Theoretical Computer Science 13 (1981) 45-60

THE TEMPORAL SEMANTICS OF CONCURRENT PROGRAMS

Amir PNUELI

Tel-Aviv University, Computer Science Division, Tel-Aviv, Israel

Abstract. The formalism of Temporal Logic is suggested as an appropriate tool for formalizing the semantics of concurrent programs. A simple model of concurrent program is presented in which n processors are executing concurrent n disjoint programs under a shared memory environment. The semantics of such a program specifies the class of state sequences which are admissible as proper execution sequences under the program.

The two main criteria which are required are

(a) Each state is obtained from its predecessor in the sequence by exactly one processor performing an atomic instruction in its process.

(b) Fair Scheduling: no processor which is infinitely often enabled will be indefinitely delayed.

The basic elements of Temporal Logic are introduced in a particular logic framework DX. The usefulness of Temporal Logic notation in describing properties of concurrent programs is demonstrated. A construction is then given for assigning to a program P a temporal formula $W(P)$ which is true on all proper execution sequences of P. In order to prove that a program P possesses a property R, one has only to prove the implications $W(P) \supset R$.

An example of such proof is given. It is then demonstrated that specification of the Temporal character of the program's behavior is absolutely essential for the unambiguous understanding of the meaning of programming constructs.

1. Introduction and overview

The approaches to definition of mathematical semantics of programs can be roughly classified into the following categories:

(a) *Operational*: In this approach we regard programs as generators of execution sequences. Each execution sequence is a sequence of program states. The set of execution sequences associated with a program can be specified by describing an interpreter which generates the sequences, given the program, or by specifying the successor relation which holds between consecutive states in any execution sequence.

(b) *Denotational*: Here we regard a program as a function from the initial state into the final state, or more generally a relation between initial and final states. The semantics is specified by a mapping from programs to the functions or relations they compute.

(c) *Deductive*. Here the stress is not so much on what a program *is* or what it *does*, but on what can be *proved* about its behavior, or about the function or relation it computes. Hoare's axiomatic system, Predicate Transformers, Dynamic Logic and Program Logic all belong to this class.

For sequential deterministic programs, all three approaches have proved very useful and fruitful. Preference for one approach to the other is usually dictated by the specific need one has for a formal semantics. Thus implementors of a language would probably prefer the operational approach, already presenting some kind of an interpreter. The denotational approach is quite useful in resolving delicate and intractable issues in language design such as recursion and parameter transfer mechanisms in an implementation independent way. It is also very beneficial when considering transformations or translation between different languages when both languages have a common semantic range. The deductive approach is of course very attractive to someone interested in verifying the correctness of programs, directly providing him with the needed tools. With the hopeful coming of age of the 'systematic programmer', these tools are increasingly used for the proper construction of programs by a systematic human or machine.

Unfortunately the attempts to extend all these approaches to deal also with nondeterministic, and in particular, parallel programs, are fraught with difficulties. Following is a partial unordered list of some of these difficulties.

(1) In a deterministic program there is only one possible execution path which may either reach the exit point with some final result, or fail or abort in some intermediate state, terminating in an error state. Alternately it may loop forever. In a nondeterministic program there are many possible execution paths each of which may display any of the options listed above. What is then the proper notion of a correct termination? Should we require that at least one path terminates and not care about the others? Or perhaps require all paths to terminate with correct answers? How about all paths either terminating or aborting (sometimes we mean blocking) but none looping? These problems have been partially dealt with by different power domain constructions and a special mechanism in Dynamic Logic.

(2) A parallel program can no longer be considered as a function from initial to terminal states. There are two reasons for that, one syntactic and the other semantic. The syntactic reason is that the modularity inherent in the Denotational Semantics method requires that for every programming construction operator which constructs a new program segment $c(P_1, P_2)$ from two smaller segments P_1, P_2 there exists a semantic operator $\mathscr{C}$ which relates the semantics of P_1 and P_2 to that of $c(P_1, P_2)$. Denoting the semantics of P by $\mathscr{M}(P)$ we need a commutation rule:

$$\mathscr{M}(c(P_1, P_2)) = \mathscr{C}(\mathscr{M}(P_1), \mathscr{M}(P_2)).$$

If we consider $\mathscr{M}(P)$ as the function computed by P, there exists no semantic operator which can relate the function computed by $P_1 \| P_2$ (P_1 run in parallel with P_2) to the functions computed respectively by P_1 and P_2. This is so since when considering P_1 separately we assume its instructions to be executed consecutively while in the execution of $P_1 \| P_2$ the instructions of P_1 are interleaved with those of P_2 creating new effects. The obvious solution is to consider $\mathscr{M}(P)$ no longer as a function into finite states but as a function into the execution sequences generated by P. Then $\mathscr{M}(P_1 \| P_2) = M(\mathscr{M}(P_1), \mathscr{M}(P_2))$, where M is a merging operator. This means that the

execution sequences generated by $P_1 \| P_2$ are all possible merges between execution sequences of P_1 and P_2.

The semantic inadequacy associated with the functional description of programs is that together with parallel programs we naturally consider programs which are models for operating systems. These programs are not run for their final result but rather for maintaining some continuous behavior. Consequently for these programs halting is synonymous with failure and in the non-failing case the notion of a terminal state is meaningless. Again the obvious extension is to consider the complete execution sequence and discuss its properties. On second inspection it seems rather fortunate that we could get away with functions into just the terminal state in the sequential case. We cannot manage with such simple range when considering parallel or even cyclic programs.

(3) The Fair Merge problem. One of the basic assumptions laid down by Dijkstra in his basic model of parallel programs is that the execution of any particular processor might be delayed for any arbitrarily long finite period, (alternately any instruction may take arbitrarily long to terminate) but may not be delayed forever. This worked beautifully in enabling us to separate qualitative from quantitative analysis and analyse properties which are completely independent of any relative rates of speeds between different processors. However, mathematically this assumption is most troublesome in its being discontinuous. Thus in considering the fair merge of two execution sequences S_1 and S_2, we include a set of sequences which have an increasingly long prefix taken from S_1 before any element of S_2 is taken, but exclude their limit which is S_1 alone. Consequently the operator M introduced above is discontinuous. Since continuity is a basic underlying assumption of the Denotational approach it seems questionable whether this operator can be accommodated within the framework of the denotational approach.

In this paper I suggest an approach to the semantics (and verification) of parallel programs which can be described as deductive-operational. It is operational in the sense that the semantic range is that of execution sequences, i.e. sequences of states arising during execution of a (parallel) program. These states include also the location in the program in which the state arises as one of their components. However, to single out only these sequences which are actually realized during a possible computation I use deductive methods. A special logic apparatus called Temporal Logic is used in order to reason about these sequences, and list their properties.

This approach to semantic was primarily motivated by the problem of verification of the properties of parallel programs. In [22] and earlier work, a clear classification is made of properties which one may want to establish for parallel programs according to the complexity of their time dependence. The simplest in this classification are the invariant properties (safety properties in [17]). This class of properties corresponds to the partial correctness notion of sequential programs and covers the important properties of partial correctness, mutual exclusion, deadlock freedom, clean execution and data integrity, for concurrent programs. Several

methods have been proposed for verifying such properties by Keller, Owicki, Ashcroft and Lamport which on closer inspection prove to be very close to one another, and seem highly adequate and reasonably efficient for proving such properties (as much as can be expected under the inherent complexity of the problem). When we consider the class of more time dependent properties, those that relate two events at different instants, we get a class that contains the notions of termination and total correctness for sequential programs, and the concepts of termination for terminating concurrent programs, and those of responsiveness, accessibility, liveness [17] and eventual fairness (in scheduling or responding) for the general concurrent programs. When we get to verifying this class of properties we find that there are very few suggestions, and the only property studied seriously is that of termination. The difficulty stems from the lack of tools for even expressing these properties formally. Temporal Logic provides an excellent and natural tool for expressing these and other properties which depend on development in time. Thus, the temporal semantics of a program is given by a formula $W(P)$ expressing the temporal properties of all its possible and legal execution sequences. Then in order to prove that a temporal property R holds for a program we only have to prove the validity of the implication.

$$W(P) \supset R.$$

This implication is interpreted as stating that any sequence of states, which is a realizable execution sequence of the program P (and hence satisfy $W(P)$) must also satisfy R.

The lack of tools for specifying any but the invariant properties of concurrent and cyclic programs also led to confusion and ambiguity in the introduction of new synchronization primitives. To the extent that some formal definition was given for these primitives it was at best partial. It usually specified under what conditions these primitives may be activated (such as $x > 0$ for the semaphore instruction $p(x)$) and what happens when it is activated (x decremented by 1), but not the frequency at which it must be activated (can it be delayed forever?). We will illustrate in the sequel at least one case in which a wrong implementation of a construct has been 'proven' correct. The problem lies in that the proof only covered the invariant property but not the temporal property which in this case should have been that of (eventual) fairness.

To summarize the benefits of the Temporal Logic approach to semantics and verification:

(a) Temporal Logic enables us to express temporal properties for which no previous formalism existed. Consequently, it

(b) provides us with semantics of programs which takes into account these properties and presents a semantic specification which is complete,

(c) provides a formalism for proving temporal properties of programs based on their temporal semantics.

2. A simple model of concurrent programs

We will present now a simple model of concurrent program which we will study and for which we will present semantics.

A concurrent program consists of n disjoint processes:

$$P = P_1 \| P_2 \| \cdots \| P_n$$

which execute concurrently, plus a set of initial conditions.

Each process can be represented as a single entry transition graph. This is a directed labeled graph whose nodes are labeled by node labels $m_i^0, m_i^1, \ldots$ for process P_i. The edges are labeled by commands of the form $c \to \alpha$, where c is a (guard) condition which may be missing and then interpreted as **true**. α is a statement which may be an assignment of the form $\bar{y} \leftarrow f(\bar{y})$ for the set of program variables $\bar{y} = \{y_1, \ldots, y_p\}$. α may also be empty. We denote the set of labels for process P_i by $L_i = \{m_i^0, m_i^1, \ldots\}$. An example of a concurrent program is given in Fig. 1.

In our model all variables are accessible to all processors. Thus synchronization is accomplished via shared memory. In the graph we preclude self loops, i.e. edges from a node to itself.

A state in our model is a pair $\langle \bar{m}, \bar{\eta} \rangle$, where $\bar{m} = (m_1, m_2, \ldots, m_n)$ is a vector of labels, $m_i \in L_i$ and $\bar{\eta} = (\eta_1, \ldots, \eta_p)$ is a set of values currently assigned to the program variables $y_1, \ldots, y_p$.

An execution sequence for a program is any sequence satisfying the following conditions:

(1) The *initial state* is $\langle (m_1^0, \ldots, m_n^0), (\eta_1^0, \ldots, \eta_p^0) \rangle$, where m_0^i are the entry labels and η_j^0 the initial values of the y's;

(2) *Multiprogramming assumption*: A successive state is obtained from its predecessor by exactly one processor executing one transition which is enabled. Thus let

$$s = \langle (m_1, \ldots, m_n), \bar{\eta} \rangle.$$

If processor i contains an edge from node m_i to node m_i' which is labeled by $c(\bar{y}) \to [\bar{y} \leftarrow f(\bar{y})]$ and $c(\bar{\eta})$ is true, then s' is a possible successor of s, where

$$s' = \langle (m_1, \ldots, m_{i-1}, m_i', m_{i+1}, \ldots, m_n), f(\bar{\eta}) \rangle.$$

Alternately we may allow idling at any stage, i.e. $s' = s$. Note that any command is considered atomic. It is now commonly accepted that if we split the instructions in a program into small enough commands then the multiprogramming model even though simulating concurrency by interleaving is adequate in modelling any desired concurrent situation.

(3) *Fair scheduling assumption*: Let E denote the exit condition of a node m of process i, i.e. the disjunction of all guards on all edges departing from m. In most of the cases this is equivalent to **true**, but this is not the case for example for nodes m_1^2 and m_2^2 in Fig. 1, where $E = (x > 0)$. A sequence is fair if whenever processor i is stuck

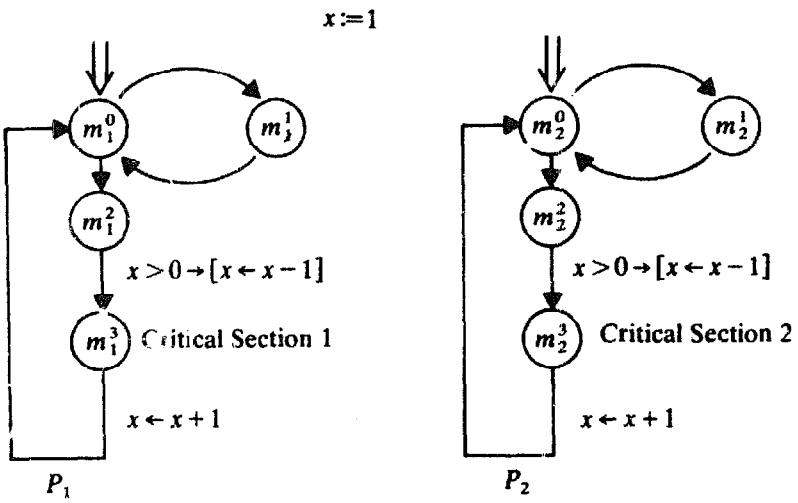


Fig. 1. Mutual exclusion.

at m, i.e. from a certain point on $m_i = m$, then E is true only at a finite number of states thereafter. Stated negatively: no processor whose exit condition is true infinitely often may be deprived forever. Note that we concentrate here on infinitely executing programs. In order to analyze terminating programs we can introduce terminal nodes which have no exits.

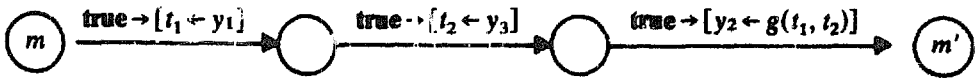
In the frequent case that $E =$ **true** we may replace the above claim by a simpler one: Every processor is eventually scheduled for execution. We only admit fair sequences among our execution sequences.

Consider now the representation of conventional programming elements in our model. Consider any program which may be run in parallel with another and contains assignment statements, tests and unconditional transfers (go to's). The corresponding graph model will contain a node for each statement representing the state just before the execution of this statement. Corresponding to each statement which is the successor of this statement, i.e. may be reached by the execution of the statement, we draw an edge from the statement node to its successor. The label of this edge depends on the statement:

(1) For a test statement of the form '**if** $p(\bar{y})$ l, r' we label the edge from m (being the current node) to l with the label $p(\bar{y})\rightarrow$, and the one to r by $\sim p(\bar{y})\rightarrow$.

(2) For a '**go to** l' statement we draw an edge from m to l which may remain unlabeled or labeled by **true**$\rightarrow$.

(3) For an assignment statement of the form'$\bar{y}_i \leftarrow f(\bar{y})$', let m' denote the node following m in the program. If we want to faithfully model a possible interference between the fetching and storing of operands, we may have to break an assignment statement into a chain of simpler assignments. Thus to fully model $y_2 \leftarrow g(y_1, y_3)$ we need the chain



t_1 and t_2 are auxiliary variables local to the process P_i. Frequently we are assured that no interference may occur and then a single edge labeled by the full assignment will suffice.

For statements of the type 1 – 3 the exit condition of a node (i.e. the disjunction of all outgoing guards) is always **true**. Hence the implication of fair scheduling for such a node is that a processor waiting at such a node will eventually be scheduled resulting in one of the transitions being taken and a new node reached.

Consider now the case that a program contains a synchronization primitive such as $p(x)$, $v(x)$, **with** r **when** B **do** $\cdots$ etc.

(4) For a '$p(x)$' statement, the node from m to m' will be labeled by '$x>0\rightarrow[x\leftarrow x-1)]$'.

(5) A '$v(x)$' is simply represented as '$x\leftarrow x+1$'.

(6) '**with** r **when** B **do**' is represented as

$$r>0\ \&\ B\rightarrow[r\leftarrow r-1]$$

with a corresponding '$r\leftarrow r+1$' at the end of the block.

For statements of this sort the exit condition is not identically **true**. Fair scheduling has to be interpreted as ensuring that if the exit condition is true infinitely often, the processor cannot remain trapped at the node. The crucial observation is that it is not sufficient to require that the processor will eventually be scheduled, because it might always get scheduled when the condition is false and no transition is possible. The stronger condition ensures that it will eventually be scheduled when the exit condition is **true**.

3. Temporal logic or reasoning about sequences

Temporal Logic is a branch of Modal Logic which was designed in order to discuss the variability of situations (or states) over time. It enables us to discuss from within one state the truth of statements either in this state or in states lying in the future or in the past of this state. The full Temporal Logic (as presented say in [23] or [27]), contains operators for referring to both past and future. In our work we found it sufficient to consider only the future fragment. Different Temporal systems exist in order to discuss different models of time such as time measured by integers, branching (non-deterministic) time, etc. In our case we concentrate on integer like time which is deterministic. Note that since we intend to reason within execution sequences, each execution sequence is deterministic (each state having exactly one successor) even though the program generating them is a non-deterministic program and hence many different execution sequences are possible.

We introduce three Temporal operators: X which states truth of properties in the **next** instant, F which states existential truth in the future, and G which states universal truth in the future. Let σ denote the sequence $\sigma = s_0, s_1, \ldots$, then ${}_i\sigma$ is the suffix subsequence ${}_i\sigma = s_i, s_{i+1}, \ldots$ for any $i \geq 0$.

Consider first a well formed Temporal formula constructed from propositions $p_1, \ldots, p_l$, the classical connectives, and the Temporal operators F, G, X. We assume that each state contains truth assignment to all the propositions $p_1, \ldots, p_l$. We may proceed to define the validity of a temporal formula on a sequence $\sigma = s_0, s_1, \ldots$. We will denote $\sigma \mid\approx W$ the fact that the formula W is true on the sequence σ. This is defined inductively as follows:

For a proposition p,

$\sigma \mid\approx p$ iff $s_0 \mid\approx p$, i.e. p is true in the state s_0,

$\sigma \mid\approx W_1 \vee W_2$ iff $\sigma \mid\approx W_1$ or $\sigma \mid\approx W_2$,

$\sigma \mid\approx \sim W$ iff $\sigma \not\mid\approx W$, i.e. it is not the case that $\sigma \mid\approx W$,

$\sigma \mid\approx XW$ iff ${}_1\sigma \mid\approx W$, i.e. W is true for the sequence $s_1, s_2, \ldots$,

$\sigma \mid\approx GW$ iff for every $i \geq 0$ ${}_i\sigma \mid\approx W$,

$\sigma \mid\approx FW$ iff there exists an $i \geq 0$ such that ${}_i\sigma \mid\approx W$.

A formula W is valid if for all sequences σ, $\sigma \mid\approx W$ is true. Thus $\sim Fp \equiv G(\sim p)$ is a valid formula.

The definition of interpretation for sequences can be extended to cover Temporal formulas containing predicates instead of just propositions. It reduces again to the ability to evaluate predicates on states.

The intuitive interpretation derived from the above is that XW is true at a state iff W is true at the *next* immediate state; GW is true at a state iff W is true at *all* future states; and FW is true at a state iff W is true at *some* future state (possibly the present). With this interpretation for the basic operators, we may interpret slightly more complex expressions:

- $p \supset Fq$ – If p is presently true, q will eventually become true,
- $G(p \supset Fq)$ – whenever p becomes true it will eventually be followed by q,
- FGp – at some future instance p will become permanently true,
- $F(p \,\&\, X \sim p)$ – there will be a future instant such that p is true at that instant but false at the next,
- GFp – every future instant is followed by a later one in which p is true, thus p is true infinitely often.

We will illustrate now how some important properties of programs can be expressed as Temporal formulas valid on their execution sequences.

Recall that an execution state is a tuple of the form $s = \langle (m_1, \ldots, m_n), (y_1, \ldots, y_p) \rangle$, $m_i \in L_i$, $y_1, \ldots, y_p$ are program variables.

In our formulas we will use propositions $m_1^1, \ldots, m_n^k$, one for each label in the graph. m_i will be true in s if it appears in the tuple $\bar{m}$, and false otherwise. This double use of m as a label and a proposition should cause no confusion. The proposition m_i being true in s means that s represents a state in which the processor P_i currently executes at m_i. In addition we will use arbitrary predicates over the $\bar{y}$ variables.

We consider first the class of properties which can be expressed as formulas of the form GW where W is classic (i.e. contains no Temporal operators). This is an *invariance* property.

(1) *Partial correctness*: Consider a single sequential program with entry m^0 and exit m^e. Let Ψ be a formula specifying the correctness of the program, i.e. Ψ is to hold on termination. Then partial correctness can be stated as:

$$G(m^e \supset \Psi).$$

This claims that it is invariantly true that whenever we reach the exit point Ψ holds. We can also add the effect of an input restriction φ by writing $m^0 \,\&\, \varphi \supset G(m^e \supset \Psi)$, meaning that if φ is initially true, then the program is partially correct.

(2) *Clean behavior*: For every instruction in the program we can write a condition which will ensure a lawful termination of the instruction. Thus if the instruction contains division this condition will include a claim that the divisor is non-zero (or not too small). If the instruction contains array reference then the condition will claim that the subscript expression is within the array bounds. Let λ_i be the legality condition for the statement departing from m^i. Then a statement assuring peaceful behavior of a program is

$$\left(G \bigwedge_i (m^i \supset \lambda_i)\right).$$

That is: whenever we reach m^i, λ_i holds.

(3) *Mutual Exclusion*: Let each of the processes P_1, P_2 contain a critical section. For simplicity assume that it consists of the single nodes m_1 in P_1 and m_2 in P_2. To claim that these sections are never simultaneously accessed we write

$$G(\sim(m_1 \,\&\, m_2)),$$

i.e. it is never the case that both m_1 and m_2 are true.

(4) *Deadlock Freedom* (Absolute): Deadlock in this context means that all processors are locked and none can move. Obviously in our model a processor can be locked in a node only if its exit condition is not identically true. Let $m_1, \ldots, m_n$ be any set of nodes with exit conditions $E_1, \ldots, E_n$ none of which is identically **true**. Then the statement that deadlock never occurs at $m_1, \ldots, m_n$ is that

$$G(m_1 \,\&\, m_2 \,\&\, \cdots \,\&\, m_n \supset E_1 \vee \cdots \vee E_n),$$

i.e. whenever we simultaneously get to $m_1, \ldots, m_n$, at least one of the exit conditions must be true. In order to exclude deadlock at all possible $(m_1, \ldots, m_n)$ tuples we should take the conjunction of all such candidate combinations. In practice only very few combinations are not identically false anyway.

Next we advance to a class of properties which require a more complicated Temporal structure for their expressions. These are properties expressible by the Temporal implication: $W_1 \supset FW_2$ or more generally $G(W_1 \supset FW_2)$:

(1) *Total Correctness*: Consider again a sequential program with entry m^0 and exit m^e and input-out specification (φ, Ψ). The statement of its total correctness with

respect to (φ, Ψ) is given by:

$$m^0 \,\&\, \varphi \supset F(m^e \,\&\, \Psi),$$

i.e. if currently the program is at m^0 and the input values satisfy φ it is guaranteed to reach m^e, and satisfy Ψ there.

(2) *Accessibility*: In the context of critical sections we often want to prove that any program wishing to enter its critical section will be granted permission to do so. Let m be a location (node) just before the entrance to the critical section expressing the wish of the program to enter its critical section. Let m' be a location inside the critical section. The property of accessibility is then expressible as

$$G(m \supset Fm'),$$

i.e. whenever P is at m it will eventually get to m'.

(3) *Responsiveness*: Suppose that our program models an operating system which receives requests for some resource from many external agents. A request from customer i is signalled by a variable r_i turning **true**. The program allocates the resource between the different customers and signals a granted request by setting a variable g_i to **true**. A reasonable correctness statement for such a situation is that every request is eventually honored:

$$G(r_i \supset Fg_i).$$

Once it has been demonstrated that the Temporal Logic language is a useful tool for expressing and formulating interesting properties of concurrent and cyclic programs (as well as some sequential programs), our next step is to present an axiomatic system in which proofs of these properties can be carried out. Such an axiomatic system called DX is presented below:

4. The system DX

Axioms. Take G and X as primitive operators (F derived as $FW = \sim G(\sim W)$):

$A1$: $G(p \supset q) \supset (Gp \supset Gq)$,
$A2$: $Gp \supset p$,
$A3$: $X(\sim p) \equiv \sim Xp$,
$A4$: $X(p \supset q) \supset (Xp \supset Xq)$,
$A5$: $Gp \supset Xp$,
$A6$: $Gp \supset XGp$,
$A7$: $G(p \supset Xp) \supset (p \supset Gp)$.

Inference rules

$R1$: (*TAU*) If A is an instance of a classical tautology then $\vdash A$.
$R2$: (*MP*) If $\vdash A$ and $\vdash (A \supset B)$ then $\vdash B$.
$R3$: (*GEN*) If $\vdash A$ then $\vdash GA$.

$A1$, $A4$ give distributivity of the logical implication over all future instances and over the next instant. $A3$ specifies the uniqueness of the next instant. $A2$ claims that the present is part of the future (a convention adopted in this system); and $A5$ claims that the next instant is part of the future. $A7$ is the induction axiom. The rule (*GEN*) is based on the assumption that all time instants are symmetric and hence anything *provable* about the present (not just true in the present) is equally provable for any other time instant and hence provable for all future instants.

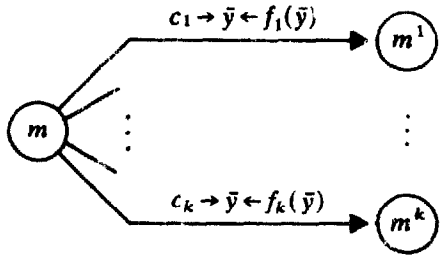
Similar but not identical systems appear in [23]. Other systems, which do not take X as primitive (it can be defined in terms of a stricter F, one which does not include the present) have been more extensively studied under the name D by Dummet and Lemmon [7, 12]. An equivalent system is classified in the general Modal Logic context as S4.3.1 [12].

The proof-theoretic properties of the system DX have been recently studied in [24]. It is shown there that the system DX is deductively complete for linear integer-like time models, i.e., sequences. It is also shown that the propositional formulas in our language are decidable. Below we will use an extension of the language by allowing terms and functions which should be evaluated over states.

5. The temporal semantics of programs

Having at our disposal the Temporal tools we will proceed to formalize the class of execution sequences generated by the concurrent programs of our model.

Consider a node in any of the processes P_i:



We denote the exit condition of m by $E = c_1 \vee c_2 \vee \cdots \vee c_k$. For the node m define first a clause:

$$N_m\colon\ m \wedge \bigvee_{i=1}^{k} [c_i \wedge Xm^i \wedge X\bar{y} = f_i(\bar{y})]. \tag{1}$$

This clause describes the instant of an active transition taken from node m. It states that one of the conditions c_i is true in the current state and that in the next state execution is at m^i and the next $\bar{y}$ is obtained by applying f_i to the current $\bar{y}$. This formulation as $X\bar{y} = f(\bar{y})$ is not strictly in our language but it can be expressed as $[(\bar{a} = \bar{y}) \supset X(\bar{y} = f(\bar{a}))]$. Alternately when the program is a finite state program all variables may be assumed to have only boolean variables. Then f is a boolean

function and we may write $X\bar{y} \equiv f(\bar{y})$ which is within the language.

Define next the claim of fair scheduling for node m.

$$F_m\colon \sim\Big[Gm \wedge GF\Big(\bigvee_{i=1}^{k} c_i\Big)\Big], \tag{2}$$

i.e. it is impossible to remain stuck at m while the exit condition $E = \bigvee_{i=1}^{k} c_i$, enabling at least one of the exits to be taken, becomes true infinitely often. In the frequent cast that $E = \mathbf{true}$ this is equivalent to $\sim Gm$ or $F(\sim m)$, i.e. we can never get stuck at m. Note that if a program contains an exit node, i.e. a node which has no outgoing transitions then $E = \mathbf{false}$ and F_m is identically true for that node, allowing execution of the relevant program to remain at the exit node.

Having defined the basic clauses for each statement we assemble them into statements about complete processes. In that assembly we make use of the following abbreviation: If $w_1, \ldots, w_k$ are formulas, then the statement $\sum_{i=1}^{k} w_i = 1$ claims that exactly one of $w_1, \ldots, w_k$ is true while all the others are false.

Consider process P_j with label set L_j;

$$A_j\colon \sum_{m \in L_j} N_m = 1, \tag{3}$$

expresses the situation that process P_j is active, i.e. some transition in it is taking place,

$$I_j\colon \sum_{m \in L_j} (m \wedge Xm) = 1, \tag{4}$$

expresses the situation that process P_j is idle, i.e. one of the label propositions is true and will remain so in the next instant,

$$B_j\colon \bigwedge_{m \in L_j} F_m, \qquad E_j\colon \sum_{m \in L_j} m, \tag{5}$$

B_j expresses the conjunction of all the fair scheduling requirements for all statements within P_j and E_j expresses the fact that exactly one location proposition is true at any instant.

We may assemble now the statements for each process to a statement for the complete program $P = P_1 \| P_2 \| \cdots \| P_n$. Define first

$$I\colon \bigwedge_{j=1}^{n} I_j \;\&\; \bar{y} = X\bar{y}, \tag{6}$$

this expresses the fact that all processes are idle, and hence the values of all variables remain the same.

Assume that the initial labels in all programs are $\bar{m}^0 = m_1^0, \ldots, m_n^0$, and that the initial values of the variables are $\bar{y} = \bar{\xi}$. Then the formula expressing the semantics of the program is

$$W(P)\colon \bar{m}^0 \;\&\; (\bar{y} = \bar{\xi}) \;\&\; G\Big(\bigwedge_{j=1}^{n} (A_j + I_j = 1) \;\&\; \Big(\Big(\sum_{j=1}^{n} A_j\Big) + I = 1\Big) \;\&\; \bigwedge_{j=1}^{n} (B_j \;\&\; E_j)\Big). \tag{7}$$

The first clause requires the correct initial conditions of the execution sequence. The second contains three subclauses which have to be invariantly maintained. The first states that at any instant each process is either active or idle. The second subclause maintains that at most one process may be active at any time, and if all are idle then the values of all variables stay the same. The third subclause ensures fair scheduling for all the statements in the program, and unique location for each process.

Note first that our semantics allows instants of complete inaction or idling. This is necessary in order to accommodate terminating programs as well as incorrect programs which may inadvertently lead to deadlocks. Even though a program is incorrect it should still have some execution sequences. However, the fair scheduling clause will prevent endless idling while there is still some possible action in one of the processes.

Another important point is the strict interleaving discipline imposed by the second subclause. At most one process may be active at any moment. This is essential since otherwise two $p(x)$ operations may be permitted to occur simultaneously allowing two processes to enter their critical sections at the same time.

The formula expressing the semantics of a program $W(P)$ imposes restrictions on possible sequences which are satisfied only by proper (and fair) execution sequences of the program. Then in order to prove that a property R expressed by a Temporal formula holds we have only to prove the statement

$$W(P) \supset R, \tag{8}$$

i.e. all execution sequences which satisfy $W(P)$ and hence are proper execution sequences of P must also satisfy R.

If indeed (8) is the basic proof principle we should be able to use it to justify all other proof methods suggested for proving properties of concurrent programs, such as [17, 20–22] etc.

Consider for example the simplest and most important proof rule, that of establishing global invariants. It states that if $Q(\bar{\xi})$ is true (i.e. initially true) and Q is inductive, i.e. preserved along each transition in each of the processes, it is invariantly true.

From Q being inductive we infer that for any statement m in any of the processes it is true that $Q \,\&\, N_m \supset XQ$. Thus for every process j, $Q \,\&\, A_j \supset XQ$. Obviously also $Q \,\&\, I \supset XQ$ since no change is taking place. Thus $Q \,\&\, (\sum_{j=1}^{n} A_j + I = 1) \supset XQ$ and hence $W(P) \supset G(Q \supset XQ)$. Since $W(P)$ implies that initially $\bar{y} = \bar{\xi}$ it also implies Q. Thus we have that

$$W(P) \supset Q \,\&\, G(Q \supset XQ)$$

which by $A7$ yields $W(P) \supset GQ$.

In a similar way (8) can be used to argue soundness for all the other proof principles expounded in [14, 17, 22] and any newly suggested ones.

The use of DX in conjunction with (8) for proving accessibility will be illustrated below as an example.

Consider Fig. 1. We wish to prove that once P_1 gets to m_1^2 it will eventually arrive at m_1^3. This represents the fact that whenever one of the processes wishes to access its critical section, this access will eventually be granted. In this case it is sufficient to prove that we never get stuck at m_1^2, i.e. $\sim G(m_1^2)$. Note that this is not an immediate consequence of the fair scheduling policy since the exit condition from m_1^2 is not identically true. The proof proceeds by assuming $G(m_1^2)$ & $W(P)$ and deriving a contradiction to $W(P)$. Hence $W(P) \supset GF(\sim m_1^2)$, i.e. we will always get to a state in which $\sim m_1^2$. The proof below enumerates only the major steps:

1	Gm_1^2	Hypothesis;
2	$G(m_1^3 + m_2^3 + x = 1)$	Can be derived by the invariance rule from $W(P)$. Incidentally this proves the mutual exclusion for m_1^3 and m_2^3;
3	$m_2^3 \supset F(m_2^0 \,\&\, x > 0)$	By the $F_{m_2^3}$ clause of fair scheduling;
4	$m_2^3 \supset F(x > 0)$	Consequence of 3;
5	$\sim m_2^3 \,\&\, \sim m_1^3 \supset (x = 1)$	By 2. Hence $F(x > 0)$;
6	$m_1^2 \supset \sim m_1^3$	by $W(P)$;
7	$\sim m_2^3 \,\&\, m_1^2 \supset (x = 1)$	by 5, 6 hence $F(x > 0)$;
8	$m_1^2 \supset F(x > 0)$	by 4 and 7;
9	$Gm_1^2 \supset [Gm_1^2 \,\&\, GF(x > 0)]$	by *GEN* applied to 8;
10	$Gm_1^2 \,\&\, GF(x > 0)$	by 1, 9 and *MP*.

Step 10 is a contradiction of the fair scheduling clause at m_1^2 hence contradicting $W(P)$. We conclude then that

$$W(P) \supset GF(\sim m_1^2)$$

as required.

In conclusion I would like to illustrate the absolute necessity of having a semantic description which specifies not only the partial correctness properties of constructs (such as our N_m) but also their Temporal Properties (the F_m clause). In a recent report Gries [9] analyzes a proposed implementation of the conditional critical section construct using semaphores. The suggested solution is essentially using semaphores to maintain a queue of all processes which have already expressed a wish to gain access of the resourse using a statement:

with r **when** B_i **do** $\cdots$.

B_i may vary from one process to another. The implementation guarantees that after termination of a critical section by any of the processes all processes currently in the queue are given a chance to test their B_i condition before any outsider is allowed into the queue. In order to prove correctness of the implementation, the following facts are established:

(1) Mutual exclusion is maintained – Only one process is admitted to a critical section.

(2) In the next cycle of testing their conditions, inside processes have higher priority than any outsiders.

(3) Any interested outsider is eventually admitted into the queue.

However, these are not sufficient to guarantee the correct Temporal behavior of the construct which I believe to also include:

(4) It is impossible for a process P_j to be indefinitely delayed while $t > 0$ & B_j is true infinitely often.

Indeed this property is not satisfied by the proposed implementation, which therefore should make it incorrect (if we accept (4) to be an essential property of the **with – when** construct). The failure is due to the fact that a strict queue discipline cannot be maintained using semaphores only. Further Temporal analysis of this and improved algorithms are discussed in [15].

This example was sketchily discussed in order to emphasize the importance of the Temporal conceptual view and tools in both formulating desired properties of programs, and analysing and proving their behavior. It shows that in the absence of proper tools and standards an incorrect implementation can be 'proved' correct. It also strongly urges that semantics of programming (and synchronization) constructs should specify both their invariance properties and also their Temporal properties. Of particular importance is their scheduling or fairness policy.

6. Conclusion

The Temporal approach to programs semantics and verification draws our attention to the richness of the class of properties that one may want to prove about programs and their behavior. It provides us with tools for formulating these properties, and then for formally proving them. The present paper concentrated on expressing the correct behavior of concurrent programs and thus specifying their semantics. This semantic specification can be used to prove other Temporal properties of the programs. It is also of great importance in the specification, study and implementation of new programming constructs and features.

In the present system the level of description is very low. Consequently, proofs of the simplest cases require many minute steps. It is hoped that a systematic experience with proofs in the system will lead to a list of derived meta-rules which will facilitate reasoning at a much higher level.

References

[1] E.A. Ashcroft, Proving assertions about parallel programs, *J. Comput. System Sci.* **10** (1) (1975) 110–135.

[2] E.A. Ashcroft and W.W. Wadge, Intermittent assertion proofs in lucid, in: B. Gilchrist, Ed., *Information Processing* 77 (North-Holland, Amsterdam, 1977) 723–726.

[3] P. Brinch Hansen, A comparison of two synchronizing concepts, *Acta Informat.* **1** (1972) 190–199.

[4] J.R. Büchi, On a decision method in restricted second order arithmetic, *International Congress on Logic Methodology and Philosophy of Science*, Stanford, CA (1960).

[5] R.A. Bull, An algebraic study of Diodorean modal systems, *J. Symbolic Logic* **30** (1965) 58–64.

[6] R.M. Burstall, Formal description of program structure and semantics of first order logic, *Machine Intelligence* 5 (1970) 79–98.
[7] M.A. Dummet and E.J. Lemmon, Modal logic between *S*4 and *S*5, *Z. Math. Logik Grundlagen Math.* 5 (1959) 250–264.
[8] N. Francez and A. Pnueli, The analysis of cyclic programs, *Acta Informat.* 9 (1978) 133–157.
[9] D. Gries, A proof of correctness of Reim's semaphore implementation of the **With – When** statement, Technical Report TR 77-314, Cornell University, Ithaca, N.Y. 14853.
[10] D. Harel and V.R. Pratt, Nondeterminism in logics of programs, *Proc. 5th ACM Symposium on Principles of Programming Languages*, Tucson, AZ (1978).
[11] C.A.R. Hoare, Towards a theory of parallel programming, in: Hoare and Perrot, Eds., *Operating Systems Techniques* (Academic Press, New York, 1972) 61–71.
[12] G.E. Hughes and M.J. Creswell, *An Introduction to Modal Logic* (Methuen, London, 1972).
[13] G. Kahn, The semantics of simple language for parallel programming, in: J.L. Rosenfeld, Ed., *Information Processing* 74 (North-Holland, Amsterdam, 1974) 471–475.
[14] R.M. Keller, Formal verification of parallel programs, *Comm. ACM* 19 (7) (1976) 371–384.
[15] L. Krablin, A temporal analysis of fairness, a forthcoming M.Sc. thesis, University of Pennsylvania.
[16] F. Kröger, LAR: a logic of algorithmic reasoning, *Acta Informat.* 8 (1977) 243–266.
[17] L. Lamport, Proving the correctness of multiprocess programs, *IEEE Trans. Software Engrg.* 3(2) (1977) 125–143.
[18] L. Lamport, Sometime is sometimes 'not never', Technical Report CSL-86, SRI International, Menlo Park, CA (1979).
[19] Z. Manna, Properties of programs and first order predicate calculus, *J. ACM* 16(2) (1969) 244–255.
[20] S. Owicki and D. Gries, An axiomatic proof technique for parallel programs, *Acta Informat.* 5 (1975) 319–339.
[21] S. Owicki and D. Gries, Verifying properties of parallel programs: an axiomatic approach, *Comm. ACM* 19(5) (1976) 279–284.
[22] A. Pnueli, The temporal logic of programs, *Proc. 19th Annual Symposium on Foundations of Computer Science*, Providence RI (1977).
[23] A. Prior, *Past, Present and Future* (Oxford University Press, London, 1967).
[24] D. Gabbay, A. Pnueli, S. Shelah and J. Stavi, The temporal analysis of fairness, *7th Annual Symposium on Principles of Programming Languages*, Las Vegas (1980).