Formal Verification of Parallel Programs

Robert M. Keller
Princeton University

Two formal models for parallel computation are presented: an abstract conceptual model and a parallel-program model. The former model does not distinguish between control and data states. The latter model includes the capability for the representation of an infinite set of control states by allowing there to be arbitrarily many instruction pointers (or processes) executing the program. An induction principle is presented which treats the control and data state sets on the same ground. Through the use of "place variables," it is observed that certain correctness conditions can be expressed without enumeration of the set of all possible control states. Examples are presented in which the induction principle is used to demonstrate proofs of mutual exclusion. It is shown that assertions-oriented proof methods are special cases of the induction principle. A special case of the assertions method, which is called parallel place assertions, is shown to be incomplete. A formalization of "deadlock" is then presented. The concept of a "norm" is introduced, which yields an extension, to the deadlock problem, of Floyd's technique for proving termination. Also discussed is an extension of the program model which allows each process to have its own local variables and permits shared global variables. Correctness of certain forms of implementation is also discussed. An Appendix is included which relates this work to previous work on the satisfiability of certain logical formulas.

Key Words and Phrases: parallel program, correctness, verification, assertions, deadlock, mutual exclusion, Petri net

CR Categories: 4.6, 5.2, 6.9, 8.1

Introduction

The theory underlying the verification of serial programs seems to be fairly well established. In contrast, the corresponding theory for parallel programs has not yet stabilized. A variety of approaches to parallel programs have appeared, ranging from attempts to extend the theory already developed for serial programs (see [1, 2, 7, 10, 19]), to the development of specialized models [11, 14, 15, 16], as well as some informal approaches [3, 6, 9].

Parallel programs are ostensibly more difficult to deal with than serial programs. Obviously, the difficulty in reasoning when simultaneous activity is possible is responsible in large part for this. But the currently open-ended use of the term "parallel program" and the large variety of models for such programs also contribute to the confusion. It is likewise true that there is as yet no widely accepted definition of "correctness" for parallel programs.

The models dealt with here are formal, but the proofs will be informal. Formal models are useful in that they allow a delineation between semantics, the property being proved, and the actual proof itself. Anyone is free to disagree that the formal property really expresses correctness, to formulate an alternative, and to construct a proof of it. Similarly, if the model is inadequate, then one is free to modify it. The situation is sometimes preferable to a completely informal proof which is subject to question because of vagueness in the model.

We will deal with two models. The "conceptual model" is highly abstract. In fact, it is abstract enough to include almost every existing parallel program model as a special case. It is in this model that we feel concepts of correctness are best formulated. In this way, we can reason about entire classes of models, free of encumberances of specific representations. The conceptual model is also useful in comparing different representations with respect to their relative ease in characterizing certain phenomena and in formulating and proving properties. Other results which demonstrate the advantages of such a model are presented in [17].

Within the conceptual model, we formulate an in-

Copyright © 1976, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

Work reported herein was sponsored by the National Science Foundation through Grant GJ 42627.

Author's address: Department of Electrical Engineering and Computer Science, Princeton University, Princeton, N.J. 08540.

duction principle which seems to be a key principle for proving correctness. The formulation at this level of abstraction is new, but the direction has been pointed by the work of others, notably [6, 7].

A second model provides a means of presenting parallel programs. Although the model is intended to illustrate the application of the conceptual model, it is in itself quite general. It appears to be fairly unique in demonstrating a facility for proving correctness of systems in which an arbitrary number of processes are executing the same reentrant program. This model, however, is not being proposed as the final model, and possible extensions of it will be discussed in the concluding section.

1. The Conceptual Model

The conceptual model we present appears to be general enough to capture most notions of parallel computation which have been proposed. Its usefulness is not so much as a practical tool, but, as the name indicates, as a way to conceive of the operation of a system for parallel computation and of the properties of such a system. Formally, we call this model a "transition system."

Definition 1.1. A *transition system* is a pair $(Q, \rightarrow)$, where Q is a set of *states* and $\rightarrow$ is a binary relation on Q, called the set of *transitions*.

We do not restrict either Q or $\rightarrow$ to be finite. When q and q' are states, we think of $q \rightarrow q'$ as meaning that there can be an *indivisible* progression from state q to state q'. In a system which allows "parallelism," for a given state q there may well be many states q' such that $q \rightarrow q'$. That is, $\rightarrow$ is not necessarily a function. Each such state represents a result of one of a number of possible *actions*, each of which is *simultaneously enabled* when the system is in state q.

We have not yet said what, in reality, is happening when the action corresponding to a transition occurs. It could be that the action represents a single *event*, or perhaps several events occurring simultaneously. However, it is usually assumed that an action does represent only a single event, and if there is any possibility of simultaneous events occurring, such an occurrence can be *represented* as a sequence of occurrences of events in some arbitrary order. This assumption will be made in what follows, and can be referred to as the *arbitration condition*. It is also the case that "events" which last over a period of time, rather than occurring instantaneously, can frequently be modeled by the instantaneous type of event by a suitable refinement of the meaning of the former. For example, at the level of memory accesses in a multicomputer system, it is usually the case that two accesses to the same memory location do not overlap in time because of a hardware arbitration device. Thus, although the machinery required to validate this assumption is nontrivial in a truly asynchronous system, we will not further discuss this problem, nor the validity of our methods under weaker assumptions.

The events which correspond to transitions may be represented by assigning *names* to the transitions. More than one transition may have the same name, and hence represent the same event. The manner in which names are assigned can be of importance in describing the system and its properties, as will be seen.

Another use of names is in the *presentation* of a system. For example, suppose that the set of states is the set of all pairs of natural numbers, $\omega \times \omega$. We wish to say that for each two states (x, y) and (x', y'), there is a transition $(x, y) \rightarrow (x', y')$ provided that $x \geq 1$, $x' = x - 1$, and $y' = x + y$. As the set of such transitions is infinite, we cannot hope to list them all. But we can present them all at once by the name

$$t\text{: } \textbf{when } x \geq 1 \textbf{ do } (x', y') = (x - 1, x + y) \qquad (1.1)$$

If the expression above were considered to be a statement of a program, then this naming would be quite natural, as the change in state from (x, y) to (x', y') corresponds to the *execution* of the statement.

It is appropriate to extend our definition of transition system as follows.

Definition 1.2. A *named transition system* is a triple $(Q, \rightarrow, \Sigma)$ where $(Q, \rightarrow)$ is a transition system and each transition is assigned one or more names in the set Σ.

According to the preceding definition, a named transition system can be *presented*, in the case where Σ is finite (or, more generally, recursive), by presenting a binary relation on Q, $\rightarrow^t$, for each t in Σ. In other words, $q \rightarrow^t q'$ means that there is a transition from q to q' with name t, and $\rightarrow$ is just $\bigcup \{\rightarrow^t \mid t \in \Sigma\}$. By the arbitration condition, we may assume that the transitions so named represent atomic, indivisible actions.[1]

In many cases of interest, each $\rightarrow^t$ is a *partial function*. That is, for any state q, there is at most one (but perhaps no) state q' such that $q \rightarrow^t q'$. In this case, we say that the named transition system is *deterministic*. If there is such a q', then we say that t is *enabled* in state q. Any such partial function may also be represented by a pair (P_t, F_t), where P_t is a (unary) predicate on Q and F_t is a partial function such that $F_t(q)$ is defined whenever $P_t(q)$ is true. That is,

$$q \rightarrow^t q'$$

iff $P_t(q)$ and $q' = F_t(q)$. P_t is appropriately called an *enabling predicate* and F_t an *action function*. In example (1.1) above, the enabling predicate is $P(x, y)$: $x \geq 1$, and the action function is $F(x, y) = (x - 1, x + y)$. More concrete examples of enabling predicates and action functions will be demonstrated in the next section.

Although it is rarely essential, all subsequent discussion in this paper will deal with the deterministic

[1] Transition names may also be used in providing a trace of the program's execution, but we do not use them as such here.

case, unless the contrary is stated explicitly. Note that "nondeterministic" has been used differently in some of the literature, e.g. [1]. Typically, this term denotes a system which is "polygenic." By *monogenic*, we mean a system such that for all states q, there is at most one q' such that $q \rightarrow q'$, and by *polygenic* we mean the absence of this condition.

In dealing with programs, it is customary to represent the state set as a set product, $Q = \Gamma \times \Xi$, where Γ is a set of *control states* and Ξ is a set of *data states*. The presentation of states in this manner provides a convenient means of decomposing the enabling predicates and action functions. In many formal models, Γ is a finite set. An example of a model in which Γ can be infinite is presented in the next section.

2. A Model for Presenting Parallel Programs

The previous section discussed the "conceptual model," which views a system as an abstract binary relation on a set of states. As a vehicle for discussing correctnes of parallel programs, we will discuss a presentational model, which, for sake of reference, will simply be called a *parallel program*. As will be seen, this model is quite general in itself.

We choose, as our representation of parallel programs, a bipartite directed graph, the nodes of which are divided into

(i) *place nodes*: representing points at which an instruction pointer of a processor may dwell,
(ii) *transitions nodes*: representing a class of transitions, each denoting an event which corresponds to the execution of a particular instruction.

The rule of formation for these graphs is that there are no arcs directed from a place node to another place node, or from a transition node to another transition node. Other than this restriction, there are no "illegal" programs, as there are in [1, 2]. When there is an arc directed from one node to another, we say that the former is an *input node* for the latter, while the latter is an *output node* for the former.

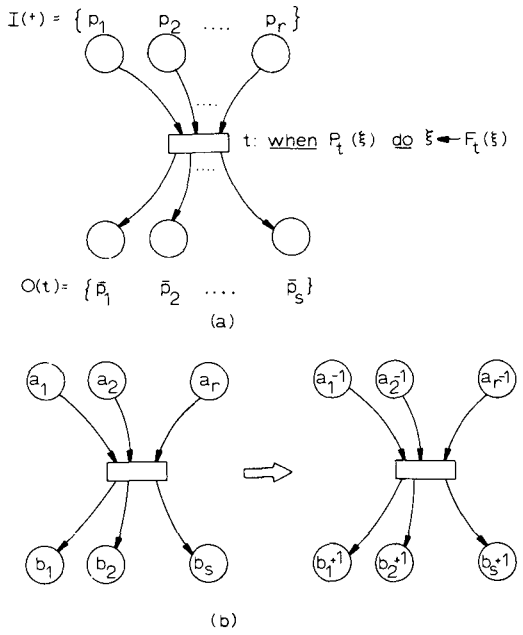
In addition, a parallel program entails a vector ξ of *program variables*. Each component of ξ is a variable, say x, which is assigned a value from a specific domain, D_x. Finally, each transition node t has attached to it an expression of the form

$$\textbf{when } P_t(\xi) \textbf{ do } \xi \leftarrow F_t(\xi) \tag{2.1}$$

Here P_t is a unary predicate, and F_t a function, on vectors of program variables.

Both place nodes and transition nodes may be assigned labels for the sake of reference. Furthermore, we consider the labels of place nodes $p_1, p_2, \ldots$ also to be indices for a set of variables $v(p_1), v(p_2), \ldots$ distinct from all program variables, which indicate the number of instruction pointers dwelling at each place.

Fig. 1 (a) representation of transition node; (b) action of transition on place variables (assuming $I(t) \cap O(t) = \emptyset$).



We therefore call these variables *place variables*. Of course, the only values a place variable may take are natural numbers. For simplicity, we may in the sequel let $v(p_i)$ be abbreviated v_i.

To express the semantics of a parallel program, we use a named transition system, where the *state* is a vector $(\gamma; \xi)$, with γ the vector of place variables and ξ the vector of program variables. Thus γ may be thought of as the "control state," and ξ the "data state," as was discussed in the previous section. Suppose that t is a transition node, labeled with an expression of the form (2.1), and that the set of input place nodes for t is $I(t)$, while the set of output place nodes for t is $O(t)$. Then there is a transition

$$(\gamma; \xi) \rightarrow^t (\gamma; \xi')$$

if, and only if

(i) for each $p \in I(t)$, $v(p) \geq 1$,
(ii) for each $p \in I(t) - O(t)$, $v'(p) = v(p) - 1$,
(iii) for each $p \in O(t) - I(t)$, $v'(p) = v(p) + 1$,
(iv) for each $p \in O(t) \cap I(t)$, $v'(p) = v(p)$,
(v) $P_t(\xi)$ is true,
(vi) $\xi' = F_t(\xi)$.

Such a transition is illustrated in Figure 1.

The reader may have observed that the action of a transition node on the place variables is similar to the action of a transition in a "Petri net," or some other similar model [11, 16]. Of course, such models are more restrictive, in that they do not completely model the action of the transition node upon program variables. The use of place variables to represent multiple instruction pointers has not, to the author's knowledge, been observed before.

Fig. 2. Representation of (a) *fork*, (b) *join*.

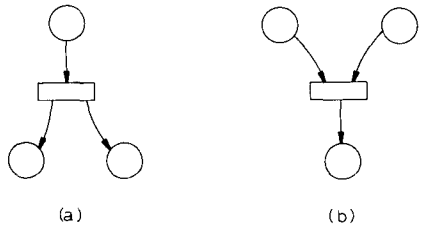
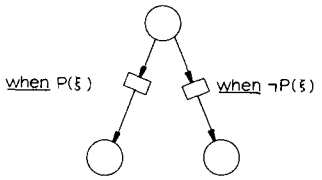


Fig. 3. Representation of a decision.



We adopt the *notational convention* that, in an expression labeling a transition node,

(i) "**when** $P_t(\xi)$" may be omitted, in which case $P_t(\xi)$ is taken to be identically true,

(ii) "**do** $\xi \leftarrow F_t(\xi)$" may be omitted, in which case ξ does not change when the transition occurs.

For example, the *fork* and *join* constructs [4] may be represented by transition nodes as shown in Figure 2. The standard "decision box" of flowcharts which tests $P(\xi)$ is represented by the two transition nodes shown in Figure 3. A *semaphore* [6] may be represented by the introduction of a program variable, say S, whose value is a nonnegative integer, with the P and V operations represented as shown in Figure 4.

Note that this representation allows a uniform treatment of decisions and "synchronizing operations." That is, a synchronizing operation is a transition node whose enabling predicate may prevent an instruction pointer from passing through, even though there is no alternative transition node through which it can pass, whereas a decision always allows an alternative. The model also allows more than one alternative to be possible for a given state, i.e. it is generally polygenic.

The reader should also note that, in Sections 2, 3, and 4, there is no *formal* significance attached to the use of place variables for representing instruction pointers. A place variable could just as well represent a program variable whose value is restricted to the natural numbers and which can only be modified in certain restricted ways. For example, a semaphore can be viewed as a place variable and the operations upon it represented by transitions, as depicted in Figure 5. The validity of this representation follows from the rules regarding the enabling condition for a transition node. The techniques to be described work equally well in either representation of semaphores.

The locus of an instruction pointer during program execution is called a *process*. We should point out that our representation is a good model, provided that we can make the abstraction that there is no need to distinguish the identity of the individual processes. In Section 6 we consider an extension which preserves the identity of processes, at the expense of greater complexity.

Finally, we may include in our definition of parallel program an *initial state* which assigns values to all place variables and program variables. We indicate the intended initial value of place variables by a number within the place. Empty places have initial value 0.

To fix our terminology, a *serial program* is one in which for every state there is only one nonzero place variable, indicating the position of the unique instruction pointer. Hence monogenic serial programs are ordinary programs and polygenic serial programs are what are sometimes called *nondeterministic serial programs.*

3. The Induction Principle

Let us return, for the moment, to the conceptual model of Section 1. Let $(Q, \rightarrow)$ be a transition system. We define the relation $\rightarrow^*$ to be the reflexive, transitive closure of $\rightarrow$. That is, $q \rightarrow^* q'$ iff there is a sequence of zero or more states such that each is related to its successor by $\rightarrow$, which starts with q and ends with q'. In this case we say q' is *reachable from* q, or simply "reachable" if q is understood. A more precise, inductive, definition of $\rightarrow^*$ is of interest.

Definition 3.1. Let $(Q, \rightarrow)$ be a transition system. We define the binary relation $\rightarrow^*$ on Q as follows:

(i) $(\forall\, q \in Q)\ q \rightarrow^* q$

(ii) $(\forall\, q, q', q'' \in Q)$ if $q \rightarrow^* q''$ and $q'' \rightarrow q'$, then $q \rightarrow^* q'$

(iii) The only elements of $\rightarrow^*$ are those which are derivable by a finite number of applications of (i) and (ii) above.

Given a transition system $(Q, \rightarrow)$, we are generally interested in the behavior of the system when started in certain initial states. It is, of course, unlikely that we are interested in considering all elements of Q as possible initial states. When a system is started in state q_0, the only interesting states are those q such that $q_0 \rightarrow^* q$. Thus we have the following definition.

Definition 3.2. Let $(Q, \rightarrow)$ be a transition system and $q_0 \in Q$. A unary predicate J on Q is said to be q_0-*invariant* if for each q such that $q_0 \rightarrow^* q$, $J(q)$ is true.

The term "invariant" has been used informally by others to denote an assertion about program variables which is preserved by the execution of a "loop" or a critical section, or by certain synchronizing primitives [3, 9, 10]. Here we are interested in a more general notion of an invariant, which involves the entire state, not just the program variables. An informal expression of the concept of an invariant, which comes closer to what we have formally defined, appears in work by Dijkstra [6].

Fig. 4. Representation of a semaphore: (a) *P*-operation, (b) *V*-operation.

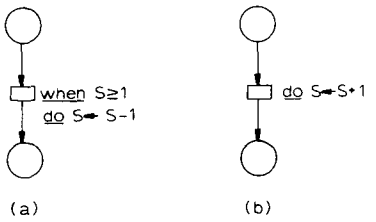


Fig. 5. Using a place variable S to represent a semaphore: (a) *P*-operation; (b) *V*-operation.

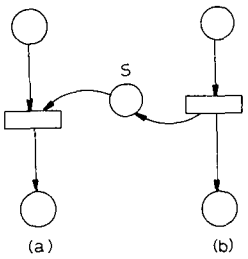
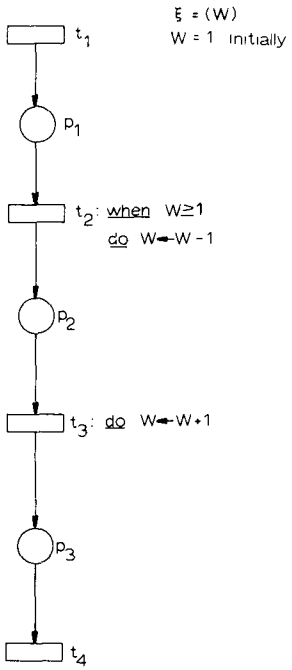


Fig. 6. A parallel program representing mutual exclusion.



In order to discuss the proof of an invariant, we introduce the following concept.

Definition 3.3. Let $(Q, \rightarrow)$ be a transition system, and $q_0 \in Q$. A unary predicate J on Q is said to be *q_0-inductive* provided that

(i) $J(q_0)$ and

(ii) $(\forall q, q' \in Q)$ [$J(q)$ and $q \rightarrow q'$] implies $J(q')$.

A key technique for proving a predicate invariant is the *induction principle* stated next.

PROPOSITION 3.1. *For any transition system $(Q \rightarrow)$, with $q_0 \in Q$, if J is q_0-inductive then J is q_0-invariant.*

PROOF. The principle follows directly enough from the definition of $\rightarrow^*$ so as not to require proof. However, the reader desiring reduction to more traditional numerical induction may carry out the proof using the number of transitions used to get from q_0 to q as an induction variable.

The power which lies in the induction principle is that it does not require a complete characterization of reachability in order to demonstrate that a predicate is invariant. In terms of a presentational model, such as the one in Section 2, the proof of (ii) amounts to showing, for each transition name t, that

$$[(q \rightarrow^t q') \wedge J(q)] \Rightarrow J(q')$$

where q and q' are *arbitrary* vectors of place variables and program variables.

All of the examples to be discussed will be with respect to a single initial state. We could extend the development to consider all initial states satisfying some predicate Φ, which would lead to the concept of Φ-invariance, Φ-induction, etc. However, for simplicity we will avoid doing so.

We now demonstrate the formal induction principle on a simple example. A similar example for a fixed number of instruction pointers appeared in [6].

Suppose that each of some arbitrary set of parallel processes has a section of code, called a *critical section*, which for some reason must be executed by at most one process at a time. In order to achieve this *mutual exclusion* of critical section executions, we provide a semaphore W, initialized to 1, upon which each process executes the *P*-operation before entering the critical section and the *V*-operation after leaving the critical section. This example is illustrated in Figure 6, using the model of Section 2.

Transition node t_1, as it has no place nodes as input nor any enabling predicate, is always enabled, and can at any time act to introduce a new process at p_1. Similarly, t_4, although not required, annihilates these processes arbitrarily. A pointer at p_2 represents a process in its critical section, so we wish to show that

$$v_2 \leq 1 \tag{3.1}$$

is invariant with respect to the initial state $(v_1, v_2, v_3, w)_0 = (0, 0, 0, 1)$, recalling the abbreviation v_i for $v(p_i)$.

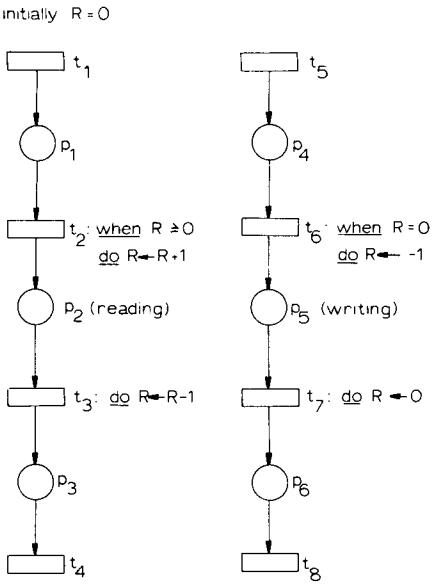
Unfortunately, (3.1) is not q_0-inductive. As is sometimes the case with inductive proofs, this predicate provides an inductive hypothesis which is too weak. We must strengthen this predicate to

$$(v_2 \leq 1) \wedge (v_2 \cdot W = 0) \wedge (W \leq 1). \tag{3.2}$$

Of course, (3.2) implies (3.1).

The strengthening from (3.1) to (3.2) may be viewed as our taking into account the fact that the value of

Fig. 7. A parallel program representing readers and writers.



semaphore W reflects some information about the number of instruction pointers at v_2, which was the purpose of introducing W in the first place.

Letting $J(v_1, v_2, v_3, W)$ denote (3.2), $J(0, 0, 0, 1)$ clearly holds and establishes the basis of the proof. Now let us assume that $J(v_1, v_2, v_3, W)$ holds and $(v_1, v_2, v_3, W) \rightarrow (v_1', v_2', v_3', W')$, and try to establish $J(v_1', v_2', v_3', W')$.

The proof is greatly simplified by observing that, since v_2 and W are natural numbers, (3.2) becomes equivalent to

$$v_2 + W \leq 1$$

As any t_i which increments v_2 also decrements W by the same amount, and vice versa, the induction step is easily established. For example, $(v_1, v_2, v_3, W) \rightarrow^{t_2} (v_1', v_2', v_3', W')$ implies that $v_2' = v_2 + 1$, and $W' = W - 1$. Thus $v_2' + W' = (v_2 + 1) + (W - 1) = v_2 + W$.

As another example, consider the version of the "reader/writer" problem which follows. This problem deals with two classes of critical sections: *read-sections* and *write-sections*. The rules are that at most one process may be in a write-section at a time, but any number may be in read-sections as long as there are none in write-sections concurrently.

A sketch of a possible solution to this problem, using a variable R for "synchronization," is shown in Figure 7. This time we wish to show that

$$v_2 \cdot v_5 = 0 \wedge v_5 \leq 1 \tag{3.3}$$

is invariant with respect to the initial state (0, 0, 0, 0, 0, 0, 0). That is, $v_2 \cdot v_5 = 0$ represents the fact that there is no simultaneous execution of read- and write-sections, and $v_5 \leq 1$ represents the execution of at most one write-section at a time.

As before, the desired invariant is too weak to be inductive, and we must prove the stronger condition, $J_1(v_1, v_2, v_3, v_4, v_5, v_6, R)$, defined to the conjunction of

(i) $v_2 \cdot v_5 = 0$,
(ii) $v_5 \leq 1$,
(iii) $R < 0 \Leftrightarrow v_5 = 1$,
(iv) $(v_2 > 0 \vee R \geq 0) \Rightarrow v_2 = R$.

Clearly $J_1(0,0,0,0,0,0,0)$ is true. Let (i)′ − (iv)′ denote (i)–(iv) with each variable primed. We must show for each $t \in \{t_1, \ldots, t_8\}$ if (i)–(iv) hold and

$$(v_1, v_2, v_3, v_4, v_5, v_6, R) \rightarrow^t (v_1', v_2', v_3', v_4', v_5', v_6', R')$$

then (i)′–(iv)′ also hold.

Since t_1, t_4, t_5, and t_8 do not involve any of the variables in (i)–(iv), we need not consider these transitions further. We consider the remaining transitions in turn.

t_2 : $R \geq 0$, $R' = R + 1$, $v_2' = v_2 + 1$.
By (ii). (iii), $v_5 = 0$, thus $v_5' = 0$, giving (i)′, (ii)′, (iii)′.
By (iv), $v_2 = R$, so $v_2' = R'$, giving (iv)′.

t_3 : $v_2 > 0$, $v_2' = v_2 - 1$, $R' = R - 1$.
By (i), $v_5 = 0$, giving (i)′. By (ii), (ii)′ follows.
By (iv), $R > 0$ so $R' \geq 0$, giving (iii)′.
By (iv), $v_2' = R'$, giving (iv)′.

t_6 : $R = 0$, $R' = R - 1$, $v_5' = v_5 + 1$.
By (ii), (iii), $v_5 = 0$, so $R' = -1$ and $v_5' = 1$, giving (ii)′, (iii)′.
By (iv), $v_2 = 0$, giving (i)′, (iv)′.

t_7 : $v_5 > 0$, $v_5' = v_5 - 1$, $R' = 0$.
By (i), $v_2 = 0$, so (i)′ and (iv)′.
By (ii), $v_5 = 1$, so $v_5' = 0$, giving (ii)′, (iii)′.

We will return to this example in a later section.

A more substantial example is provided in Figure 8, where we illustrate a parallel program which implements the reader/writer type of synchronization using semaphores, in the manner of Courtois et al. [5]. In this case, the desired invariant is expressed by

(i) $v_{16} \leq 1$,
and
(ii) $v_{16} \cdot v_7 = 0$.

In constructing a proof of these invariants, the author successively strengthened the predicate by conjoining the predicates in the following list.

(iii) $v_{16} \cdot w = 0$,
(iv) $r = v_6 + v_7 + v_8$,
(v) $v_{16} \cdot r = 0$,
(vi) $r \cdot w = 0$,
(vii) $w \leq 1$,
(viii) $v_3 \cdot w = 0$,
(ix) $v_{11} > 0 \Rightarrow r = 0$,
(x) $v_{11} \cdot w = 0$,
(xi) $(v_3 + v_6 + v_7 + v_8 + v_9 + v_{11}) \cdot w = 0$,
(xii) mutex $+ v_2 + v_3 + v_4 + v_6 + v_8 + v_9 + v_{10} + v_{11} = 1$.

The predicates in this list were discovered as follows. In

order to carry out the induction step, it suffices to show that for each i, $[J(q) \wedge q \rightarrow^{t_i} q'] \Rightarrow J(q')$. We may proceed by assuming $J(q) \wedge \neg J(q')$ and then deriving a contradiction for each i such that $q \rightarrow^{t_i} q'$. The form of J can be observed to eliminate most of the t_i from consideration immediately. If unable to carry out the proof, then, aided by intuition, we add enough predicates to the list to make the proof go through. We do not have to discard the work done so far, because the new addition simply strengthens the predicate.

We illustrate with the first two steps:

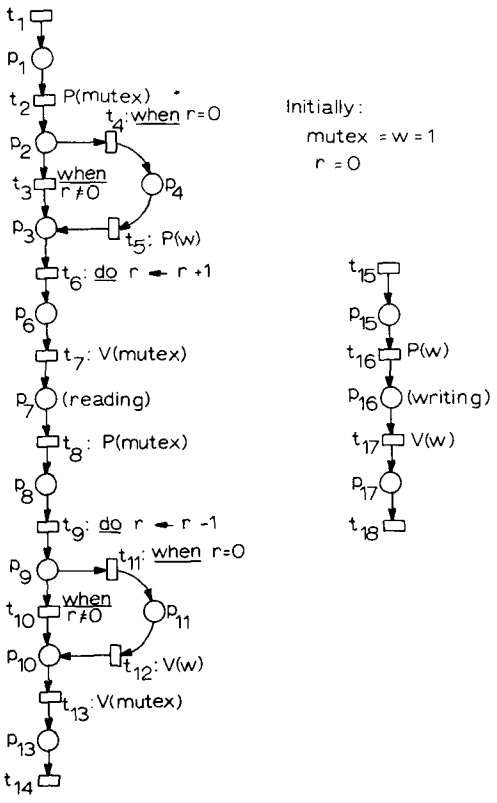
(i) Assume $v_{16}' > 1$, but $v_{16} \leq 1$. Then $v_{16} = 1$, and we must have $i = 16$. Hence $w > 0$. By introducing (iii) we get a contradiction.

(ii) Assume $v_{16}' \cdot v_7' = 0$. Since no transition affects both v_{16} and v_7, either $v_{16} > 0$ and $i = 7$, or $v_7 > 0$ and $i = 16$. In the former case, $v_{16} \cdot v_7 > 0$, implying that $v_{16} \cdot r > 0$ by (iv), but this contradicts (v). This prompted the introduction of (iv), (v). In the latter case, $v_7 \cdot w > 0$, implying, by (iv), that $r \cdot w > 0$. However, by introducing (vi), we get a contradiction.

Of course, we must be careful when using this approach that each predicate introduced is actually invariant. If a noninvariant predicate is introduced, we will be unable to complete the induction.

A further refinement of Theorem 3.1, which relates our work to [1] is discussed in the Appendix.

Fig. 8. An implementation of reader/writer control using semaphores.



4. Place Assertions

As mentioned in Section 1, in many models the state set can be written as $Q = \Gamma \times \Xi$, where Γ is finite. If we write $\Gamma = \{\gamma_0, \gamma_1, \ldots, \gamma_H\}$, then any unary predicate J on Q can be expressed as a finite conjunction, namely

$$J(\gamma; \xi): \bigwedge_{j=0}^{H} [\gamma = \gamma_j \Rightarrow A_j(\xi)] \tag{4.1}$$

where $A_0, A_1, \ldots, A_H$ are unary predicates on Ξ. It is customary to call the predicates A_j *assertions*. A well-known special case of our induction principle for serial programs is the "assertions method" or "Floyd's method" [7]. For each pair of control states (γ, γ'), let

$$T(\gamma, \gamma') = \{t \mid (\exists \xi, \xi')\ (\gamma, \xi) \rightarrow^t (\gamma', \xi')\}.$$

Then the induction step involved in the proof that J of (4.1) is inductive is equivalent to proving

$$\bigwedge_{\gamma,\gamma'} \bigwedge_{t \in T(\gamma,\gamma')} [A_\gamma(\xi) \Rightarrow [[P_t(\xi) \wedge \xi' = F_t(\xi)] \Rightarrow A_{\gamma'}(\xi)]]. \tag{4.2}$$

(4.2) will be recognized as the conjunction of *verification conditions* in the standard nomenclature of Floyd's method for serial programs. In this case, the control states correspond to places and there is precisely one pair (γ, γ') for each transition name t.

Assume that γ_H is a unique "halting" control state. The assertions method may then be used to prove "partial correctness" with respect to a predicate ψ, which means that the predicate

$$J(\gamma; \xi) : (\gamma = \gamma_H) \Rightarrow \psi(\xi) \tag{4.3}$$

is $(\gamma_0; \xi_0)$-invariant. It is generally necessary to prove the stronger predicate (4.1) to be $(\gamma_0 : \xi_0)$-inductive in order to conclude that (4.3) is $(\gamma_0; \xi_0)$-invariant.

Essentially the same representation can be used for parallel programs when the set of control states is finite. This approach is sometimes termed "converting to a nondeterministic serial program," and is the essence of [1, 19]. By identifying places of a second parallel program with the control states (i.e. multisets of places) of the original program, the equivalent polygenic serial program is displayed explicitly. Figures 9 and 10 demonstrate this equivalence.

Unfortunately, the representation of a parallel program as a polygenic serial program is not always useful. First of all, it is inconvenient if the number of control states happens to be large. Second, we are required to enumerate the control states and construct an assertion for each. Another aspect, unimportant for serial programs, but perhaps important for parallel programs, is that some of the structure present in the control state set is obliterated when the equivalent polygenic serial program is formed. Finally, the set of reachable control states may be difficult to determine a priori, opening the possibility of much redundancy in the proof.

When the number of control states is infinite, a

Fig. 9. A simple parallel program.

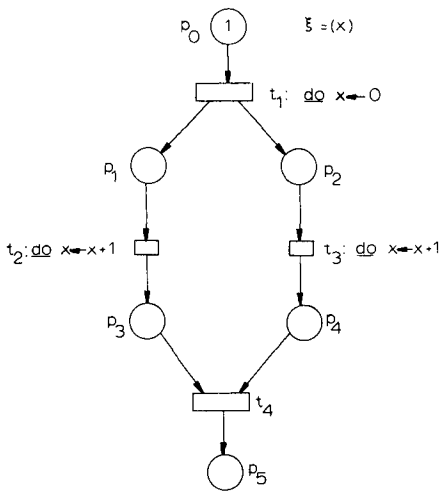
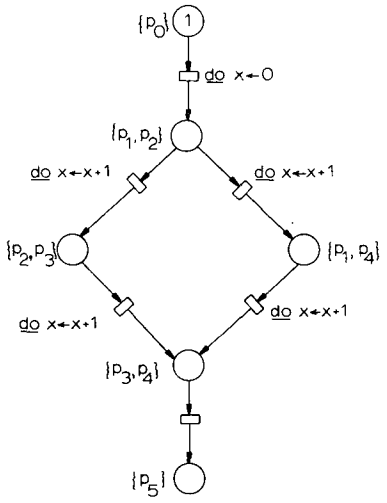


Fig. 10. A polygenic serial program representing the parallel program of Figure 9.



transformation of the form shown in Figures 9 and 10 is impossible. However, by using place variables, we can represent the original parallel program as a polygenic program whose set of data states combines both the control and data states of the original program. Figure 11 shows this for the program of Figure 6. Of course, most of the structure of the original program is obliterated in the process.

The author views it as being unfortunate that the theory of correctness of parallel programs has proceeded mainly by generalization of the assertions method, rather than by using, and perhaps specializing, the induction principle directly. The latter seems to lead to much more succinct characterizations of what is actually being proved.

Ashcroft [2] has presented a special case of the induction principle which assigns assertions to places of the original *parallel* program. In other words, if we represent the control state as the set of places at which instruction pointers are positioned, then the inductive predicate to be proven is

$$J(\gamma;\ \xi) : \bigwedge_{p \in \gamma} A_p(\xi). \tag{4.4}$$

We call this the *parallel place assertions* method. Unfortunately, a predicate of the form (4.4) is generally weaker than the strongest inductive predicate which can be proven. In other words, there are true statements about the states of parallel programs which cannot be expressed in this form. We state this as follows.

PROPOSITION 4.1. *The parallel place assertions method is incomplete, even when restricted to the case that at most one instruction pointer dwells at each place.*[2]

To prove this, we use a simple example. Observing the parallel program of Figure 9, it is clear that the strongest assertion which can be made at each place consistent with a proof that (4.4) is inductive, is

$$A_0 : \text{true}, \quad A_1 : x = 0 \vee x = 1,$$
$$A_2 : x = 0 \vee x = 1, \quad A_3 : x = 1 \vee x = 2,$$
$$A_4 : x = 1 \vee x = 2, \quad A_5 : x = 1 \vee x = 2.$$

The reader may wish to show that, after substituting the above expressions into (4.4), the resulting predicate is indeed inductive with respect to the initial state shown.

We now show that no stronger predicate can be proved, unless it is not required that the predicate be of the form (4.4). This is the case for $p \in \{p_0, p_1, p_2, p_3, p_4\}$ because, based purely on whether $p \in \gamma$, $A_p(\xi)$ is all that can be said. In other words, knowing that $p_3 \in \gamma$ only allows us to conclude that $x = 1$ or $x = 2$, depending on whether $p_2 \in \gamma$ or not. We then have that A_5 is the strongest possible assertion at p_5, it being the conjunction of the assertions A_3, A_4, as indicated by the verification condition for t_4. However it is the case that the stronger assertion

$$x = 2$$

holds at p_5, and this can readily be proved by induction, e.g. by constructing the equivalent polygenic serial program, and using assertions, or by proving the following predicate to be inductive (assuming $x_0 = 0$):

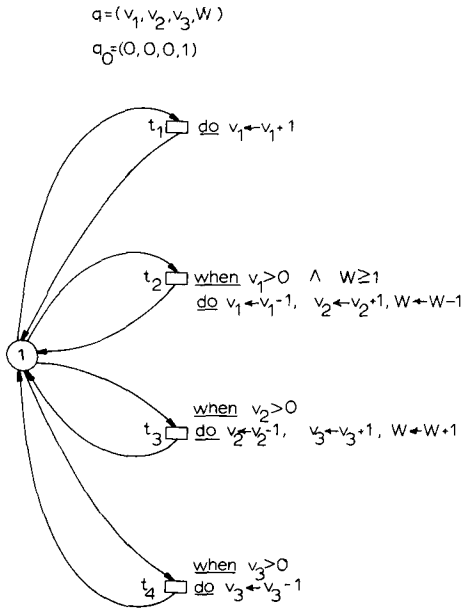
$$2v_0 + v_1 + v_2 + x = 2 \wedge 2v_5 + v_3 + v_4 = x.$$

5. Termination and Deadlocks

The discussion in the previous section has involved the use of the induction principle to show that certain relations among program variables and place variables must always hold. We demonstrated the usefulness of this approach in the case where mutual exclusion is interpreted as the correctness condition. However, there are other conditions which might be required before a parallel program can be considered correct.

[2] See, however, footnote 3, p. 380.

Fig. 11. Another representation of the parallel program shown in Figure 6.



In some cases, it is desirable that a parallel program always *terminate* for certain initial values. This is the case with a "problem" program, designed to complete a specific task. On the other hand, many parallel programs, such as those found in operating systems, or, at least, abstractions of such programs, are designed *not* to terminate, or to terminate only in abnormal situations. One might then wish to prove that a parallel program *never* (rather than *always*) terminates. That is, the program in question serves as a "resource allocator" for other programs, and is assigned the task of insuring that these programs do not permanently usurp resources. (In our formalism, resources can be represented by program variables.)

We consider here programs in which termination is undesirable. The complementary case is discussed in the Appendix.

Let us define *active* to be the disjunction of the enabling predicates for each transition. Hence *active*(q) holds when at least one transition is enabled in state q. So the property that a program never terminates when started in state q_0 can be proved as

active is q_0-invariant.

Unfortunately, this condition is probably too coarse to be useful, for the following reason. It is possible for *active* to be q_0-invariant, while the progress of one or more processes still becomes impeded, and remains so forever. This phenomenon is commonly called *deadlock*.

Several definitions for deadlock have been proposed for more restricted models [cf. 8, 12, 16]. We will argue in the following paragraphs that any means of proving that deadlocks cannot occur can be viewed as proving certain predicates invariant, and thus that such means may be amenable to treatment by the induction principle.

One approach to deadlock analysis involves the selection of a certain set of *key transitions*. It is desirable that these transitions always be capable of becoming enabled, possibly after some other sequence of transitions has occurred. In other words, if P_t is the enabling predicate for transition name t, we can define *live*$_t(q)$ to mean

$$(\exists q')q \rightarrow^* q' \quad \text{and} \quad P_t(q').$$

Then the desirable property, which represents the absence of potential deadlocks, is that for each key transition name t, the predicate *live*$_t$ must be q_0-invariant, where q_0 is the initial state as usual. This condition is a refinement of the notions of "safe" or "secure" as in [8, 12].

Before giving an example of the proof of the in-invariance of *live*$_t$, we digress slightly.

In the terminology introduced in the Appendix, *live*$_t(q)$ can be stated as "P_t is q-reachable." This is the same as saying that $\neg P_t$ is not q-invariant. Although we have an inductive method for proving a predicate invariant, we have no apparent way of proving a predicate not invariant, short of actually exhibiting a sequence of states from q to some q' such that $P_t(q')$ holds. For example, if we consider the program in Figure 7, it is clear that for each transition name t, there is a state q reachable from $q_0 = (0,0,0,0,0,0,0)$ such that $P_t(q)$ holds.

As seen in the Appendix, we can express a theorem for noninvariance. However, in view of the difficulty in showing the nonexistence of the predicate in question, this result may not be of much use.

Unfortunately, proving the q-reachability of P_t is only secondary to the proof of the invariance of *live*$_t$. Another approach is to express *live*$_t$ as a logical formula involving only the program variables, and then applying the induction principle. However we see no way of deriving such an expression for programs in general. Consequently, to prove the invariance of *live*$_t$ for specific programs, it is useful to attempt to use various heuristics. A discussion of some possibilities follows.

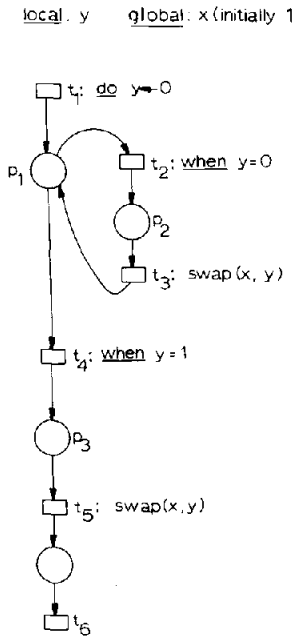
An intuitively attractive way to show that *live*$_t$ is q_0-invariant is to show the existence of a state q_H, and an associated predicate, *homing*$_{q_H}$ defined by *homing*$_{q_H}(q) : q \rightarrow^* q_H$ and to demonstrate that

(i) *homing*$_{q_H}$ is q_0-invariant,

(ii) for each key transition t, *live*$_t(q_H)$.

A state q_H having property (*i*)is called a *home state*. (A similar, but not identical concept of a home state was mentioned in an independent study by H.C. Lauer [26].) In other words, (i) says that no matter what state the system goes to, it can always get to state q_H while (ii) says that from q_H a state can be reached in which t becomes enabled. It is easy to see that the following holds.

Fig. 12. Parallel program with arbitrary number of instruction pointers and local variables; "swap (x, y)" means interchange the values of x and y.



PROPOSITION 5.1. *If there exists a q_H such that (i) and (ii) above hold, then $live_t$ is q_0-invariant.*

Like some other predicates discussed above, *$homing_{q_H}$* is not easy to characterize, but in some cases which we have examined, it appears to be more natural to prove. We illustrate this approach on the readers/writers program of Figure 7.

We assume that the invariance of the mutual exclusion predicate $J_1(v_1, v_2, v_3, v_4, v_5, v_6, R)$ given in (3.4) has already been shown, as discussed earlier. As a home state q_H, we select the initial state $q_0 = (0,0,0,0,0,0,0)$. It was previously mentioned that for each transition name $t_1, \ldots, t_8$ we can show $live_t(q_0)$, simply by constructing a sequence of states which enables the appropriate transition. For example, to see that $live_{t_7}(q_0)$, observe that

$$q_0 \to^{t_5} (0,0,0,1,0,0,0)$$
$$\to^{t_6} (0,0,0,0,1,0,-1)$$
$$\to^{t_7} (0,0,0,0,0,1,-1).$$

Although not so here, it is very likely that the enumeration of a transition sequence to show the reachability of a home state will generally be tedious to construct. Hence we propose another method, which can be considered a generalization of Floyd's technique for proving termination [7].

Given a transition system $(Q, \to)$, we say that a function $\nu: Q \to \omega$ (any well-ordered set will do in place of ω) is a *norm with zero-state q_H* provided that

(i) $\nu(q) = 0$ iff $q = q_H$,

(ii) $(\forall q \in Q)\ \nu(q) \neq 0 \Rightarrow$
$(\exists q')[\nu(q') < \nu(q) \quad \text{and} \quad q \to^* q']$.

We can easily prove

PROPOSITION 5.2. *If a norm with zero-state q_H exists, then q_H is a home state.*

Now to show that *$homing_{q_0}$* is q_0-invariant for the previous example, we claim that the function $\nu : Q \to \omega$ defined by

$$\nu(v_1, v_2, v_3, v_4, v_5, v_6, R) =$$
$$v_1 + v_2 + v_3 + v_4 + v_5 + v_6 + |R|$$

(where $|R|$ is the absolute value of R) is a norm with respect to q_0. We will assume the invariance of J_1, as defined and proved in Section 3, to show this. Suppose $\nu(q) > 0$, where $q = (v_1, v_2, v_3, v_4, v_5, v_6, R)$. We consider separately the cases $R = 0$, $R < 0$, $R \geq 0$. In each case, a sequence of transitions can be exhbited which takes q to q', and $\nu(q') < \nu(q)$. For example, if $R < 0$ then from J_1, $v_5 = 1$. Hence executing t_7 produces q' with the desired property. The other cases are dealt with similarly.

Note that in this example, the induction principle was used only indirectly to prove absence of deadlocks. That is, the proof relied on the invariance of a previously proven predicate. The reader may note that the proof may not be carried through without assuming the previously proven predicate.

6. Extensions

The examples presented in previous sections were simple, in the interest of demonstrating as clearly as possible the principles being used for proofs. This section points out methods for extending the previous techniques to more complex examples.

First, it is clear that we can add to any parallel program as many *auxiliary variables* as we wish. These variables can be affected by the action of transitions, so long as they do not appear in enabling predicates, so that the behavior of the program remains unaffected.[3] An interesting example of such a variable would be the introduction of a "time clock" variable, τ. Each action would be augmented by an action $\tau \leftarrow \tau + 1$. In addition, any action of a transition t could set some other auxiliary variable u_t to the value of τ, indicating the "time" at which this action last occurred. Of course we do not pretend that τ represents real time, but the values of variables u_t for two or more transitions could be used to advantage to compare the *relative* ordering of events.

Concerning the use of a single invariant to indicate correctness, we showed in a previous section that the ability to decompose the invariant into "assertions" is not lost. However another type of decomposition appears to be of some use. We have in mind the formula-

[3] A presentation by S. Owicki (June, 1975) has brought us to realize that Proposition 4.1 does not hold when we allow the introduction of arbitrary auxiliary variables. In this case, we can simulate the action of transitions on place variables.

tion of a series of invariants, $J_1, J_2, \ldots$. We begin by proving J_1. Then to prove the invariance J_i, $i > 1$, inductively, it may be helpful to assume $J_1, \ldots, J_{i-1}$. Indeed, this was the technique used for proving the correctness of the readers/writers program in the previous section. Although this approach is useful in structuring the proof, it is not essential, because we could just as well have attempted to prove $\bigwedge J_i$ at the outset.

The stepwise decomposition described above may prove very powerful in conjunction with a technique which removes transitions as we progress from step to step. For example, at one stage i we prove an invariant J_i which amounts to saying that mutual exclusion holds among certain places. At the next stage, we can then remove the mechanism (i.e. the transitions) responsible for insuring mutual exclusion. We replace this mechanism with a predicate Φ_i on the state set abstracted from J_i, which expresses that mutual exclusion holds. Then we assume that a transition $q \rightarrow q'$ can occur only in case $\Phi_i(q')$ holds. Thus Φ_i may be assumed in the proof of an invariant J_{i+1}. Special cases related to this approach are the "constraints" of [2] and the "reduction" technique of [21].

To present a further extension, it was mentioned earlier that the model of Section 2 does not attempt to distinguish one process from another. In cases where this representation is accurate, proofs are simplified, because the state of the system is representable by the *number* of processes whose instruction pointers reside at a particular place, rather than *which* processes have this property.

There are, however, cases in which this representation is not accurate. An outstanding example is that in which each process has its own set of local variables, in addition to the global variables which are shared by all processes. This case may be represented in the model we have presented, if the number of processes is fixed a priori. In this case, we treat all variables as being potentially global, but some of them are just not accessed by some processes.

In the interest of economy in representation greater than such a scheme provides, and also to allow an *arbitrary* number of dynamically created processes, each being allocated its own set of local variables, we present the next extension. An example of a system where this can occur is with a PL/I asynchronously called procedure [13].

We consider the same graph representation for parallel programs as before, except that certain variables are distinguished as *local*, while others are *global*. We can then assume a sequence of *process indices* $\pi_1, \pi_2, \pi_3, \ldots$. The state is given by

(i) for each global variable x, the value assigned to x;
(ii) for each local variable y, and each process index π, the value assigned to $y(\pi)$;
(iii) for each process index π, the place $i(\pi)$ at which its instruction pointer dwells.

We avoid the programming language issues of the time at which local variables get allocated, and of more complicated hierarchies of global and local variables as would be present, say, with "block structure." Of course, place variables could be retained as auxiliary variables, i.e. $\nu(p)$ is just the cardinality of the set $\{\pi \mid i(\pi) = p\}$.

An invariant in this extension is a predicate on the global, local, and instruction pointer variables, where the last two have qualifying process indices attached. These indices can either be fixed to represent a particular process, or free to represent quantification over all processes. For each process π, the transition name t in the program is qualified by π. The induction step requires that the predicate be shown invariant for each $t(\pi)$. But this requires little more work than in the original version. We simply have to consider the effect of $t(\pi)$ on global variables and local variables $x(\pi)$, but not on $x(\pi')$ where $\pi' \neq \pi$, since the latter are unaffected.

An example is provided by the program of Figure 12. This program demonstrates an implementation of a mutual exclusion mechanism adopted from [6]. The variable x is global, but y is local. Thus there is a variable $y(\pi)$ for each process π. The initial state has $x = 1$ and no processes. The predicate whose invariance is to be shown is

$$J_5 : \pi_1 \neq \pi_2 \Rightarrow [(i(\pi_1) \neq p_3) \vee (i(\pi_2) \neq p_3)].$$

That is, there is no pair of distinct processes whose instruction pointers dwell at p_3 simultaneously. This is conveniently proved by showing the following invariants.

$$J_1 : (i(\pi) = p_3) \Rightarrow (y(\pi) = 1),$$
$$J_2 : (x + \sum_{\pi} y(\pi)) = 1,$$
$$J_3 : y(\pi) \in \{0, 1\} \wedge x \in \{0, 1\},$$
$$J_4 : (\pi_1 \neq \pi_2) \Rightarrow [(y(\pi_1) \neq 1) \vee (y(\pi_2) \neq 1)].$$

It is clear that J_5 is a tautological consequence of J_1 and J_4.

PROOF OF J_1. J_1 is true for the initial state. Furthermore, we need only show that J_1 is preserved by transitions $q \rightarrow q'$ defined by $t_4(\pi)$, for only then can $J_1(q')$ be false. We see that $y'(\pi) = y(\pi)$, however, and since $y(\pi) = 1$ is the enabling condition for $t_4(\pi)$, $y'(\pi) = 1$ and $J_1(q')$ holds.

PROOF OF J_2. Clearly J_2 holds for the initial state and is preserved by all transitions.

PROOF OF J_3. Same observation as for J_2.

Finally, J_4 is a direct consequence of J_2 and J_3.

The technique described appears to have the very desirable property that the effort required for proofs of systems with multiple processes increases only with the size of the program rather than the number of processes executing the program.

It is also surprising that the number of invariants proved is not greater than the number of assertions which would result from assigning one assertion per place. That is, the fact that invariants rather than asser-

tions are used does not mean that the proof is bound to be more complicated.

We now demonstrate a further use of local variables. Our view of semaphores up to this point has been abstract. An operating system which employs semaphores is likely to use an implementation which is somewhat more constrained than the abstract model would suggest. Typically, for example, processes which are temporarily disabled because of a wait on a semaphore will become re-enabled in a first in–first out order. That is, a queue is maintained which maintains the identity of the processes in the order in which they do the P-operation. It is clear that a queue data structure can be modeled in our formalism. However we propose an alternate representation which is likely to yield simpler proofs. This scheme employs local variables and is shown in Figure 13. The idea is that when a process π does the P-operation, w is incremented, then local variable $n(\pi)$ is set to the current value of w. The process then waits until variable s exceeds $n(\pi)$. The latter may hold initially, if the initial value of the semaphore is large enough. Otherwise, it will hold when sufficiently many V-operations have been performed. It is now reasonable to ask how well our previous, more abstract, representation of semaphores models the implementation version. This topic will be allowed to remain for future investigation.

When we identify processes explicitly, the previous formulation of deadlock-freedom is too coarse. We wish to modify that definition so that a process can terminate itself without this being considered as deadlock. One way of handling this is to adopt the convention that certain transitions $t(\pi)$, e.g. those transitions represented in the program by a transition node which is connected to no place, are defined to be *exit* transitions. We then define $live_\pi(q)$ for a process π and state q to hold iff either there exists a q' such that $q \rightarrow^* q'$ and some $t(\pi)$ is enabled in q', or π has executed an exit transition before state q is reached.

7. Discussion and Conclusions

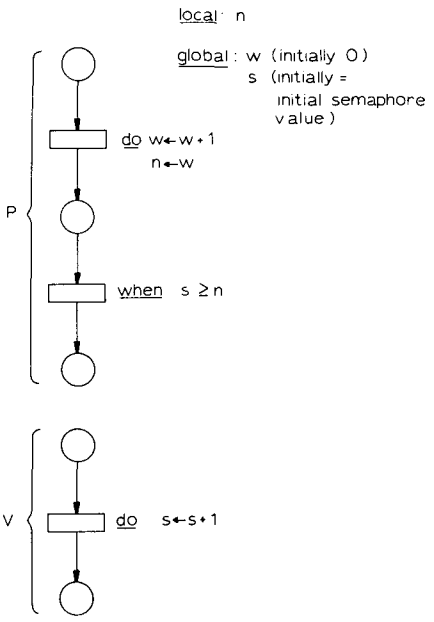
We have presented an abstract model for parallel programs, and within it an induction principle, and demonstrated its usefulness in proving correctness. Instances of this principle have appeared in various forms before [1, 2, 3, 6, 7], but we feel that our model allows it to be stated somewhat more succinctly than usual. It is hoped that such a model will thus contribute to the unification of parallel program models and verification techniques for them.

We also presented a model for the presentation of parallel programs. This model allows an arbitrary number of processes to effectively be executing the same program. The semantics of the model can be discussed without any prerequisite of "legality" [1, 2]. Another advantage of the model is that it is consistent with other more restrictive models. For example, certain parallel schemata [14, 15] are instances of the present model with part of the interpretation missing. An even further abstraction involves making the predicate and function labels trivial, stripping the program down to a bare control structure. In this case we have a Petri net [11, 16]. A model such as the one considered here then provides a natural motivation for the consideration of parallel program schemata, Petri nets, etc. An extension of the model was presented which provided for an arbitrary number of dynamically created processes to have their own local variables, as well as have global variables. It was also shown that the techniques presented naturally include the familiar "assertions" method for serial programs. Hence no generality is lost.

We have found the concept of place variable to be very *suggestive* when faced with the construction of proofs. The examples whose correctness we have proved in this paper were simple ones, chosen to serve the interest of clarity. The author is confident that the techniques are applicable to more complex examples. These include, for example, more elaborate interpretations of the semaphore concept, which involve implementation.

One can consider the relation between certain invariants discussed here and these other models. For example, does the invariance of the predicate $live_t$ become decidable if we remove the predicates and functions? Unfortunately, this question is open. We do know, on the other hand, that there is a decision procedure for the invariance of any predicate J, where the negation $\neg J$ has a certain monotonicity property and where J is computable in a special sense. (cf. [18]).

Fig. 13. A scheme for representing first in–first out semaphore operations.



For further research, we would like to suggest the investigation of restricted classes of the representation used here, and techniques for successively refining representations to simplify proofs. In addition, work is needed in answering questions about what the relevant correctness conditions are. We have suggested some possibilities in this paper, but there are undoubtedly other aspects, especially those dealing with implementation, on which we have not touched. Finally, we have little understanding of the process of proceeding from the desired invariant to an invariant strong enough to be proved by the induction principle. It appears, for example, that the ease in making this step relates directly to the degree of structure and understandability of the program itself.

Appendix

We now present additional relations between the concepts of invariant predicates, inductive predicates, and certain other predicates. We will point out the relation between such predicates and termination of parallel programs, and the relation of our work to the satisfiability of certain logical formulas, such as are discussed in [1]. Our aim is to present some definitional suggestions, and to present extensions and more succinct explanations of some results in the work cited.

Let $(q, \rightarrow)$ be a transition system with $q_0 \in Q$.

Definition 1. Let R be the predicate such that $R(q)$ iff $q_0 \rightarrow^* q$.

Thus "J is q_0-invariant" may be simply written "$R \Rightarrow J$."

PROPOSITION 1. *R is q_0-inductive.*

PROOF. Follows directly from the definition of $\rightarrow^*$.

PROPOSITION 2. $R \Rightarrow K$ iff there is a q_0-inductive predicate J such that $J \Rightarrow K$.

PROOF. If $R \Rightarrow K$ then choose J to be R and use proposition 1. Conversely, if J is q_0-inductive then $R \Rightarrow J$, and combining this with $J \Rightarrow K$, we have $R \Rightarrow K$.

Remark. Since for certain transition systems, e.g. a parallel program with a finite set of transition names, "J is q_0-inductive" can be expressed as a logical formula, say $Ind(J)$, we can summarize the above by

$$R \Rightarrow K \text{ iff } (\exists J)(Ind(J) \wedge J \Rightarrow K)$$

In other words, $R \Rightarrow K$ *iff* the formula $Ind(J) \wedge J \Rightarrow K$ with predicate variable J is *satisfiable*.

We introduce the following auxiliary concept for comparison later in the exposition.

Definition 2. A unary predicate K is said to be *q_0-reachable* if there exists a q such that $q_0 \rightarrow^* q$ and $K(q)$.

PROPOSITION 3. *K is q_0-reachable iff $\neg K$ is not q_0-invariant.*

We now continue the discussion of termination begun in Section 5. We mentioned there that a program could be shown *never* to terminate by showing that *active* is invariant. Now suppose that we want to show that a program *always* terminates. A little thought will show that this property cannot be expressed as an invariant or non-invariant.[4] What we need is another concept, which is expressed below.

Definition 4. A predicate J is called *q_0-subinvariant* if there exists an infinite sequence of states $q_0, q_1, q_2, q_3, \ldots$ such that $(\forall i) q_i \rightarrow q_{i+1}$ and such that $(\forall i) J(q_i)$.

Definition 5. A predicate J is called *q_0-inevitable* if $\neg J$ is not q_0-subinvariant.

For example, every computation terminates iff there is no infinite state sequence in which *active*(q) is true for every state q[5] iff $\neg$*active* is q_0-inevitable.

Analogous to the induction principle, there is a "subinduction" principle for proving subinvariants.

Definition 6. *J is called q_0-subinductive if*

$$J(q_0) \wedge (\forall q)[J(q) \Rightarrow (\exists q')(q \rightarrow q' \wedge J(q'))].$$

PROPOSITION 4. *If J is q_0-subinductive, then J is q_0-subinvariant.*

PROPOSITION 5. *K is q_0-subinvariant iff there exists a J such that $J \Rightarrow K$ and J is q_0-subinductive.*

Thus K is q_0-subinvariant iff the formula

$$\text{subind}(J) \wedge J \Rightarrow K$$

is *satisfiable* by some J, where subind(J) is the expression in Definition 6.

The following diagram expresses the relationship between the terms we have introduced.

K is q_0-invariant	K is q_0-subinvariant
$\Updownarrow$	$\Updownarrow$
$\neg K$ is not q_0-reachable	$\neg K$ is not q_0-inevitable
$\Updownarrow$	$\Updownarrow$
$(\exists J) J \Rightarrow K$ and J is q_0-inductive	$(\exists J) J \Rightarrow K$ and J is q_0-subinductive

We can then conclude that a program always terminates iff there does *not* exist a J such that J is subinductive and $J \Rightarrow$ *active*. Whether this result is of practical use depends on the difficulty in showing a predicate to be unsatisfiable, and this depends on the functions and predicates used in the program. We may be better off in reverting to heuristic techniques of the form discussed in Section 5 in order to show that a program always terminates.

We now mention a somewhat subtle concept, which can complicate consideration of termination and related properties. As we have been discussing, any infinite state sequence $q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow \ldots$ is considered a computation. However, there may be a reason to exclude some such sequences. For example, if for some

[4] An exception is the case of a monogenic program. Then there is only one computation, so either every computation terminates or every computation does not. There is no third possibility. Thus a monogenic program always terminates iff *active* is not invariant.

[5] The observant reader will note that *active* can be replaced with *true*.

k we have a transition t enabled in $q_k, q_{k+1}, q_{k+2}, \ldots$, but for each i $q_i \rightarrow^t q_i'$, where $\neg active(q_i')$, then it might be considered improper to say that there is an infinite computation, because the infinite sequence exists by virtue of t not occurring. If we require that all computations satisfy a "finite-delay property" [14, 15], namely that a transition cannot be enabled forever without actually occurring, then such a sequence would not stand as a computation. The point we wish to make is that conditions relating to termination are not characterizable as previously indicated when the finite-delay property is required. We leave further consideration of this question to future investigation.

A related question is that of *livelock*, to use the term of [2]. It is noted that our consideration of the absence of deadlock involved proving the *existence* of certain state sequences. In other words, deadlock can always be avoided by the execution of a certain sequence of transitions from a given state. However, it may be the case that, even though deadlock is impossible, certain valid computations exclude the execution of a certain key transition, and this may happen whether or not the finite-delay property is satisfied. The reader can observe that this occurs in the example of Figure 12, as pointed out in [6]. This condition can seemingly be related to the concept of *inevitability* described earlier. That is, we wish to show that in each state q, the enabling of certain key transitions is q-inevitable. Unfortunately, the property of being a key transition in this sense may be one which varies depending on q. Once again, we leave the question of how to best prove the absence of livelock to future investigation.

Whether or not a program is designed to terminate, there may be associated the notion of *determinacy*. This means all possible computations (i.e. state-sequences proceeding from a given initial state) are equivalent modulo some equivalence relation. The equivalence relation to be used depends on which version of determinancy one wishes to adopt. For example, it could be required that all reachable halting states have the same values of certain data variables, or that all computations are halting, or all computations are halting or all are nonhalting.

Determinacy, then, says that all computations are equivalent, but does not necessarily say what a computation *does*, i.e. what invariant holds for all states. The advantage of considering determinacy is that, by examining only *one* computation, conclusions can be drawn for all computations. It may even be the case that a determinate parallel program can be transformed into an equivalent serial program and properties proved for the latter can be used to derive properties of the former. The most useful approaches in this direction seem to be sufficient conditions for determinacy which are easily tested [14, 15]. The presentation of a second-order formula whose truth expresses determinacy [1] is not likely to be of much practical interest. Unfortunately, the equivalence relations for which determinacy is easily expressed are not necessarily the most useful and conversely, the proper notion of determinacy is not terribly apparent in some interesting practical problems. For example, the notion of determinacy for an airline reservation system [2] appears difficult to capture.

Acknowledgments. The author wishes to thank two anonymous referees for their critical evaluation, Y.S. Kwong for his reading of the manuscript, and Elaine Weyuker and James Storer for pointing out errors. Thanks also to Thomas Doeppner for several conversations which influenced the formulation of programs with local variables.

Received April 1974; revised July 1975

References

1. Ashcroft, E., and Manna, Z. Formalization of properties of parallel programs. *Machine Intelligence 6* (1970) 17–41.
2. Ashcroft, E.A. Proving assertions about parallel programs. *J. Comp. Sys. Sci. 10*, 1 (Jan. 1975), 110–135.
3. Brinch Hansen, P. A comparison of two synchronizing concepts. *Acta Informatica 1* (1972), 190–199.
4. Conway, M. A multiprocessor system design. AFIPS Conf. Proc., Vol. 24, AFIPS Press, Montvale, N.J., 1963, pp. 139–148.
5. Courtois, P.J., Heymans, R., and Parnas, D.L. Concurrent control with readers and writers. *Comm. ACM 14*, 10 (Oct. 1971), 667–668.
6. Dijkstra, E.W. Hierarchical ordering of sequential processes. *Acta Informatica 1* (1971), 115–138.
7. Floyd, R.W. Assigning meanings to programs. *Proc. Symp. in Appl. Math*, Vol. 19, Amer. Math. Soc., Provincetown, R.I., 1967, pp. 19–32.
8. Habermann, A.N. Prevention of system deadlocks. *Comm. ACM 12*, 7 (July 1969), 373–377.
9. Habermann, A.N. Synchronization of communicating processes. *Comm. ACM 15*, 3 (March 1972), 177–184.
10. Hoare, C.A.R. Towards a theory of parallel programming. In *Operating Systems Techniques*, Hoare and Perrot (Eds.), Academic Press, New York, 1972, pp. 61–71.
11. Holt, A., and Commoner, F. Events and conditions. Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, June 1970, pp. 3–52.
12. Holt, R.C. On deadlock in computer systems. Tech. Rep. CSRG-6, Computer Systems Research Group, U. of Toronto, Apil 1971.
13. IBM PL/I Reference Manual, Form C28-8201-1, March 1968.
14. Karp, R.M., and Miller, R.E. Parallel program schemata, *J. Computer Sci. 3* (May 1969), 147–195.
15. Keller, R.M. Parallel program schemata and maximal parallelism. *J. ACM 20*, 3 (July 1973), 514–537; and *J. ACM 20*, 4 (Oct. 1973), 696–710.
16. Keller, R.M. Vector replacement systems: a formalism for modeling asynchronous systems. TR 117, Computer Sci. Lab., Dep. of Electrical Eng., Princeton U., Dec. 1972 (revised Jan. 1974).
17. Keller, R.M. A fundamental theorem of asynchronous parallel computation, In *Parallel Processing*, T.Y. Feng (Ed.), Springer-Verlag, Berlin, 1975.
18. Keller, R.M. Generalized Petri nets as models for system verification (to appear).
19. Lauer, H.C. Correctness in operating systems, Ph.D. Th., Carnegie-Mellon U., Sept. 1972.
20. Levitt, K.N. The application of program-proving techniques to the verification of synchronization processes, AFIPS Conference Proc., Vol. 41, 1972 FJCC, AFIPS Press, Montvale, N.J., 1972, pp. 33–47.
21. Lipton, R.J. Reduction: A method of proving properties of systems of processes. Research Rep. No. 40, Yale U. Dep. of Computer Sci., March 1975.