FAIRNESS AND THE AXIOMS OF
CONTROL PREDICATES

by

N. Francez and S. Katz

Technical Report #455
June 1987
Fairness and the Axioms of Control Predicates
Nissim Francez and Shmuel Katz
Department of Computer Science, The Technion, Haifa, Israel

The North has receded, but the South has not yet arrived.
--Reuven Miran, "42 Degrees in the Shade"
Every three lines intersect at a point, if the point is thick enough.
--Folk theorem

Abstract: Many recent axiomatic definitions for structured programming languages include control predicates, which are an abstraction of location counters. The usual axioms identify control locations so as to imply that "no time" is needed to pass from the end of one statement to the next, and in particular from the end of a loop body back to the test at the head of the loop. Although this is reasonable in many contexts, here we focus on difficulties which arise in the context of fair concurrent models (both with message passing and with shared memory concurrency). An axiomatic framework is examined in which it becomes clear that if all the axioms are to be maintained with common representation mappings, there are difficult new requirements which need to be satisfied by an implementation. Several approaches to resolving the difficulty are considered, and in particular it is suggested to replace some axioms of the form $P \Rightarrow Q$ by $P \Rightarrow \text{eventually}(Q)$, where $P$ and $Q$ are control predicates, thereby separating control states previously identified.


Key words and phrases: Fairness, control predicates (at, in, after), concurrency, axiomatic semantics.

Note: A talk based on this paper was presented at the Colloquium on Temporal Logic and Specification, Altrincham, Cheshire, April, 1987.
1. Introduction

In reasoning about the properties of distributed and concurrent programs, it has become customary to introduce control predicates [OL, L2, L3]. These predicates constitute an abstraction of lower-level state variables such as location counters or hidden buffers. Part of the axiomatic definition of a programming language then consists of axioms characterizing the properties of such control predicates. Since control predicates are most commonly used in stating and proving liveness properties, expressing that some progress occurs in the computation, usually these axioms involve temporal operators, such as "eventually". No claim is made about real-time properties by the axioms.

The three most commonly used predicates of this type are $at(S)$, $in(S)$, and $after(S)$, for any program segment $S$. Intuitively, $at(S)$ holds when $S$ is "about to be executed," $in(S)$ holds while $S$’s execution is "in progress," and $after(S)$ holds if $S$’s execution has "just terminated". The axioms for these predicates make this intuitive interpretation more precise.

In this paper we claim that some of the widely accepted axioms interrelating these control predicates are incompatible with other requirements, arising from the consideration of enabledness and fairness in concurrent and distributed programs.

For any atomic statement or test in the program, the axioms state that eventually the computation will complete this statement once it is begun. Thus no "real" computation is assumed to be immediate (not to take time). However, as part of the abstraction mentioned above, the axioms eliminate explicit reference to any changes in the location counter due, for example, to the linear representation of the concrete, executable image of a program. These manifest themselves as transfers of control (i.e., either conditional or unconditional jumps) generated by the compiler of a structured language. A typical axiom (referred to as ICE, immediate conditional exit) states that

$$after(S_1) \Rightarrow after(if\ B\ then\ S_1\ else\ S_2),$$

(ICE)

thus abstracting away the jump over $S_2$ in a typical linearization of the conditional construct, and assuming that no time is needed (at the program level) to reach the end of the statement. Indeed, such hidden transfers of control are not usually noticeable, as they do not modify any visible state component. As Lamport justly stresses [L1, L2], the correctness of an implementation is always shown under some conventions about the interface between the abstract and concrete level. This interface can be expressed by using what is known as representation mappings. Under the commonly used interface conventions, a mapping that associates the transfer of control to either the representation of the preceding or the following atomic action of the program is satisfactory.
It is the claim of this paper that when fairness is considered, the required interfaces change so as to create an incompatibility with some of the control predicate axioms. With fairness the difference between being "before" a transfer of control or "after" it is visible in concurrent models. This is because fairness involves the enabledness of certain actions, which need not hold before, but may hold after a transfer, or vice versa. Several possible attitudes may be assumed about this incompatibility, and these are discussed in the continuation. The conclusion we reach is that it is desirable to consider being after one statement as a different state from being at the beginning of the statement which follows it according to the semantics of the programming language.

The rest of the paper is organized as follows. In section 2, the main problem is explained in the context of sequential nondeterminism. Though the problem is of less significance in that context, it is easier to state and explain. In section 3, the problem is shifted to the context of fairly communicating processes, where its significance is greater, while section 4 considers the problem in the context of shared-memory concurrency. In section 5 we show that even without a fairness assumption, some other difficulties related to liveness are solved by our suggested solution.

2. Fair nondeterminism and the control axiom for repetition

This section considers nondeterministic repetition in order to clearly present the issues, even though the real significance of the problem arises in fair concurrency, as seen in the following sections. We first consider a common axiomatic formulation of the control behaviour of a deterministic while loop, having the form $S :: \text{while } B \text{ do } S_1$. As seen in [OL, L2], the temporal axioms include one which we call ILB (immediate loopback) stating that $after(S_1) \Rightarrow at(S)$. The effect of this axiom is to identify the states satisfying $after(S_1)$ with a subset of those satisfying $at(S)$, though $at(S)$ may hold under additional circumstances (e.g., when the control first reaches $S$). We shall refer to such a situation as a partial identification of control states.

The natural generalization of this axiom to nondeterministic repetition, formulated for the iterative statement $S :: *[\Box_i B_i \rightarrow S_i]$ of GC (guarded commands) [D] is

$$\bigwedge_{i=1,n} [after(S_i) \Rightarrow at(S)].$$

We focus on this formulation. The $i$-th possibility in such a statement is termed direction $i$, with $B_i$ being its guard, and $S_i$ its body. The usual (informal) meaning of such a nondeterministic repetition is: find an enabled direction $i$, execute its body, and repeat the process until no enabled direction can be found, and then exit. If we consider the predicate enabled as the interface with a resolver of nondeterministic choices in an implementation, we naturally intend that
whenever a direction is enabled, it can be chosen.

The predicate enabled \( i \) is particularly important in the context of a fair nondeterministic repetition (considered here in its strong version \([F]\)), which has to satisfy the additional property

\[
\bigwedge_{i=1,n} \left[ \text{infinitely-often enabled} \left( i \right) \Rightarrow \text{infinitely-often at} \left( S_i \right) \right].
\]

This means that the body of an infinitely-often enabled direction is infinitely often chosen for execution.

One possible definition of enabled in terms of the control of a given program can be expressed as

\[
\text{enabled} \left( i \right) \overset{\text{def.}}{=} \text{at} \left( S \right) \land B_i.
\]

An immediate consequence of \( ILB \) and \( E \) is

\[
\bigwedge_{i=1,n} \bigwedge_{k=1,n} \left[ \text{after} \left( S_i \right) \land B_k \Rightarrow \text{enabled} \left( k \right) \right].
\]

We now consider a typical low-level implementation, taking the form of assembly-language-like commands, each having a memory address. The implementation must have states which allow defining the control predicates in a consistent manner, maintaining axioms such as \( ILB \) and derived conclusions, such as \( EE \). The commands are of two kinds:

1) Operations (like LOAD, ADD, STORE, etc.), which manipulate data (disregarding here the issue of argument addressing, e.g., index-registers).

2) Sequencers (JUMP, JUMPTURE, etc.), which manipulate the transfers of control to impose the required order of execution.

A (low-level) program is a (linear) sequence of such commands. The low-level state has an additional component known as the location counter, denoted here by \( lc \), the values of which are addresses of memory locations. We adhere to the convention that operations are placed contiguously whenever possible, and their semantics includes an augmentation of \( lc \) not explicit in the program. Whenever a violation of contiguousness is needed, sequencers are used. We assume they have the standard effect on \( lc \), e.g., \( \text{JUMP} \alpha \) has the effect of \( lc := \alpha \).

To cope with the resolution of nondeterministic branching (as present in \( GC \)), we stipulate a low-level command \( SCH \ ed - list \). The argument of this command is a (nonempty) list of addresses, and its effect is to assign one of these addresses to \( lc \). A typical use of \( SCH \) is in interfacing an underlying scheduler (hence the name of the command), the list of addresses being the starting addresses of the program sections corresponding to the enabled directions computed by the guard evaluation part. Whether the scheduling is fair or not is reflected in the
control axioms applied.

The implementation of the various GC commands under these conventions is the usual one. An implementation of the nondeterministic iteration is shown in Figure 1, to which we relate in the sequel. The update ed-list section includes the evaluation of the guards and subsequent insertion of all addresses of directions with true guards into ed-list.

We now have to define a mapping of the possible lc values in executions of a low-level program that will express the high-level control predicates and their relationships.

One natural definition of at(S) is straightforward: at(S) should hold iff the value of lc is the address of the first low-level command "belonging" to S. This can be precisely defined by structural induction on S, given the "compiler" generating the low-level implementation. Similarly, one natural definition of after(S) is that it holds iff the previous value of lc is the address of a command "belonging" to S, while the current value of lc is not. Note that the after(S) predicate is not the value of an instantaneous state but also depends on the past. As also seen in [AD], this seems to be a necessary property of definitions for after.

However, this natural pair of definitions violates both the ICE and the ILB axioms. For example, the command JUMP α after the evaluation of S₁ does not belong to the implementation of S₁ and by the above definition after(S₁) has to hold while lc denotes that location. On the other hand, when lc is at JUMP α then at(S) clearly does not hold under the definition above, violating the implication in ILB.

\[
\alpha : [\text{update ed-list}]
\]
\[
\text{JUMP EMPTY ed-list, end}
\]
\[
\text{SCH ed-list}
\]
\[
\alpha_1 : [\text{operations for } S_1]
\]
\[
\text{JUMP } \alpha
\]
\[
\ldots
\]
\[
\alpha_n : [\text{operations for } S_n]
\]
\[
\text{JUMP } \alpha
\]
\[
\text{end: NOP}
\]

Figure 1: the implementation of a nondeterministic iteration
There seem to be two alternative ways of replacing the natural definitions by others that would preserve the axioms. First, we observe that due to nesting of structures, it is not possible to consider just one sequencer as the "source of trouble". Thus if we have a select statement on the right hand side of one of the directions in a repetition, two sequencers will be executed after finishing the code for one of the alternatives and before reaching the beginning of the loop: one to reach the end of the code for the select statement and one to return to the beginning of the repetition. In most syntax driven translators, the number of transfers of control between executions of nonsequencers will depend on the depth of nesting of high-level control structures.

**Definition:** A connecting interval between $S$ and $S'$ (where $S'$ is not a subprogram of $S$) is a finite sequence $\pi$ of sequencers (either conditional or unconditional), each assigning $lc$ the address of the next, so that the last command before $\pi$ belongs to $S$, while the value of $lc$ after $\pi$ belongs to $S'$ and not to $S$.

We now consider possible approaches to the mapping of values of $lc$ while executing commands in a connecting interval. According to the first approach, both $after(S)$ and $at(S')$ hold for all commands in a connecting interval. When applied to the implementation of a loop, this definition preserves ILB. However, due to EE, it has the strange effect that directions become enabled before control reaches the guard evaluation part, and thus without the scheduler being "aware" of this enabling until later.

According to the second approach, $in(S)$ is defined to hold for all $lc$ values in a connecting interval of $S$ to $S'$. This is also rather unnatural, prohibiting compositional definitions. With this definition, $in(x:=e)$ remains true long after the "real" assignment terminates, due to sequencers that were generated by the context in which this assignment is embedded. Since $in(S) \Rightarrow \neg after(S)$ is true in all axiom systems for control predicates, only after the connecting interval has completed execution would $after(S)$ become true. This option has the undesirable property that implications of the form $in(S) \Rightarrow \cdots$ cannot be associated with cutpoints, i.e., syntactic locations in the program, as the same cut point might be in different statements during different executions. Thus for a select statement $S$ containing a right-hand side $S_1$ guarded by $x>0$, it is natural to claim $in(S_1) \Rightarrow x>0$ when $x$ is not changed in $S_1$. However, this would not be true for the above definition of the predicate $in$ which also must be true at the end of the select statement.

A third alternative, in which none of $in(S)$, $after(S)$, or $at(S')$ hold during a connecting interval, requires abandoning ILB and is discussed in the next section.

In the context of fair nondeterminism it does not really matter which interpretation of enabledness is chosen, since "nothing interesting" can happen during the hidden transitions. This is due to the sequential unique thread of control. If the guard $B_j$ is already true at the end of the
body $S_j$, but before the evaluation point of the guards is reached, it will nevertheless remain true until evaluated. Thus the abstracting away of the hidden transition may still be justified.

This situation changes when fair concurrency is considered for a distributed model with communicating processes, as seen in the following section. There the above questions are more difficult to answer, and depending upon the approach taken, may materially influence the requirements for a correct implementation.

3. Fair communicating processes and control predicate axioms

In this section we consider a subset of CSP [H] consisting of programs in a normal form [AC] with fairness assumptions on the execution [KdR, GFK], and investigate its interaction with the immediate loopback axiom seen previously.

A program consists of $m \geq 1$ communicating processes, denoted by $P :: [P_1 \parallel \ldots \parallel P_m]$. Each process $P_i$ either has no loops, or has the form

$$P_i :: Init_i;$$

$$\star \left[ B_j^i; \alpha_j^i \rightarrow S_j^i \right].$$

The processes are disjoint in their state spaces, i.e., have no shared variables. The entire iterative statement of $P_i$ is denoted by $S^i$. A direction now becomes a pair $(i,j)$ of the $j$-th possibility in the iterative statement of process $P_j$. The body of such a direction is $S_j^i$, while the guards consist of $B_j^i$, which is a boolean expression over the local state (and is thus uninfluenced by computations in other processes), and $\alpha_j^i$, which is a communication command (and can also be absent). These may have the form of an output command $P_k ! e$, or an input command $P_k ? x$, for $k \neq i$. The former is interpreted as meaning that the value of $e$ will be sent to $P_k$, while the latter means that a value is to be received from $P_k$ and stored in the variable $x$. The syntactic function $\text{target}$ is defined by

$$\text{target}(P_k ? x) = \text{target}(P_k ! e) = k.$$

Communication is synchronized in that a pair of commands $P_k ! e$ in $P_i$ and $P_i ? x$ in $P_k$ can only be executed jointly. Whichever process arrives first at the point where the guards are evaluated must wait until a matching partner reaches its own guard evaluation. Without fairness, any of the matching pairs with true guards may be selected.

The immediate loopback axiom $ILB$ may now be formulated in this context as:
Analogously to \( E \), the predicate \( \text{enabled}((i,j),(k,l)) \) is given by

\[
\text{enabled}((i,j),(k,l)) = \text{at}(S^i) \wedge \text{at}(S^k) \wedge \text{target}((i,j))=k \wedge \text{target}((k,l))=i \wedge B_1^i \wedge B_1^k
\]  

(\text{JE})

Note that the definition \( \text{JE} \) (jointly enabled) requires the joint presence of the processes involved in the communication part of the guard \( \text{at} \) the corresponding iterative statements. As previously, from \( \text{ILB} \) and \( \text{JE} \), we may conclude:

\[
\bigwedge_{i=1,m} \bigwedge_{j=1,n_i} \bigwedge_{k=1,m} \bigwedge_{l=1,n_k} \left[ \text{after}(S_j^i) \Rightarrow \text{at}(S^i) \right].
\]  

(ILB)

From the above implication, if the directions in process \( i \) become enabled immediately after executing a direction body \( S_j^i \), but before the execution of the local transfer of control, this means that a fair scheduler may enter into action at that stage. Thus if the choice \( (i,j) \) is made as the direction of \( P_i \) which must be chosen next, then the implementation must "freeze" \( P_k \) so that the choice made can indeed be executed. This is a very strong restriction on correct implementations, one not met by the usual direct implementations.

One possibility of adding a fairness requirement is to require:

\[
\text{infinitely–often enabled}((i,j),(k,l)) \Rightarrow \text{infinitely–often} \left[ \text{after}(S_j^i) \wedge \text{at}(S^k) \right].
\]  

(F)

This is known as strong communication fairness [GFK], and is one of the common definitions of fairness for CSP.

Other possible definitions exist, and cause difficulties similar to those to be described below. For example, in the case of weak process fairness it is necessary to first define that a process is enabled if one of its directions is jointly enabled with a direction in another process, and then express that the process will execute if it is continuously enabled. Even in the case of strong communication fairness, a variant is possible in which both \( \text{at}'s \) need not hold simultaneously. The fairness constraint is then relevant for all executions equivalent to one in which the \( \text{at}'s \) hold simultaneously. Such a requirement can be expressed using the recent Interleaving Set Temporal Logic [KP]. We shall not expand on this issue here.

We now turn to the low-level implementation and the mapping of \( lc \) values to the control predicates. Once again we consider an assembly-language implementation. The main difference between \( GC \) and the current situation is that for CSP the handshaking scheduler (whether fair or unfair) is global to the communicating processes \( P_i, i=1,m \). To interface a process with this global scheduler, a local instruction \( SCH ed–list \) is considered, but this time with a more
complicated argument, conveying also the communication requests in the open guards. Upon its completion, the granted communication has taken place, and both \( lc \)'s of the communicating partners are assigned the starting locations of the chosen directions. The linearization is again by means of transfers of control, raising questions similar to the ones in the previous section. However, in this context of fairly-communicating processes, resolving these questions does have significant semantic consequences. In particular, the answers given have a detectable effect on the termination of programs.

As an example of the problems which arise from the difficulty of satisfying \( ILB \), the definition of \( enabled \ J E \), and the above definition of fairness \( F \), consider the program \( S \) seen in Figure 2.

Whether or not this program should terminate depends on the answers to the questions in the previous section. The program has three processes, \( P \), \( Q \), and \( R \). The program will terminate if and only if the communication between process \( R \) and process \( P \) ever occurs. That is, the second guard of \( R \) and the second guard of \( P \) must be jointly available, so that the communication is enabled, and then it must be chosen. If the communication is infinitely-often enabled, by the definition of fairness it must be executed, and thus the program must terminate.

Consider the low-level computation in which the following sequence of actions occurs: (1) \( P \) communicates with \( Q \) using the first guard of each; (2) \( Q \) sends a value back to \( P \), (simultaneously) executing the bodies of the directions chosen; but then (3) \( Q \) continues to evaluate its

\[
S :: [ P \parallel Q \parallel R ], \text{ where:}
\]

\[
P :: x := true ; *[ x ; Q ! true \rightarrow Q ? x ]
\]

\[
\quad x ; R ? x \rightarrow Q ! false ; Q ? x ]
\]

\[
Q :: y := true ; *[ y ; P ? y \rightarrow P ! y ]
\]

\[
\quad y ; R ! y \rightarrow R ? y ]
\]

\[
R :: z := true ; *[ z ; Q ? z \rightarrow Q ! z ]
\]

\[
\quad z ; P ! false \rightarrow z := false ]
\]

Figure 2: Example

Technion - Computer Science Department - Technical Report CS0455 - 1987
guards, in preparation for the next iteration, while \( P \) delays temporarily before the jump back to the head of the iteration statement (where the guards are evaluated); (4) \( R \) evaluates its guards and \( Q \) then communicates with \( R \) using the first guard of each and (5) \( R \) sends a value back to \( Q \), executing the respective bodies as before; now (6) \( Q \) again continues to the guard evaluation stage at the head of the iteration, while \( R \) stops before the hidden jump; (7) \( P \) "wakes up" and evaluates its guards. Now the entire sequence is repeated ad infinitum.

Is this computation fair? It depends precisely on whether we insist on maintaining all of the above definitions, and how we map the implementation level to the program level. If \( \text{after}(S_1^P) \) is true at stage (4), where \( \text{at}(S^R) \) clearly holds, and if \( \text{after}(S_1^R) \) holds at stage (1) where \( \text{at}(S^P) \) clearly holds, then by the axioms we have seen (especially ILB), the sequence is unfair since the communication between \( P \) and \( R \) is infinitely-often enabled, but is not executed. If, on the other hand, \( \text{in}(S_1^P) \) remains true even after stage (3) until stage (7) is reached, and similarly \( \text{in}(S_1^R) \) is true from stage (6) until stage (4) is again reached, then the critical communication is never enabled, and this sequence is thus fair.

It is very likely that the above sequence could occur even in an implementation which is intended to ensure fairness. Thus there is a potential incompatibility. Among the possible approaches to rectifying the problem are:

1. Retain the axioms and mappings so that the above sequence is unfair, but impose heavier demands on correct implementations in order to avoid such sequences. From the implication JEE, if the directions in process \( i \) become enabled immediately after a body \( S_j^i \), but before the execution of the local transfer of control, this means that the fair scheduler may enter into action at that stage. Thus if the choice \((i,j)\) is made as the direction of \( P_i \) which must be chosen next, then the implementation must "freeze" \( P_k \) so that the choice which has been made can indeed be executed. As already mentioned, this is a very strong restriction on correct implementations, and does not seem to be a practical solution.

2. Change the control axioms in order to separate \( \text{after}(S_1) \) from \( \text{at}(S_2) \) where previously those control states had been partially identified. Below we suggest two ways in which this may be done. Both have in common that an \textit{eventuality} is required between visible states even when only sequencers are used to separate them in the implementation. In previous axioms eventualities were used only when considering an atomic operation implemented by at least one operation command, possibly preceded or followed by sequencers. We claim that no harm is done to the logical system by such eventualities across (hidden) sequencers, and that the fair computations so defined have the advantage of requiring less from their correct implementation. This idea can be realized in two ways for our context:
(2a) Replace ILB by its counterpart ELB (eventual loopback) which only guarantees for each $i$ and $j$ that

$$\text{after}(S^i_j) \Rightarrow \text{eventually}(\text{at}(S^i_j))$$

(ELB)

The enabling condition is thus not immediately true when a direction in the body of the loop is finished. This possibility is consistent with the "triggering" instruction SCH of the suggested implementation, which activates the scheduler.

(2b) Retain ILB (and the definition of at a loop) but modify the definition of enabledness so that the condition need only hold at some later state, not immediately when at the loop becomes true. This involves adding an explicit high-level control state after the boolean parts of the guards have been evaluated, when the decision to choose a communication is about to be made. Only then is a subset of the directions considered to be enabled, and the "application" to the scheduler is made. Thus the new definition of enabled would be obtained from JE by substituting after(test$(S^i)$) for at$(S^i)$ and after(test$(S^k)$) for at$(S^k)$. A similar state has been called after(test$(S)$) by Lamport [L2] but he partially identified it with having already made a choice by using the axiom IC (immediate choice)

$$\wedge_i \text{after}(\text{test}(S^i)) \Rightarrow \exists_j \text{at}(S^j_i).$$

(IC)

Instead, we suggest again requiring only an eventuality by using the axiom EC (eventual choice):

$$\wedge_i \text{after}(\text{test}(S^i)) \Rightarrow \text{eventually}(\exists_j \text{at}(S^j_i)).$$

(EC)

Since (as will be described in Section 5) there seem to be other situations in which it is necessary to differentiate between after and at the "next" statement, we prefer the solution in (2a), or, even better, adding both possibilities. The needed liveness properties can still be proven from these "more liberal" control axioms, and safety properties only require a frame axiom using the (strong) Until operator. This axiom must express that properties of the proper state true when after held will remain true until at the next statement, assuming no interference from other processes has occurred.

4. Fair shared-memory concurrency and control predicate axioms

Similar considerations to those in the previous section apply also to a concurrent model with shared memory. For such models control predicate axioms have already been explicitly presented in the literature [OL, L2] and include ILB. We consider, following [OA], a language with concurrent processes sharing variables, where in addition to the standard sequential
composition, branching, and looping, there is also an await statement [OG] of the form
\[ S :: \text{await } B \text{ then } S_1 \]. This means that once such a statement \( S \) in a process \( P \) is reached, control of \( P \) is delayed at that point until \( B \) becomes true, and \( S \) is chosen to execute from among the statements which have been reached in some process and for which the await condition is true.

Then \( S_1 \) will be executed as if it were an atomic statement. Operationally, as soon as \( \text{after} (S) \) holds, all other processes waiting to perform an await statement will compete to execute next.

Again a similar low-level implementation may be considered, with similar problematic interpretations of the control predicates in terms of \( lc \) values.

The relevant fairness notion can be either on the process level, or among all of the potential awaits. The one we have chosen to consider here can be formulated as follows. First, for \( S :: \text{await } B \text{ then } S_1 \) (in \( P_i \)), we define, as usual, \( \text{enabled}(S) = \text{at}(S) \land B \). Then, we have the fairness requirement

\[
\forall S [ \text{infinitely -often enabled}(S) \Rightarrow \text{infinitely -often at}(S_1) ].
\]

The difficulty can manifest itself even with the "innocent" axiom ISC (immediate sequential composition). This is the "obvious" statement that for \( S :: S_1 ; S_2 \), we have

\[
\text{after}(S_1) \Rightarrow \text{at}(S_2).
\]

Suppose that both \( S_1 \) and \( S_2 \) are await statements (with conditions \( B_1, B_2 \), respectively). We would like the resources of the first await to be freed as soon as possible after completion of the body, and thus are tempted to define the control state \( \text{after} (S_1) \) as true even before the condition \( B_2 \) has been evaluated in the implementation of \( S_2 \). This would allow other await statements to "compete" for the choice of being the next await to execute. However, by the definitions, if \( B_2 \) is in fact true when \( \text{after}(S_1) \) holds (even if \( B_2 \) has not yet been evaluated), then the fair scheduler may choose \( S_2 \) as the next await to execute. As a consequence, a correct implementation should prevent any other process from modifying the shared variables, so that \( B_2 \) remains true during the time needed to "pass the semicolon" and actually evaluate \( B_2 \). This again is a very strict requirement, not satisfied by most implementations. As previously, it seems more realistic to merely require \( \text{after} (S_1) \Rightarrow \text{eventually}(\text{at}(S_2)) \), thus separating the states. The additional separation between being at the statement and having evaluated the guard (being after the test, but not having committed to the direction in which to proceed) may also be desirable.
5. Additional benefits

It is interesting to note that in the approach of Pnueli [P], which deals with unstructured flow programs, the problem we have considered does not arise. In his axioms, the concept of "after" is never used, only eventual reachability of at(l) from at(l'), where l and l' are consecutive labels which always have computation between them. However, whenever being after a statement has semantic significance as well as being at the statement, and liveness properties are to be proven, extreme care must be taken in defining control predicates, even when issues of fairness are not directly applicable.

The first example of a difficulty with the coarse identification of concrete locations implied by the usual control axioms is indicated by Apt and Delporte-Gallet [AD]. The problem described there concerns phenomena like

\[ \text{after}(x := 0) \Rightarrow (x = 0) \]

which need not hold (even for sequential programs) if the loop axioms identify after(x := 0) with other locations. Thus, their conclusion also is to have finer identifications of concrete control locations. In their operational semantics, this conclusion is implemented by considering, for example, the fi in if B then S₁ else S₂ fi as causing an explicit transition, and similarly for the else and for a ";". Their semantics also leads to an additional explicit transition between ending the body of a loop and reaching the loop again.

In addition, we claim that our suggestion allows a clean temporal treatment of the skip statement. A desirable property of such a statement is that skip can be inserted arbitrarily (but finitely) often into a program without affecting the essential semantics of the program. However, if ILB and other immediate implications are required to hold, the arbitrary insertion of skip would require the truth of at(skip) ⇒ after(skip), i.e., that skip takes no time. This can be seen most easily by noting that for S₁;S₂ the relation after(S₁) ⇒ at(S₂) is true, and that this should remain true for S₁;skip;S₂. But at(skip) ⇒ after(skip) in turn conflicts with the common view that x := x is equivalent to skip, since the assignment does take time according to all existing axiom systems.

According to a natural definition not identifying control states, the immediate implications are replaced by axioms of the form at(S₁) ⇒ eventually(at(S₂)) and the semantics of skip is defined by at(skip) ⇒ eventually(after(skip)) and a frame axiom. With these new definitions and using the transitivity of eventually, it is clear that insertion of arbitrarily many skip statements will indeed have no effect.
6. Conclusions

We have considered the interaction between the high-level control predicates used in program verification, such as \textit{at}(S), \textit{in}(S) and \textit{after}(S) and the specification of fairness requirements. The main difficulties arise due to the coarse identification of low-level invisible states. In particular, sequences of invisible transfers of control, present in some standard implementations, were considered. The usual axioms imply that such sequences "take no time", sometimes putting a heavy burden on a time-sensitive correct implementation. Joint enabledness is such a time-sensitive concept, revealing difficulties of consistently maintaining the standard fairness definitions and the conventional control axioms. The problems caused by joint enabledness in the context of fairness have already been noted in [GFK] and [BK], but were not related by either to the control axioms. Here we have seen that the "immediate loopback" axiom was seen to impose unacceptable restrictions under some of its possible interpretations. We believe that a finer-grained theory of control predicates solves this and several other problems, and poses no serious new difficulties.

It has been pointed out by L. Lamport (private communication) that some of the problems considered here are related to the liveness aspects of the definition of an \textit{atomic} operation. Our interest, however, has been more in the effective axiomatic definition of high-level language constructs. During this research we were surprised to find that virtually no attention has been paid to the question of the correct implementation of control predicates, and that the axiomatic definitions have been evaluated only for their usefulness in proving liveness properties of programs. It is clearly essential to realize the nature of the restrictions imposed by such axioms on an implementation of the language, and to possibly reject axioms which impose too high a price.

Acknowledgements

The part of the first author was partially supported by the Foundation for research in electronics, computers, and communications, Israeli Academy of Sciences and Humanities. Our attention to this problem was drawn partly due to an example of Luc Bouge (quoted in [AFK]). Michael Rodeh pointed out some interesting consequences of the definition of control predicates for cached architecture implementations.
References


