indicates that the program permits a nondeterministic choice between the computations associated with

*Part of this work was done while the author was with the Laboratory for Computer Science at the Massachusetts Institute of Technology, and was supported by NSF grants MCS76-18461 and MCS77-19754.
**Current address: Department of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel.

each branch. We postulate the existence of such a tree for every program and every possible start state, in any implementable nondeterministic programming language.

It is not at all clear, however, how a single computation is extracted from such a tree or, for that matter, what a computation is. One can envision a mechanism following a path from the root down the tree, keeping track of the changes in states, and choosing (say at random) how to branch at each branching point. One can also envision a mechanism with access to more than one processor, following some of the alternatives of a branch point in parallel.

In Section 2 we present a general definition of an execution method (avoiding questions of practicality, efficiency, etc.) as assigning to each computation tree a set of possible *traversals*. A traversal can be thought of as a path in the tree traced by a possibly multiple-processor mechanism. An *execution* of the program using such a method will consist of 'carrying out' one of the traversals. Thus, an execution can succeed, fail, or diverge depending, respectively, upon whether the traversal selected is of finite depth and has final states, is of finite depth but has no final states, or is of infinite depth.

Falling naturally out of this setup is the correspondence between execution methods and definitions of total correctness, i.e., the property of a program being guaranteed to terminate with the expected results. For example, if no traversal of infinite depth is associated with a tree there is no need to require that the tree be free of infinite paths. Accordingly, we define the notion of total correctness, as being generic, i.e., parameterized by execution methods.

The rest of the paper is devoted to the illustration of this general approach with the aid of 'real' execution methods and a 'real' programming language. In Section 3 a variety of execution methods are described and four particular ones are singled out, corresponding to four different definitions of total correctness. In Section 4 the guarded commands programming language of Dijkstra [6] is introduced and the computation trees of such programs defined. An analysis of the properties required by Dijkstra of the weakest precondition (guaranteeing total correctness) is then carried out with our four methods of execution in mind. It is shown in Section 5 that these properties correspond to the most restrictive of the four methods – depth first search with no backtracking. In other words, a guarded commands program which is totally correct a la Dijkstra will be totally correct with respect to any one of the four methods but, in general, not vice versa; a significant relaxation of the requirements for a program to be totally correct can be achieved by adopting any one of the other three. Some of the results of Section 5 have been obtained also by Hoare [10] using his logic of traces, and related work can be found in [1, 15, 16, 17]. The upshot of Sections 4 and 5 is that, although the definitions in [5, 6] give rise to a unique well-defined notion of total correctness for this particular programming language, dealing with other languages or with other execution methods seems to require a general framework such as that proposed in Sections 2 and 3.

2. Computation trees, execution methods and total correctness

Let us assume we are given a set S of *states*. We would like to think of a nondeterministic program as being some well-formed expression in some programming language, with which, whenever a start state $s \in S$ is selected, a pattern of possible computation can be associated. This pattern should consist of possible changes the state s can go through, triggered by parts of the program. Were the program deterministic, this computation would take the form of a finite or infinite *sequence* $s = s_0, s_1, s_2, \ldots$. Being nondeterministic, however, we want to view the computation as a *tree* labelled with states, the root being labelled with s. Branch points of the tree should correspond to the nondeterministic choices in such a way that the sequence of states along any path from the root down the tree corresponds to a possible sequence of changes during a computation, i.e., a *computation sequence.* If such a path is finite and terminates in a leaf labelled, say, s′, we should be able to conclude that it is possible for the program to terminate in state s′, and if it is infinite that it is possible for the program to diverge, i.e., enter an 'infinite loop'. We must also provide means for indicating when an abnormal termination occurs, that is, a finite computation sequence ending prematurely (e.g., division by zero, false 'guards', deadlock, etc.). For the purposes of this paper there is no need to distinguish between these kinds of abnormalities; we can term any such computation sequence a *failure,* and use a special symbol, say F, to label leaves corresponding to the last state in such sequences.

The understanding of the rest of this paper is dependent upon the reader accepting our classification of the computations of nondeterministic programs; infinite ones, finite 'bad' ones and finite 'good' ones (in the sequel, respectively, divergences, failures and terminations). This classification has been made, either explicitly or implicitly, in [2, 7-10, 12, 13]. Accordingly, from now on we will assume that the semantics of the nondeterministic programming languages we discuss are given in such a way that a computation tree as described above, and as defined more formally below, exists for any program in any given start state.

Let Σ be some set, and let N be the set of natural numbers. We use u and v to range over N*, i.e., over finite sequences of natural numbers, with λ standing for the empty sequence. A *tree over Σ* is a partial function $T: N^* \to \Sigma$, whose domain, dom(T), is non-empty and satisfies: if $u \in \mathrm{dom}(T)$ and $v < u$, then $v \in \mathrm{dom}(T)$, where $<$ is the natural lexicographical ordering on N* (e.g. $0101 < 01010 < 01011$). If $u \in \mathrm{dom}(T)$ we say that u is a *node* of T and is *labelled* by T(u). λ is the *root* of T. We shall use standard terminology for trees, such as *path, descendent,* etc. For $u \in \mathrm{dom}(T)$, let $A = \{i \mid ui \in \mathrm{dom}(T)\}$. Then u is said to be *of degree* $k+1$, *of infinite degree,* or *a leaf,* respectively, if $A = \{0, 1, 2, \ldots, k\}$, $A = N$ or $A = \phi$.

Let S be a fixed set of states, and let F be a symbol not in S. Let α be a program in a given programming language, and let $s \in S$. The *computation tree of α in* s, denoted $ct(\alpha, s)$, is a tree over $S \cup \{F\}$ whose root is labelled by s and in which F labels leaves only. In other words, $ct(\alpha, s)(\lambda) = s$, and if $ct(\alpha, s)(u) = F$, then $u0 \notin \mathrm{dom}(ct(\alpha, s))$.

Some further notational conveniences for computation trees are the following.

$$\text{final}_\alpha(s) = \{t \mid t \in S \text{ and } t \text{ labels a leaf of } ct(\alpha, s)\};$$

$$\text{fail}_\alpha(s) = \begin{cases} 1, & \text{there is a leaf in } ct(\alpha, s) \text{ labelled } \boldsymbol{F}, \\ 0, & \text{otherwise;} \end{cases}$$

$$\text{loop}_\alpha(s) = \begin{cases} 1, & \text{there is an infinite path in } ct(\alpha, s), \\ 0, & \text{otherwise.} \end{cases}$$

The concepts of fail_α and loop_α correspond, respectively, to the derivatives and coderivatives of [9], Block(α) and Loop(α) of [2], the blind alleys and $\boldsymbol{b}(\alpha)$ of [10], and $fl(\alpha)$ and $dv(\alpha)$, the latter also $[\alpha]^+$, of [3]. The present notation is similar to that used in [7]. A program α is said to be of *finite nondeterminism in* s if each node of ct(α, s) is of finite degree; it is said to be of *bounded nondeterminism in* s if there is a constant k such that each node of ct(α, s) is of degree at most k; it is said to be of *finite nondeterminism* if it is of finite nondeterminism for every s $\in$ S; it is said to be of *bounded nondeterminism* if there is a k such that for every s $\in$ S each node of ct(α, s) is of degree at most k. Note that even the strongest of these properties, namely that α is of bounded nondeterminism, does not preclude the possibility of dom(ct(α, s)) being infinite, or indeed of final_α (s) being infinite; it might be possible (see Section 3) for a program to be able to go through an infinite sequence of states with a (2-way) choice at each state either terminating or continuing to compute. We are aware of the opinion (see e.g., [6, 10]) that, practically speaking, one can restrict oneself to programming languages for which every program is of bounded nondeterminism, and indeed Section 4 of our paper is concerned with one such language. This point, however, is controversial (see [3]), so we see no reason to exclude other possibilities from the general discussion. Some ideas on unbounded nondeterminism appear in [4].

We now want to capture, in a general way, the intuitive notion of *executing* a nondeterministic program α in state s. We would like to think of such an execution as being induced by some mechanism M which is 'started' in state s (a fact indicated, say, by the contents of part of its memory) and which goes through some process of state changes until (hopefully) reaching a state s' $\in$ final_α (s). Clearly, one way of doing this is to have M trace a path down ct(α, s) (say the 0-path, λ, 0, 00, 000, . . .) starting from the root. We would like, however, to provide for more than these simple single-path downward traces. For example, we would like to include the cases where M can traverse parts of the tree backwards (e.g., by backtracking when it sees something it does not like, such as an $\boldsymbol{F}$-leaf), or pursue more than one branch simultaneously.

Motivated by the desire to include reasonable traversal methods rather than by the need to exclude unreasonable ones, we have arrived at the definitions presented below. It is clear though, that additional work has to be done if such a general definition is to reflect in some sense the 'practical' methods. For the purposes of this paper, however, the definitions suffice. In Section 3 we describe some particular methods in a manner which is more algorithmic and which points at least to their computability; they are all included in the following definition as special cases, as are the various constructs appearing in [4].

Let T be a tree over Σ. A *traversal* H of T is a tree over dom(T) such that

(1) $H(\lambda)=\lambda$, and

(2) If $H(u)=v$ and $H(ui)=w$, then for some j, either $v=wj$ or $w=vj$.

The labels of H are to be viewed as the nodes of T. H can be thought of as a generalized path through T starting from its root; travelling down H along some path corresponds to a path in the undirected version of T.

Denote by TN the set of trees over N^* and by tr(T) the set of traversals of T. Clearly, $\mathrm{tr}(T)\subset TN$. Let L be a programming language. An *execution method* E for L is a function.

$$E:L\times S\to 2^{TN}$$

such that for every α and s, $E(\alpha,s)\subset \mathrm{tr}(\mathrm{ct}(\alpha,s))$. In other words, an execution method assigns to each program and starting state a set of traversals of the appropriate computation tree. As noted above, we ignore here the question of feasibility of execution methods.

The intuition behind the definition will hopefully become clearer in the next section. For now it suffices to note that a mechanism M using method E will traverse $\mathrm{ct}(\alpha,s)$ in a manner captured by some traversal $H\in E(\alpha,s)$, the paths of H being pursued by M simultaneously.

For $H\in E(\alpha,s)$ and u a node of H, denote by lab(u) the label of the node of $\mathrm{ct}(\alpha,s)$ labelling u; i.e., $\mathrm{lab}(u)=\mathrm{ct}(\alpha,s)(H(u))$. Now define

$$\mathrm{fin}_H(s)=\{\mathrm{lab}(u)\,|\,\mathrm{lab}(u)\in \mathrm{final}_\alpha(s) \text{ and u is a leaf of H}\}.$$

$\mathrm{fin}_H(s)$ is the set of states which are final states of α when started in s, and which are 'found' by traversal H (in the sense that they correspond to the leaves of H).

The total correctness of a program α with respect to input and output conditions P and Q is, intuitively, the property that, starting in any state satisfying P, α is guaranteed to terminate properly in a state satisfying Q. In our context P and W will be taken to be subsets of the fixed sets of states S ($s\in S$ 'satisfies' P if $s\in P$), and the vague notion of being 'guaranteed to terminate properly' will be relativized to an execution method.

Let E be an execution method for L; let $\alpha\in L$, $P\subset S$ and $Q\subset S$. We say that α is E-*totally correct with respect to* P *and* Q if for every $s\in P$ and for every $H\in E(\alpha,s)$,

(1) there is a bound on the length of the paths of H, and

(2) $\mathrm{fin}_H(s)\cap Q\neq\phi$.

The intuition is that for a program α to always succeed in achieving Q when started in a state $s\in P$, any traversal H selected must be 'traversable' in finite time, and must lead to the finding of a final state satisfying Q. Our definition thus isolates the nondeterminism in α, by reducing it to the single choice of a traversal. Of course, in practice the traversal actually chosen might be constructed step by step by making appropriate choices during the computation. We shall see in the next section that many plausible execution methods assign only special, simple kinds of traversals to programs, with the result that the property of α being totally correct by that method can be described quite succinctly in terms of the computation trees of α and the conditions P and Q.

The alert reader will no doubt have some objection to our choice of part (2) in the above definition, over the more natural

(2′) $\mathrm{fin}_H(s) \neq \phi \wedge \mathrm{fin}_H(s) \subset Q$.

First, let us say here that the rest of the paper is invariant under this change; in particular, the execution methods we analyze in Section 3 and utilize in 5, have the property that the total correctness of α with respect to P and Q according to these methods is the same either way. It is precisely this observation which convinced us to give the definition which, in general, is the weaker of the two. The definition as stated will allow us to regard as totally correct with respect to P and Q programs which have final states in which Q is false, on condition that they also have final states in which Q is true. This will happen, though, only for very special (and strange) execution methods in which some property Q is 'built-in', enabling the execution mechanism itself to check for Q, rather than that check be part of the program. Our point is that making (2) weak does not prevent us from achieving strong notions of total correctness; these can always be obtained by limiting the set of traversals as seen in the next section. However, the reader might feel more comfortable with (2′) and is free to use it instead.

We close this section with a definition of the *weakest-precondition* of a program with respect to an assertion Q. Again, let E be an execution method for L, $\alpha \in L$ and $Q \subset S$. Define

$$\mathrm{wp}_E(\alpha, Q) = \{s \mid s \in S \text{ and } \alpha \text{ is E-totally correct with respect to } \{s\} \text{ and } Q\}.$$

We note that α is E-totally correct with respect to P and Q iff $P \subset \mathrm{wp}_E(\alpha, Q)$. Consequently, in the remainder of this paper we will investigate the notion of total correctness via the related notion of weakest precondition. Specifically, the following equivalent definition of wp_E will be used:

$$s \in \mathrm{wp}_E(\alpha, Q) \text{ iff}$$

$$(\forall H \in E(\alpha, s))(\mathrm{fin}_H(s) \cap Q \neq \phi \text{ and the depth of } H \text{ is bounded}).$$

3. Some execution methods

Throughout the rest of the paper it will be convenient to identify a tree as a function, with its graph. In other words, 'ct(α, s) is infinite' will replace 'dom(ct(α, s)) is infinite', '$\{(\lambda, u), (0, v)\} \subset \mathrm{ct}(\alpha, s)$' will replace '$\mathrm{ct}(\alpha, s)(\lambda) = u$ and $\mathrm{ct}(\alpha, s)(0) = v$', etc.

Our first method, which we call *depth search* and denote by D, is obtained by imagining a mechanism starting from λ, the root of ct(α, s), and proceeding along a single path down the tree, making nondeterministic choices at branch points. This process terminates when any leaf is reached; the label of that leaf is the resulting state, the computation being a failure if the leaf is labelled F. The process does not terminate if the path chosen is infinite, i.e., a divergence.

In terms of the previous section, $D(\alpha, s)$ is the set of all 'maximal downward' paths from the root of $ct(\alpha, s)$, i.e., all infinite paths and all paths ending in leaves. For example, if $\{(\lambda, s), (i_1, s_1), (i_1i_2, s_2), \ldots, (i_1 \cdots i_k, s_k)\} \subset ct(\alpha, s)$ and if $i_1 \cdots i_k$ is a leaf, then $\{(\lambda, \lambda), (0, i_1), (00, i_1i_2), \ldots, (0^k, i_1 \cdots i_k\} \in D(\alpha, s)$.

Let us now characterize the states in $wp_D(\alpha, Q)$ and thus understand the meaning of D-total correctness. Let $Q \subset S$. It is clear that for every $H \in D(\alpha, s)$, $fin_H(s)$ is either empty (if H corresponds to a failure or a divergence) or a singleton (otherwise), and that

$$\bigcup_{H \in D(\alpha, s)} fin_H(s) = final_\alpha(s).$$

Hence, the requirement, for every such H, that $fin_H(s) \cap Q \neq \phi$, is equivalent to requring that $|fin_H(s)| = 1$, and that the single element of $fin_H(s)$ be an element of Q. Since any failure or divergence in $ct(\alpha, s)$ would contribute an H to $D(\alpha, s)$ violating this, it follows that this requirement amounts to requiring that $ct(\alpha, s)$ be free of failures and divergences, and that, furthermore, all states in $final_\alpha(s)$ be in Q. The requirement that there be a bound on the length of paths in each $H \in D(\alpha, s)$ is subsumed by the above since here H can be unbounded iff it is infinite, iff there is an infinite path in $ct(\alpha, s)$, i.e., a divergence. To summarize, we have

Lemma 3.1. *For every* α *and* $Q \subset S$,

$$wp_D(\alpha, Q) = \{s \mid final_\alpha(s) \subset Q \wedge fail_\alpha(s) = 0 \wedge loop_\alpha(s) = 0\}.$$

In other words, α is D-totally correct with respect to P and Q iff α, started in any state satisfying P, cannot diverge or fail, and furthermore, every possible finite state satisfies Q. This description of wp_D is quite intuitive; using depth search, a path is chosen 'blindly' and following it might lead to a failure or a divergence, or alternatively, to any one of the final states of α in s.

The second method, a variation of D, is *depth first with backtracking*, DT. Here too elements of $DT(\alpha, s)$ will be degenerate, 1-degree trees, i.e., paths. The method is obtained by imagining a mechanism similar to the previous one moving down the tree. Here though, if it reaches an F-leaf it backtracks to the most recent branch point and pursues a previously unselected alternative to the one it came from. If no such alternative branch exists it backs up to the next recent point, etc. If it returns in this manner to the root λ, having exhausted all its branches, the procedure terminates unsuccessfully. This last situation can occur if and only if $ct(\alpha, s)$ is finite and all its leaves are labelled with F.

$DT(\alpha, s)$ consists of paths of the following three possible types: infinite ones, finite ones ending in nodes corresponding to termination leaves of $ct(\alpha, s)$, and finite ones ending in nodes corresponding to the root. As remarked, the latter case occurs iff both others do not. As an example, if $s_2 \in S$ and $\{(\lambda, s), (0, \mathcal{F}), (1, s_1), (10, s_2)\} \subset ct(\alpha, s)$ and 10 is a leaf (here 10 is the sequence consisting of 1 followed by 0), then $\{(\lambda, \lambda), (0, 0), (00, \lambda), (000, 1), (0000, 10)\} \in DT(\alpha, s)$. Here too, we note that $\bigcup_{H \in DT(\alpha, s)} fin_H(s) = final_\alpha(s)$ and that for each $H \in DT(\alpha, s)$, $|fin_H(s)| \leq 1$. Let us now characterize $wp_{DT}(\alpha, Q)$. As before, any infinite path in $ct(\alpha, s)$ contributes an infinite traversal to $DT(\alpha, s)$, violating the boundedness property. However, another phenomenon has the same effect: suppose some node u of $ct(\alpha, s)$ is of infinite degree, and infinitely many (though not necessarily all) of its sons have the property that the subtree rooted in them is finite and all of its leaves are labelled $\mathcal{F}$. Since u is reachable from the root λ, an infinite traversal H of $ct(\alpha, s)$ exists, corresponding to the mechanism selecting as alternatives at u only sons of the aforementioned kind. The traversal finds only $\mathcal{F}$-leaves on each subtree and backtracks to u infinitely often. Hence H is infinite and $H \in DT(\alpha, s)$. Calling such a node u an *∞-failure*, we see that requiring that no $H \in DT(\alpha, s)$ be unbounded amounts to requring that $loop_\alpha(s) = 0$ and also that $ct(\alpha, s)$ has no ∞-failures.

Turning to the $fin_H(s) \cap Q \neq \phi$ requirement, any unbounded H violates it, as does the presence of any $s' \in final_\alpha(s) - Q$ (such an s' will be the state corresponding to the single leaf of at least one finite traversal in $DT(\alpha, s)$). Moreover, if $ct(\alpha, s)$ is finite and all of its leaves labelled with $\mathcal{F}$, $DT(\alpha, s)$ will consist, as remarked above, of finite traversals H with $fin_H(s) = \phi$. These cases exhaust all the possibilities of $H \in DT(\alpha, s)$ violating $fin_H(s) \cap Q = \phi$. In particular, a failure in $ct(\alpha, s)$ does not necessarily imply $s \notin wp_{DT}(\alpha, Q)$, although it does imply $s \notin wp_D(\alpha, Q)$. We thus conclude

Lemma 3.2. *For every* α *and* $Q \subset S$,

$$wp_{DT}(\alpha, Q) = \{s \mid final_\alpha(s) \neq \phi \wedge final_\alpha(s) \subset Q \wedge loop_\alpha(s) = 0 \wedge ct(\alpha, s) \textit{ has no } \infty\textit{-failures}\}.$$

Since a tree which is of finite nondeterminism cannot have ∞-failures, we also have

Lemma 3.3. *For every* α *of finite nondeterminism and* $Q \subset S$,

$$wp_{DT}(\alpha, Q) = \{s \mid final_\alpha(s) \neq \phi \wedge final_\alpha(s) \subset Q \wedge loop_\alpha(s) = 0\}.$$

These characterizations are also intuitively clear; using backtracking, a failure in $ct(\alpha, s)$ does not cause a failure in execution and hence $fail_\alpha(s) = 0$ is not required. Note that, since $fail_\alpha(s) = 0 \wedge loop_\alpha(s) = 0$ implies that $final_\alpha(s) \neq \phi$, $wp_D(\alpha, Q) \subset wp_{DT}(\alpha, Q)$ for any α of finite nondeterminism.

Many other single-path execution methods are plausible, including methods which allow backtracking only to some fixed height, or methods of predetermined behaviour such as, e.g., *left-search*, in which the leftmost branch is always chosen. One traversal is then associated with $ct(\alpha, s)$, namely the 0-path (λ, λ), $(0, 0)$, $(00, 00)$. . . , it being finite or infinite depending on the case at hand. Note that in this example not all elements of $final_\alpha(s)$ are 'covered' by some traversal H.

Let us now describe two methods which produce traversals which are trees (i.e., they are not single paths), and which, for programs of finite nondeterminism, will turn out to be dual to D and DT. The first, *breadth search*, B, is obtained by imagining a mechanism moving down through $ct(\alpha, s)$, but splitting up and pursuing all branches encountered simultaneously. The relative speeds of the various processors assigned to the different paths by the mechanism are assumed not to be known, nor indeed whether or not the simultaneity is real; it is possible for the paths to be pursued by advancing a step at a time at some nondeterministically chosen branch. We do, however, assume *fairness* of a sort, in that at each moment every possible path has the property that it will be advanced upon eventually. In other words, no path will be followed to an infinite depth while another is followed only to a finite depth. The process terminates when at least one leaf of $ct(\alpha, s)$ is reached.

In order to characterize wp_B, we note that $B(\alpha, s)$ will consist of

(a) precisely one tree, an infinite one,

(b) infinitely many trees of bounded depth, or

(c) finitely many trees of bounded depth,

according to whether

(a) $ct(\alpha, s)$ has no leaves at all,

(b) $ct(\alpha, s)$ has leaves and also an infinite path, or

(c) $ct(\alpha, s)$ has leaves but no infinite path,

respectively.

The reason there are infinitely many trees in case (b) is that we must account, in $B(\alpha, s)$, for every possible advance along the infinite paths before reaching a leaf. For example, if $\{(\lambda, s), (1, s'), (0, s_1), (00, s_2), \ldots, (0^k, s_k), \ldots\} \subset ct(\alpha, s)$ and 1 is a leaf, then for every k, $\{(\lambda, \lambda), (1, 1), (0, 0), (00, 00), \ldots, (0^k, 0^k)\} \in B(\alpha, s)$, but, say, $\{(\lambda, \lambda), (0, 0), (00, 00), \ldots, (0^k, 0^k)\} \notin B(\alpha, s)$. Hence, in order to satisfy the boundedness requirement for every $H \in B(\alpha, s)$, we must require that $ct(\alpha, s)$ has at least one leaf; equivalently, $fail_\alpha(s) = 1 \vee final_\alpha(s) \neq \phi$. In this case we also have $\bigcup_{H \in B(\alpha,s)} fin_H(s) = final_\alpha(s)$. Note now that for every $s' \in final_\alpha(s)$ there is an $H \in B(\alpha, s)$ such that $fin_H(s) = \{s'\}$. Also, if $ct(\alpha, s)$ has an $\boldsymbol{F}$-leaf, then there is an $H \in B(\alpha, s)$ such that $fin_H(s) = \phi$, e.g., a traversal with one leaf corresponding to that $\boldsymbol{F}$-leaf. It follows from these remarks that $fin_H(s) \cap Q \neq \phi$ for every $H \in B(\alpha, s)$ reduces to requiring that $fail_\alpha(s) = 0 \wedge final_\alpha(s) \subset Q$. And so we have:

Lemma 3.4. *For every* α *and* $Q \subset S$,

$$wp_B(\alpha, Q) = \{s \mid final_\alpha(s) \neq \phi \wedge final_\alpha(s) \subset Q \wedge fail_\alpha(s) = 0\}.$$

The intuition here is also clear; the breadth search method is fair, that is, an infinite path will never be taken if the tree has at least one leaf. On the other hand, since $\boldsymbol{F}$-leaves are leaves and cause a traversal to terminate unsuccessfully, they have to be outlawed.

The latter problem can be eliminated from the breadth search method in a way analogous to the overriding of $\boldsymbol{F}$-leaves in depth search by backtracking. Here we simply ignore $\boldsymbol{F}$-leaves, terminating our breadth search only upon reaching a non-$\boldsymbol{F}$, i.e., a termination leaf. Otherwise, this method, which we call *breadth search with ignoring*, BG, is the same as B. In order to characterize $\mathrm{wp_{BG}}$, note that case (a) above can also occur when $\mathrm{ct}(\alpha, \mathrm{s})$ has infinite paths but has only $\boldsymbol{F}$-leaves. We leave the reader to convince himself that whenever $\mathrm{final}_\alpha(\mathrm{s}) \neq \phi$, each $\mathrm{H} \in \mathrm{BG}(\alpha, \mathrm{s})$ will be of bounded depth and $\mathrm{fin_H(s)} \neq \phi$. Hence, all we need in addition, to ensure the properties required of the weakest precondition is $\mathrm{final}_\alpha(\mathrm{s}) \subset \mathrm{Q}$ (for the same reasons as in method B). And so,

Lemma 3.5. *For every α and* $\mathrm{Q} \subset \mathrm{S}$,

$$\mathrm{wp_{BG}}(\alpha, \mathrm{Q}) = \{\mathrm{s} \mid \mathrm{final}_\alpha(\mathrm{s}) \neq \phi \wedge \mathrm{final}_\alpha(\mathrm{s}) \subset \mathrm{Q}\}.$$

Variations of breadth search are also possible, in which the number of processors available is limited, as in *k-breadth search*, kB. Every traversal in $k\mathrm{B}(\alpha, \mathrm{s})$ has no more than k paths. Also, combinations of depth and breadth search are possible, say by taking 2-breadth search and allowing backtracking on reaching $\boldsymbol{F}$-leaves. One might also allow a 'clever' backtracking feature in which a final state s′ is tested for some important property Q and backtracking occurs if $\mathrm{s}' \notin \mathrm{Q}$. Here Q is 'built-in' to the execution method itself rather than being a property of interest in the context of a particular program only. We do not further elaborate on these methods here. As mentioned earlier, through, a more refined definition of what an execution method is would be necessary if it is to exclude impractical and useless methods which the reader can no doubt see our definition admits. Some relevant results and remarks about the computability of various kinds of nondeterminism appear in [4].

The four methods D, DT, B, and BG will be used in the next section to analyze the properties required by Dijkstra [5, 6] for his nondeterministic language of guarded commands. We summarize the results of this section needed in the next, urging the reader to convince himself that these hold also when (2) is replaced by (2′) in the definition of total correctness in Section 2.

Theorem 3.1. *Let* L *be a programming language of bounded nondeterminism. For any* $\alpha \in \mathrm{L}$ *and for any* $\mathrm{Q} \subset \mathrm{S}$,

$$\begin{aligned}
\mathrm{s} \in \mathrm{wp_D}(\alpha, \mathrm{Q}) \quad &\textit{iff} \quad \mathrm{final}_\alpha(\mathrm{s}) \neq \phi \wedge \mathrm{final}_\alpha(\mathrm{s}) \subset \mathrm{Q} \wedge \mathrm{fail}_\alpha(\mathrm{s}) = 0 \wedge \mathrm{loop}_\alpha(\mathrm{s}) = 0,\\
\mathrm{s} \in \mathrm{wp_{DT}}(\alpha, \mathrm{Q}) \quad &\textit{iff} \quad \mathrm{final}_\alpha(\mathrm{s}) \neq \phi \wedge \mathrm{final}_\alpha(\mathrm{s}) \subset \mathrm{Q} \wedge \qquad\qquad \mathrm{loop}_\alpha(\mathrm{s}) = 0,\\
\mathrm{s} \in \mathrm{wp_B}(\alpha, \mathrm{Q}) \quad &\textit{iff} \quad \mathrm{final}_\alpha(\mathrm{s}) \neq \phi \wedge \mathrm{final}_\alpha(\mathrm{s}) \subset \mathrm{Q} \wedge \mathrm{fail}_\alpha(\mathrm{s}) = 0,\\
\mathrm{s} \in \mathrm{wp_{BG}}(\alpha, \mathrm{Q}) \quad &\textit{iff} \quad \mathrm{final}_\alpha(\mathrm{s}) \neq \phi \wedge \mathrm{final}_\alpha(\mathrm{s}) \subset \mathrm{Q}.
\end{aligned}$$

Thus, our four methods are complementary in the sense that besides the reasonable requirements on the set of final states of α in s, they represent the four possibilities of allowing or disallowing failures and divergences. It follows, of course, that (omitting α and Q) $\mathrm{wp_D} = (\mathrm{wp_{DT}} \cap \mathrm{wp_B} \cap \mathrm{wp_B})$, and $(\mathrm{wp_D} \cup \mathrm{wp_{DT}} \cup \mathrm{wp_B}) \subset \mathrm{wp_{BG}}$.

The reader familiar with *dynamic logic* (see e.g., [7, 14]) will recognize the correspondence of concepts: $\text{final}_\alpha(s) \neq \phi$, $\text{final}_\alpha(s) \subset Q$, $\text{fail}_\alpha(s) = 0$ and $\text{loop}_\alpha(s) = 0$ are, respectively, $s \vDash \langle\alpha\rangle \mathit{true}$, $s \vDash [\alpha]Q$, $s \vDash \text{fail}_\alpha$ and $s \vDash \text{loop}_\alpha$ (the latter two from [7]). Moreover, Lemma 3.3 for example, by virtue of it holding for every S, simply states the following *validity*:

$$\vDash (\text{wp}_{\text{DT}}(\alpha, Q) \equiv (\langle\alpha\rangle \mathit{true} \wedge [\alpha]Q \wedge \neg \text{loop}_\alpha)).$$

4. Guarded commands

In this section, a particular programming language, the *guarded commands* language, GC, of Dijkstra [5] will be described as a tool for presenting the results of the next section. The syntax of GC will be defined, assuming a given set Δ of primitive operations. Then, given an arbitrary set of states S and meanings for the elemnts of Δ, the computation tree, $\text{ct}(\alpha, s)$, will be defined for every $\alpha \in \text{GC}$ and $s \in S$. (The computation trees of a somewhat more general language, that of regular programs over Δ, have been defined in [7, 8].)

Let Δ be a set of symbols. Elements of Δ, denoted by a, b, . . . , will be called *primitive programs*, and can be thought of as assignment statements. The set of programs GC is defined inductively as follows:

(1) any primitive program $a \in \Delta$ is in GC,
(2) for any α and β in GC, $(\alpha ; \beta)$ is in GC,
(3) for any $P \subset S$, $R \subset S$ and α and β in GC,

$$\text{IF } P \to \alpha \,\Box\, R \to \beta \text{ FI} \quad \text{and} \quad \text{DO } P \to \alpha \,\Box\, R \to \beta \text{ OD}$$

are in GC.

Let an arbitrary set of states S be given, along with an interpretation of the elements of Δ as functions from states to states; i.e., $\bar{a} : S \to S$ is given for every $a \in \Delta$. $\bar{a}(s)$ can be thought of as the state in which one ends up when carrying out program a in state s. We assume throughout that there is some element in Δ, which we denote by skip, the interpretation of which is the identity function on S.

Intuitively, $(\alpha ; \beta)$ corresponds to α followed by β. $\text{IF } P \to \alpha \,\Box\, R \to \beta \text{ FI}$ corresponds to the program obtained by testing P and R; if P holds α can be executed, if R holds β can, if both hold either α or β are executed (the choice being nondeterministic) and if neither hold the program fails. $\text{DO } P \to \alpha \,\Box\, R \to \beta \text{ OD}$ corresponds to repeatedly executing $\text{IF } P \to \alpha \,\Box\, R \to \beta \text{ FI}$ while either P or R (or both) hold, and terminating properly when neither hold. The reader might want to view it as *while* $P \vee R$ *do* $\text{IF } P \to \alpha \,\Box\, R \to \beta \text{ FI}$ *od*. Note that one can generalize by allowing the elements of Δ to be interpreted as nondeterministic primitive programs, e.g., by letting $\bar{a}(s) \subset S$. However, since GC is presented here merely as an example of a language and since Dijkstra [5] has as primitive operations only (deterministic) assignments, we leave it as it is.

Define the computation trees ct(α, s), for $\alpha \in$ GC and s $\in$ S, inductively as follows:

(1) if a $\in \Delta$, then ct(a, s) $= \{(\lambda, s), (0, \bar{a}(s))\}$,

(2) for any α, β, $\in$ GC, ct((α; β), s) = ct(α, s) $\cup \{(uv, l) \mid (u, s') \in ct(\alpha, s)$ for some $s' \in S$, u is a leaf of ct(α, s) and $(v, l) \in ct(\beta, s')\}$,

(3) for any P $\subset$ S, R $\subset$ S and $\alpha, \beta \in$ GC,

$$ct(\text{IF } P \rightarrow \alpha \,\square\, R \rightarrow \beta \text{ FI}, s) = \begin{cases} \{(\lambda, s), (0, \boldsymbol{F})\}, & s \notin P \cup R, \\ ct(\alpha, s), & s \in P - R, \\ ct(\beta, s), & s \in R - P \\ \{(\lambda, s)\} \cup \{(0u, l) \mid (u, l) \in ct(\alpha, s)\} & \\ \qquad \cup \{(1u, l) \mid (u, l) \in ct(\beta, s)\}, & s \in P \cap R, \end{cases}$$

$$ct(\text{DO } P \rightarrow \alpha \,\square\, R \rightarrow \beta \text{ OD}, s) = \bigcup_{n=0}^{\infty} A_n,$$

where $A_0 = \{(\lambda, s)\}$ and $A_{n+1} = \{(uv, l) \mid (u, s') \in A_n$ for some $s' \in P \cup R$, u is a leaf of A_n and $(v, l) \in ct(\text{IF } P \rightarrow \alpha \,\square\, R \rightarrow \beta \text{ FI}, s')\}$.

We now show that this definition for GC conforms to the requirements of computation trees in the previous section.

Theorem 4.1. *For any* $\alpha \in$ GC *and* s $\in$ S, ct(α, s) *is indeed a computation tree.*

Proof. The only part of the definition of a computation tree in Section 2 which is not trivially verified for the trees of GC is the fact that ct(α, s) is a function, i.e., that if $(u, l) \in ct(\alpha, s)$ and $(u, l') \in ct(\alpha, s)$, then $l = l'$. This, however, follows quite easily from the inductive definition of ct; the two cases where an overlap might have been possible are when some (u, s') is present in one tree and, for some (v, l) in another, (uv, l) is added. In both these cases (i.e., in the definitions for (α; β) and for A_{n+1}),

(a) u is a leaf of the first tree and hence, for v $\neq \lambda$, uv was not a node of that tree at all, and

(b) for v $= \lambda$, the label of v in the second tree is s', so that (u, s') is simply 'added' to (u, s'), with no effect.

Note that ct(α, s) is of bounded nondeterminism, each node being of degree ≤ 2.

The intuition behind programs in GC is described in greater detail in [6]. Failures occur when an IF statement is reached and both P and R are 'false' in the current state s', i.e., $s' \notin P \cup R$. Divergences occur when the 'body' IF $P \rightarrow \alpha \,\square\, R \rightarrow \beta$ FI of a DO statement can be repeatedly executed without reaching a state $s' \notin P \cup R$.

Denote by FAIL the always failing program IF $\phi \rightarrow$ skip$\square$ $\phi \rightarrow$ skip FI, and by LOOP the always diverging program DO S $\rightarrow$ skip$\square$ S $\rightarrow$ skip OD. Note that for every s $\in$ S, ct(skip, s) $= \{(\lambda, s), (0, s)\}$ and ct(FAIL, s) $= \{(\lambda, s), (0, \boldsymbol{F})\}$. Also, ct(LOOP, s) is an infiite tree with no finite paths.

We close this section with an example of a very simple set of states and primitive program in GC which will serve us in the next section. Define

$$S_N = \{(i, j) \mid i, j \in N\},$$

i.e., the set of all pairs of natural numbers. Think of (i, j) as being the current values of 'variables' x and y. The primitive operations (in Δ) that we use are: $x \leftarrow 0$, $y \leftarrow 0$, $x \leftarrow x+1$ and $y \leftarrow y+1$, defined respectively by $\overline{x \leftarrow 0}\,(i, j) = (0, j)$, $\overline{y \leftarrow 0}\,(i, j) = (i, 0)$, $\overline{x \leftarrow x+1}\,(i, j) = (i+1, j)$, and $\overline{y \leftarrow y+1}\,(i, j) = (i, j+1)$. For any fixed n we let $x \leftarrow n$ abbreviate the program $(x \leftarrow 0; x \leftarrow x+1; \ldots; x \leftarrow x+1)$ with n appearances of $x \leftarrow x+1$, and similarly for $y \leftarrow n$. Thus, e.g., $\overline{x \leftarrow n}\,(i, j) = (n, j)$. Finally, define $[x = n]$ to be the set $\{(n, j) \mid j \in N\}$ and $[y = n]$ to be $\{(i, n) \mid i \in N\}$. In the sequel we will freely use similar sets such as $[x \leq n]$.

5. Properties of weakest preconditions

In [5, 6], Dijkstra introduced the language of guarded commands and defined, by induction on the structure of α, a set $wp(\alpha, Q)$ for every $\alpha \in GC$ and $Q \subset S$. In the words of [6]: "We shall use the notation $wp(\alpha, Q)$ to denote the weakest precondition for the initial state of the system such that activation of α is guaranteed to lead to a properly terminating activity leaving the system in a final state satisfying the post condition Q." One of the purposes of this section is to illustrate that the key word in this description is 'activation', by analyzing the wp of [5, 6] in view of four different methods of activation, or execution.

Dijkstra postulated four *healthiness properties* for weakest preconditions, which are to hold for every choice of a set of states S, every $\alpha \in GC$ and every P, $Q \subset S$:

H1: $wp(\alpha, \phi) = \phi$,
H2: If $P \subset Q$, then $wp(\alpha, P) \subset wp(\alpha, Q)$,
H3: $wp(\alpha, P \cap Q) = wp(\alpha, p) \cap wp(\alpha, Q)$,
H4: (Continuity) If $P_0, P_1, \ldots$ are subsets of S such that $(\forall n)(P_n \subset P_{n+1})$, then $wp(\alpha, \bigcup_n P_n) = \bigcup_n wp(\alpha, P_n)$.

Our first result concerns the extent to which these properties define a unique notion of wp, even for the language GC. It is shown that the weakest preconditions corresponding to methods D and DT both comply with all these requirements. Next, we present Dijkstra's definition of $wp(\alpha, Q)$ for $\alpha \in GC$, and show that this definition 'assumes' method D, in the sense that $wp(\alpha, Q) = wp_D(\alpha, Q)$ for every $\alpha \in GC$ and $Q \subset S$, and for every S, and that in general $wp \neq wp_X$ for $X \in \{DT, B, BG\}$.

Works related to the results of this section are those of de Roever [15], de Bakker [1], Wand [17], Plotkin (see [16]) and Hoare [10]. In particular, many of the entries of the tables in Theorems 5.1 and 5.2 below for methods D and DT, have been established independently in [10] using a logic of traces (which correspond to single-path traversals). All the results of this section appear, in somewhat different form, in [7].

Returning to properties H1–H4, for $X \in \{D, DT, B, BG\}$ we say that Hi *holds of* X if, for every S, $\alpha \in GC$, interpretations for the elements of Δ, and P, $Q \subset S$, property Hi is true when wp is replaced by wp_X.

Theorem 5.1.. *Let* 1 *in column* X *mean that the properties in the corresponding row hold of* X, *and let* 0 *mean otherwise. Then the following table summarizes the situation*:

	D	DT	B	BG
H1–H3	1	1	1	1
H4	1	1	0	0

Proof. (i) H1–H3 hold of D, DT, B and BG:

We show H1 and leave H2 and H3 to the reader. Since for any $X \in \{D, DT, B, BG\}$,

$$wp_X(\alpha, Q) \subset \{s \mid final_\alpha(s) \neq \phi \wedge final_\alpha(s) \subset Q\},$$

and $final_\alpha(s) \subset \phi$ is the same as $final_\alpha(s) = \phi$, it follows that

$$wp_X(\alpha, \phi) \subset \{s \mid final_\alpha(s) \neq \phi \wedge final_\alpha(s) = \phi\} = \phi.$$

(ii) H4 does not hold of B or BG, i.e., there is a set of states S, $\alpha \in BG$ and $P_0, P_1, \ldots \subset S$ such that H4 is not true for B or BG:

Take $S = S_N$ and, for every n, P_n to be $[x \leq n]$, i.e., $\{(i, j) \mid j \in N, i \leq n\}$. Certainly, $P_n \subset P_{n+1}$. Now α will be the program 'set x to any natural number', which in GC can be written as

$$\alpha : ((x \leftarrow 0; y \leftarrow 0); \text{DO } \phi \rightarrow \text{skip} \,\Box\, [y = 0]$$
$$\rightarrow (\text{IF } S_N \rightarrow (y \leftarrow 1) \,\Box\, S_N \rightarrow (x \leftarrow x + 1) \text{ FI) OD})$$

(This program, in the terminology of [7, 14] and others, is simply $x \leftarrow 0$; $(x \leftarrow x + 1)^*$; $y \leftarrow 1$.) It can be shown that $final_\alpha(i, j) = \{(k, 1) \mid k \in N\}$, and that $fail_\alpha(i, j) = 0$. Both these remarks are true for any $(i, j) \in S_N$, and in addition, $\bigcup_n P_n = S_n$, so that $final_\alpha(i, j) \subset \bigcup_n P_n$ and $wp_B(\alpha, \bigcup_n P_n) = wp_{BG}(\alpha, \bigcup_n P_n) = S_N$. However, $wp_B(\alpha, P_n) = wp_{BG}(\alpha, P_n) = \phi$ for every n, the reason being that $\{(k, 1) \mid k \in N\}$ is a subset of no set of the form $[x \leq n]$. Hence,

$$\bigcup_n wp_B(\alpha, P_n) = \bigcup_n wp_{BG}(\alpha, P_n) = \phi.$$

(iii) H4 holds for D and DT:

Let $P_0, P_1, \ldots$ be subsets of S such that $(\forall n)\ (P_n \subset P_{n+1})$. The claim will follow immediately from the fact that

$$\left\{s \mid loop_\alpha(s) = 0 \wedge final_\alpha(s) \subset \bigcup_n P_n\right\} = \bigcup_n \{s \mid loop_\alpha(s) = 0 \wedge final_\alpha(s) \subset P_n\}.$$

Clearly, the first set contains the second. Let s be a state in the first set. For $s' \in final_\alpha(s)$, denote by $k(s')$ the least integer i such that $s' \in P_i$. Since $ct(\alpha, s)$ is of finite nondeterminism and has no infinite paths, by König's infinity lemma it is finite. Hence so is $final_\alpha(s)$. Thus, by the nondecreasing property of $\{P_n\}$, letting $n_0 = \max\{k(s') \mid s' \in final_\alpha(s)\}$, we have $final_\alpha(s) \subset P_{n_0}$.

We now present the inductive definition of wp(α, Q) which was given in [5]. We rewrite the definition in the form of equivalences, so that proving wp = $\mathrm{wp_X}$ for some $X \in \{D, DT, B, BG\}$ will amount to showing that these equivalences hold when wp is replaced by $\mathrm{wp_X}$. Let $\bar{P}$ stand for S − P.

E1: $\mathrm{wp(skip, Q) = Q}$,
E2: $\mathrm{wp(FAIL, Q)} = \phi$,
E3: $\mathrm{wp(a, Q)} = \{s \mid \bar{a}(s) \in Q\}$ for $a \in \Delta$,
E4: $\mathrm{wp}((\alpha ; \beta), Q) = \mathrm{wp}(\alpha, \mathrm{wp}(\beta, Q))$,
E5: $\mathrm{wp}(\mathrm{IF}\ P \to \alpha \,\square\, R \to \beta\ \mathrm{FI}, Q) = (P \cup R) \cap (\bar{P} \cup \mathrm{wp}(\alpha, Q)) \cap (\bar{R} \cup \mathrm{wp}(\beta, Q))$,
E6: $\mathrm{wp}(\mathrm{DO}\ P \to \alpha \,\square\, R \to \beta\ \mathrm{OD}, Q) = \bigcup_{n=0}^{\infty} H_n$,

where $H_0 = \bar{P} \cap \bar{R} \cap Q$, and $H_{n+1} = H_0 \cup \mathrm{wp}(\mathrm{IF}\ P \to \alpha \,\square\, R \to \beta\ \mathrm{FI}, H_n)$.

Theorem 5.2. *Let* 1 *and* 0 *have meanings as in Theorem* 5.1. *Then the following table summarizes the situation*:

	D	DT	B	BG
E1–E3	1	1	1	1
E4–E6	1	0	0	0

Proof. (i) E1–E3 hold for D, DT, B, BG:

(E1): By definition, $\mathrm{ct(skip, s)} = \{(\lambda, s), (0, s)\}$ so that $\mathrm{final_{skip}(s) = s}$ and $\mathrm{loop_{skip}(s) = fail_{skip}(s) = 0}$. It follows that $\mathrm{wp_X(skip, Q)} = \{s \mid \mathrm{final_{skip}}(s) \subset Q\} = \{s \mid s \in Q\} = Q$.

(E2): By definition, $\mathrm{ct(FAIL, s)} = \{(\lambda, s), (0, \boldsymbol{F})\}$, so that $\mathrm{final_{FAIL}}(s) = \phi$, implying also $\mathrm{wp_X(FAIL, Q)} = \phi$.

(E3): Since $\mathrm{loop_a(s) = fail_a(s) = 0}$ and $\mathrm{final_a}(s) = \bar{a}(s)$ for every $a \in \Delta$, E3 follows.

(ii) E4 does not hold for DT, B or BG:

Again, let $S = S_N$, $Q = S$ and let α be $\mathrm{IF}\ S \to (x \leftarrow 1) \square\, S \to (x \leftarrow 2)\ \mathrm{FI}$. For each of DT, B and BG we define a program $\beta \in \mathrm{GC}$ such that α, β and Q violate E4.

(DT): Let β be $\mathrm{IF}\ [x = 1] \to \mathrm{skip} \square [x = 1] \to \mathrm{skip\ FI}$. Note that $\mathrm{ct}(\alpha, (i, j))$ has no failures or divergences and that $\mathrm{final}_\alpha(i, j) = \{(1, j), (2, j)\}$. Consequently, $\mathrm{final}_{(\alpha;\beta)}(i, j) = \{(1, j)\}$ and $\mathrm{loop}_{(\alpha;\beta)}(i, j) = 0$. Since $\mathrm{final}_{(\alpha;\beta)}(i, j) \subset S$ it follows that $\mathrm{wp_{DT}}((\alpha ; \beta), Q) = \mathrm{wp_{DT}}((\alpha ; \beta), S) = S$. However

$$\mathrm{wp_{DT}}(\alpha, \mathrm{wp_{DT}}(\beta, S)) =$$
$$= \{s \mid \mathrm{final}_\alpha(s) \neq \phi \wedge \mathrm{loop}_\alpha(s) = 0 \wedge \mathrm{final}_\alpha(s) \subset$$
$$\{s' \mid \mathrm{final}_\beta(s') \neq \phi \wedge \mathrm{loop}_\beta(s') = 0 \wedge \mathrm{final}_\beta(s') \subset S\}\}$$
$$= \{(i, j) \mid \{(1, j), (2, j)\} \subset \{(k, l) \mid \mathrm{final}_\beta(k, l) \neq \phi\}\} = \phi.$$

The last equality follows from $\mathrm{final}_\beta(2, j) = \phi$.

(B:) Let β be $\mathrm{DO}\ [x = 1] \to \mathrm{skip} \square [x = 1] \to \mathrm{skip\ OD}$. Similarly to the previous case one can show that since $\mathrm{final}_\beta(1, j) = \phi$ but $\mathrm{fail}_\beta(1, j) = 0$, $\mathrm{wp_B}((\alpha ; \beta), S) = S$ but $\mathrm{wp_B}(\alpha, \mathrm{wp_B}(\beta, S)) = \phi$.

(BG:): Let β be any one of the above two. The rest of the reasoning is similar.

(iii) E4 holds for D:

We first state the following two facts which can easily be established for any S, $s \in S$ and $\alpha, \beta, \in GC$, using the definition of $ct(\alpha; \beta, s)$:

$$loop_{(\alpha;\beta)}(s) = 0 \quad \text{iff} \quad loop_\alpha(s) = 0 \wedge (\forall s' \in final_\alpha(s))(loop_\beta(s') = 0),$$

$$fail_{(\alpha;\beta)}(s) = 0 \quad \text{iff} \quad fail_\alpha(s) = 0 \wedge (\forall s' \in final_\alpha(s))(fail_\beta(s') = 0).$$

Now $s \in wp_D((\alpha;\beta), Q)$ iff $(final_{(\alpha;\beta)}(s) \neq \phi \wedge final_{(\alpha;\beta)}(s) \subset Q \wedge fail_{(\alpha;\beta)}(s) = 0 \wedge loop_{(\alpha;\beta)} = 0)$ iff

$$(\exists s' \in final_\alpha(s))(final_\beta(s') \neq \phi) \wedge fail_\alpha(s) = 0 \wedge loop_\alpha(s) = 0$$
$$\wedge (\forall s' \in final_\alpha(s))(final_\beta(s') \subset Q \wedge fail_\beta(s') = 0 \wedge loop_\beta(s') = 0). \qquad (1)$$

We have to show that (1) holds iff

$$final_\alpha(s) \neq \phi \wedge fail_\alpha(s) = 0 \wedge loop_\alpha(s) = 0 \wedge (\forall s' \in final_\alpha(s))(final_\beta(s') \neq \phi$$
$$\wedge final_\beta(s') \subset Q \wedge fail_\beta(s') = 0 \wedge loop_\beta(s') = 0) \qquad (2)$$

iff $s \in wp_D(\alpha, wp_D(\beta, Q))$. All that is needed for this is to show that, under the assumption (1), $final_\beta(s') \neq \phi$ for every $s' \in final_\alpha(s)$. This follows from the fact that for each $s' \in final_\alpha(s)$, $ct(\beta, s')$ is a nonempty tree free of failures and divergences.

(iv) E5 does not hold for DT, B or BG:

Taking $S = S_N$, $P = R = Q = S$ and α to be skip, we let β be FAIL, LOOP or either of these for cases DT, B and BG respectively. In each case the left hand side of E5 is equal to S and the right to ϕ. We omit the details.

(v) E5 holds for D:

Straightforward using the definition of ct(IF $P \to \alpha \,\Box\, R \to \beta$ FI, s).

(vi) E6 does not hold for DT, B or BG:

Here, too, there is a general structure to the three counter-examples. Let $S = S_N$ and $Q = S$. Taking γ to be FAIL for the DT case, LOOP for the B case and either for the BG case, we define our program δ : DO $P \to \alpha \,\Box\, R \to \beta$ OD to be DO $[x=0] \to (x \leftarrow x+3) \,\Box\, [x \leq 2] \to (x \leftarrow x+1$; IF $[x=1] \to (x \leftarrow x+1) \,\Box\, [x \neq 1] \to \gamma$ FI) OD. We claim that for any $j \in N$, $(0, j) \in wp_X(\delta, S)$ but $(0, j) \notin \bigcup_n H_n$, where $X \in \{DT, B, BG\}$ and δ contains the appropriate version of γ. Certainly, $(3, j) \in final_\delta(0, j)$, and trivially $final_\delta(0, j) \subset S$. For case DT, there is no divergence in $ct(\delta, (0, j))$ because x will 'reach' the value 3 in at most three iterations and γ is FAIL, not LOOP. For case B, $ct(\delta, (0, j))$ is failure-free because $[x=1] \cup [x \neq 1] = S$ and γ is LOOP, not FAIL. For the BG case these remarks are irrelevant. It follows that $(0, j) \in wp_X(\delta, S)$. Now $H_0 = [x \neq 0] \cap [x > 2] \cap S = \{(i, j) \mid i > 2\}$, so that $(0, j) \notin H_0$. For $(0, j)$ to be in H_1, say for case D, we would have had to have $final_{\delta IF}(0, j) \subset H_0$, where δIF stands for δ with IF $\cdots$ FI replacing DO $\cdots$ OD. However, it is clear that (by carrying out $x \leftarrow x+1$ twice) we have $(2, j) \in final_{\delta IF}(0, j)$, whereas $(2, j) \notin H_0$. One can now show, by induction on k, that if $s \in H_k$ and $k \geq 2$, $final_{IF}(s') \neq \phi$ for every $s' \in final_{IF}(s)$, where IF stands for an arbitrary program of the form IF $P \to \alpha \square R \to \beta$ FI. However, in our case we observe that $(3, j) \in \text{final}_{\delta IF}(0, j)$ but $\text{final}_{\delta IF}(3, j) = \phi$, so that for any $k \geq 2$, $(0, j) \notin H_k$.

(vii) E6 holds for D:

We will show in detail only one direction of this rather tedious proof, the other, being very similar, is omitted. Let π denote an arbitrary program of the form IF $P \to \alpha \square R \to \beta$ FI, and let π^* stand for DO $P \to \alpha \square R \to \beta$ OD. Assume $s \in \text{wp}_D(\pi^*, Q)$ for some $Q \subset S$. We have to show that for every such s, $s \in H_k$ for some k (Note the difficulty here: while there is a correspondence between $A_0, A_1, \ldots$ and $H_0, H_1, \ldots$ in the definitions of $\text{ct}(\pi^*, s)$ and in E6, this correspondence is not perfect; A_{i+1} corresponds to adding an iteration π to A_i as a 'suffix', i.e., this π is the last iteration in A_{i+1}, while in H_{i+1} π is added as a 'prefix', i.e., as the first iteration in H_{i+1}.) Here we are using H_k to denote the set obtained in E6 with wp_D replacing wp. It is an easily-proved fact that the H_i are nondecreasing, in the sense that $H_i \subset H_{i+1}$ for every i. Now since $s \in \text{wp}_D(\pi^*, Q)$, $\text{ct}(\pi^*, s)$ is finite, and we can denote by k(s) the least integer p such that $\text{ct}(\pi^*, s) = \bigcup_{n=0}^{p} A_n$, where the A_n are as in clause (3) in the definition of ct.

Let us now show that for every $s \in \text{wp}_D(\pi^*, Q)$, $s \in H_{k(s)}$, by induction on k(s). If s is such that $k(s) = 0$, then $\text{ct}(\pi^*, s) = A_0 = \{(\lambda, s)\}$ and hence $\text{final}_{\pi^*}(s) = \{s\}$, so that by $s \in \text{wp}_D(\pi^*, Q)$ we know that $s \in Q$. However, from $A_1 = \phi$ it is evident that $s \notin P \cup R$. And so for $k(s) = 0$, $s \in \bar{P} \cap \bar{R} \cap Q = H_0$. Assume that $s \in \text{wp}_D(\pi^*, Q)$, that $k(s) > 0$, and that the claim holds for every s' with $k(s') < k(s)$. Let s' be an arbitrary element of $\text{final}_\pi(s)$. (There is at least one such s' by the assumptions that $k(s) \geq 1$ and $\text{loop}_{\pi^*}(s) = \text{fail}_{\pi^*}(s) = 0$.) Thus, there is a leaf u of $\text{ct}(\pi, s)$ labelled with s'. Clearly then, by the definition of ct, $\text{ct}(\pi^*, s') = \{(v, l) \mid (uv, l) \in \text{ct}(\pi^*, s)\}$. We know that $\text{ct}(\pi^*, s)$ is failure-free, and divergence-free, hence so is $\text{ct}(\pi^*, s')$. Moreover, this implies that $\text{final}_{\pi^*}(s') \neq \phi$. Finally, $\text{final}_{\pi^*}(s') \subset \text{final}_{\pi^*}(s) \subset Q$. Putting these together we have $s' \in \text{wp}_D(\pi^*, Q)$. The characterization of $\text{ct}(\pi^*, s')$ above implies that $k(s') < k(s)$, hence by the inductive hypothesis $s' \in H_{k(s')}$. But by our remark above concerning the sets H_k, since $k(s') \leq k(s) - 1$, it follows that $H_{k(s')} \subset H_{k(s)-1}$. We have shown that $\text{final}_\pi(s) \subset H_{k(s)-1}$. Since $\text{final}_\pi(s) \neq \phi$ and $\text{loop}_{\pi^*}(s) = \text{fail}_{\pi^*}(s) = 0$ (from which we get $\text{loop}_\pi(s) = \text{fail}_\pi(s) = 0$), we obtain $s \in \text{wp}_D(\pi, H_{k(s)-1})$ and hence $s \in H_{k(s)}$.

This completes the proof of the theorem.

Thus, Theorem 5.1 points to the fact that H1–H4 are not adequate for singling out a particular notion of total correctness, and Theorem 5.2 shows one execution method which, for the language GC, is 'behind' E1–E6, in the sense that its corresponding notion of total correctness is the same as that defined by E1–E6.

6. Conclusions

Our results in Section 5 can be viewed as providing rigorous support of the intuition behind the introduction of E1–E6 as rules for 'constructing' totally correct programs in Dijkstra [6]. We have shown that at least one reasonable execution method is consistent with the unique notion of wp defined by E1–E6. However, it seems that in order to be able to define wp for other, perhaps more general programming languages, or to be able to define wp's corresponding to other methods of execution, a general framework such as that suggested in Section 2 and 3 is necessary.

Acknowledgment

Some early ideas leading to this work were developed jointly with V.R. Pratt and appeared in [8]. During that period, we had many helpful discussions with N. Dershowitz and A. Shamir. The detailed remarks and suggestions of A.R. Meyer and M. O'Donnell led to the development of Sections 2 and 3. We are grateful to these individuals and to an anonymous referee for their help. Finally, the results of Section 5 appeared in preliminary form in [7].

References

[1] J.W. de Bakker, Recursive programs as predicate transformers, in: E.J. Neuhold, Ed., *Formal Specifications of Programming Concepts* (North-Holland, Amsterdam, 1978, 165–179.)
[2] A. Blikle, *An Analysis of Programs by Algebraic Means*, Banach Center Publications **2** (Polish Scientific Publishers, Warsaw, 1977).
[3] H.J. Boom, A weaker precondition for loops, Technical Report IW 104/78 Mathematisch Centrum, Amsterdam (1978).
[4] A.K. Chandra, Computable nondeterministic functions, *Proc. 19th IEEE Symposium on Foundations of Computer Science*, Ann Arbor, MI (1978) 127–131.
[5] E.W. Dijkstra, Guarded commands, nondeterminancy and formal derivation of programs, *Comm. ACM* **18** (8) (1975) 453–457.
[6] E.W. Dijkstra, *A Discipline of Programming* (Prentice-Hall, Englewood Cliffs, NJ, 1976).
[7] D. Harel, *First-Order Dynamic Logic*, Lecture Notes in Computer Science **68** (Springer, Berlin, 1979).
[8] D. Harel and V.R. Pratt, Nondeterminism in logics of programs, *Proc. 5th ACM Symposium on Principles of Programming Languages*, Tucson, AZ (1978) 203–213.
[9] P. Hitchcock and D. Park, Induction rules and termination proofs, in: M. Nivat, Ed., *Automata, Languages and Programming* (North-Holland, Amsterdam 1973).
[10] C.A.R. Hoare, Some properties of predicate transformers, *J. ACM* **25**(3) (1978) 461–480.
[11] D. König, *Theorie der endlichen und unendlichen Graphen* (Leipzig, 1936); reprinted (Chelsea, New York, 1950).
[12] R. Kurki-Suonio, Nondeterminism, parallelism and intermittent assertions, *Proc. International Conference on Mathematical Studies of Information Processing*, Kyoto, Japan (1978).
[13] G.D. Plotkin, A powerdomain construction, *SIAM J. Comput.* **5**(3) (1976) 452–487.
[14] V.R. Pratt, Semantical considerations on Floyd-Hoare logic, *Proc. 17th IEEE Symposium on Foundations of Computer Science*, (1976) 109–121.
[15] W.P. de Roever, Dijkstra's predicate transformer, nondeterminism, recursion and termination, in: *Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science, **45** (Springer, Berlin, 1976) 472–481.
[16] W.P. de Roever, Equivalence between Dijkstra's predicate transformer semantics and Smyth's powerdomain semantics as found by G. Plotkin, Manuscript, (1977).
[17] M. Wand, A characterization of weakest preconditions, *J. Comput. System Sci.* **15**(2) (1977) 209–212.